Multithreading in Flight Reservation System

Abhishek Mahakal[1*] and Vedant Lanjewar[1*†]

[1*]Computer Science, Pennsylvania State University, 777 W Harrisburg Pike, Harrisburg, 17057, Pennsylvania, USA.

*Corresponding author(s). E-mail(s): amm10344@psu.edu; vsl5052@psu.edu;

†These authors contributed equally to this work.

Abstract

This report discusses the use of multi-threading in an airline reservation system to improve the data transfer process. The system typically involves a large amount of data being transmitted through various servers, which can result in slower transfer rates. The report proposes the use of multiple threads that can perform actions simultaneously, allowing for faster data transfer. To test the effectiveness of this approach, two remote database servers located in W. Virginia were created and tested with and without multi-threading. The results show that multi-threading significantly improves the data transfer process, with an 81.19% increase in speed when fetching 327680 documents. This project demonstrates the potential benefits of using multi-threading in airline reservation systems to improve data transfer rates.

Keywords: Multi-threading, remote database server, airline reservation

1 Introduction

An airline reservation system is a complex system that involves multiple data transfers across various entities. It is a crucial component of the airline industry as it enables customers to book flights, make payments, and manage their itineraries. At its core, the system is designed to facilitate the flow of information between the airline and its customers, as well as between the airline and other third-party service providers.

The airline reservation system is composed of several subsystems that work together to manage the various aspects of the reservation process. For example, the system must handle flight schedules, seat availability, and pricing information. Additionally, it must process payment transactions and issue electronic tickets.

One of the primary challenges of the airline reservation system is managing the massive amounts of data that are generated by the system. This data includes customer information, flight schedules, seat availability, pricing information, and payment transactions. To handle this data, the system must be designed to efficiently store, process, and transmit information.

In order to do so we have determined that incorporating multithreading is a viable solution. Multithreading allows for multiple tasks to be executed simultaneously, thereby reducing the processing time required to retrieve data from the back-end.

To implement multithreading, we have opted to use Node.js, a popular open-source JavaScript runtime environment that enables the creation of server-side applications. Node.js offers several advantages, including its ability to handle large numbers of requests, its support for asynchronous programming, and its compatibility with many popular databases.

To create worker threads, we are utilizing Node.js' built-in "worker-threads" module. This module provides an easy way to create new threads that can execute tasks concurrently. By assigning each worker thread to access a different database, we can retrieve data from multiple sources in parallel, further reducing the processing time required.

[1]Threads operate similarly to processes in that they possess their own instruction pointer and can execute one JavaScript task at a time. However, unlike processes, threads do not possess their own memory but rather reside within a process's memory. Using the worker-threads module, a process can be created with multiple threads that can execute JavaScript code in parallel. Additionally, threads are capable of communicating with each other through message passing or by sharing data in the process's memory. Because of this, threads are lightweight compared to processes since the creation of a thread does not require additional memory from the operating system.

[1]The libuv library, which is utilized by Node.js, provides four supplementary threads to each Node.js process. These threads are responsible for handling I/O operations separately, and once they are completed, the event loop adds the corresponding callback to a microtask queue. When the main thread's call stack is clear, the callback is placed on the call stack and then executed. It is important to note that although the I/O task of reading a file

or a network request is performed in parallel by these threads, the callback associated with the given I/O task does not execute in parallel. Once the I/O task is completed, the callback runs in the main thread.

Aside from the four threads provided by libuv, the V8 engine also provides two threads that manage automatic garbage collection and other related tasks. As a result, each process has a total of seven threads: one main thread, four Node.js threads, and two V8 threads.

To store the documents we used MongoDB database. MongoDB is a type of database that is focused on documents, which are similar in structure to JSON. This document-oriented approach makes it simpler to manage and arrange data that lacks a fixed schema, which is particularly advantageous for the airline industry where data arrives in varying formats and flight schedules and pricing are subject to frequent modifications. MongoDB empowers developers to adjust rapidly to these changes by storing data as it is received, without the need to be concerned about predefined schemas.

2 Literature Survey

2.1 How To Use Multithreading in Node.js

[1] Describes worker-thread module in Node.js enables the creation of threads for executing numerous JavaScript tasks simultaneously. When a thread completes a task, it relays the output to the main thread for use in other parts of the program. The benefit of using worker threads is that CPU-bound tasks can be distributed among multiple workers, freeing the main thread from blocking. This approach is useful when dealing with Node.js applications that have CPU-intensive tasks that may obstruct the main thread, and the ”worker-thread” module allows for offloading these tasks to a separate thread to avoid blocking.

2.2 Handlebars and MongoDB with Express

[2] The article explains the process of integrating Handlebars view engine and MongoDB with Express in Node.js, a web development framework. Using Express, developers can quickly generate the skeleton of a web application. The Express app follows a Model-View-Controller (MVC) architecture, which is reflected in the folder structure of the app directory. Handlebars is a JavaScript library that enables the creation of reusable webpage templates consisting of HTML, text, and expressions. The expressions, enclosed in double curly braces, are embedded within the HTML document.

2.3 Getting Started With NodeJs MongoDB

[3] The article describes MongoDB is a versatile, document-oriented database that delivers exceptional performance and scalability. The database operates

on the principles of collections and documents. Written in C++, MongoDB is a NoSQL database. A collection in MongoDB is a grouping of documents, each of which can have varying fields. Typically, all documents in a collection serve a common purpose. In MongoDB, a document is comprised of key-value pairs, and documents possess a dynamic schema. This implies that documents in the same collection are not required to have the same structure or set of fields. Integrating Node.js with MongoDB is crucial for modern applications that require large data management, rapid feature development, and flexible deployment. Traditional databases struggle to support these requirements, while MongoDB provides scalability, high availability, and the ability to scale from single-server deployments to complex, multi-site architectures.

3 Methodology

3.1 Server Setup

The project consists of 2 remote servers and a client. The 2 remote servers are the airline and hotel databases which are hosted in a distant location, in this case, W. Virginia. [3] These databases are handled by MongoDB.The client is the local machine on which the user is accessing the service which is set up using NodeJs Express. This client machine acts as an endpoint that connects to the databases by creating requests on the local machine and forwarding those to the two databases simultaneously via multi-threading. The architecture for our setup is show in figure 1.

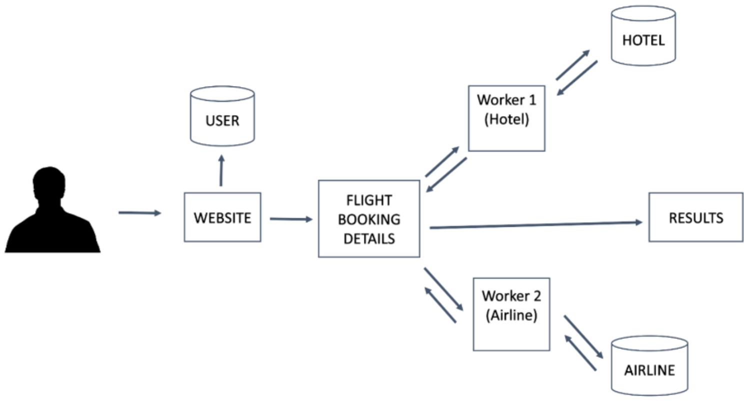


Figure 1: Architecture

3.2 Client Machine

In order for the user to interact with the airline reservation service, a graphical user interface has been created in the form of a website. This website is created using HTML and CSS to give a visual representation as to what is going on. [2] The backend is created using NodeJs which houses the worker-threads module that helps with the task of multi-threading.

The website consists of a login / signup page as shown in figure 3 for the user to log in or register if accessing the service for the first time. The user cannot proceed until he/she has logged in. Once the user has logged in, the session is activated and cookies are stored for that particular user.

On the landing home page as shown in figure 2, the user is prompted to either log out or book tickets. On clicking the logout button, the user is instantly logged out and the current session is destroyed. Whereas on clicking the book tickets button, the user is redirected to a page,figure 4, where he fills in all the details of the flights such as the departure location, arrival location, and date. Once he hits the submit button after making the selection, the request will be forwarded to the databases and the required information is fetched and displayed to the user on the next page, showin in figure 5 and figure 6.

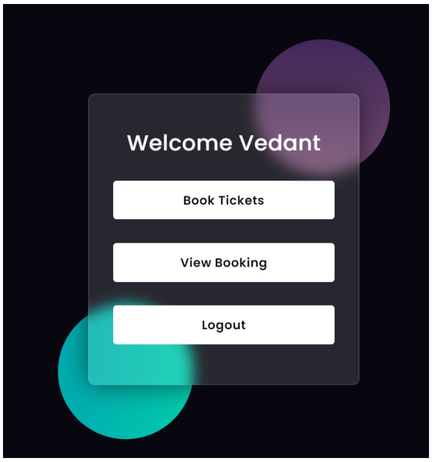


Figure 2: Home Page

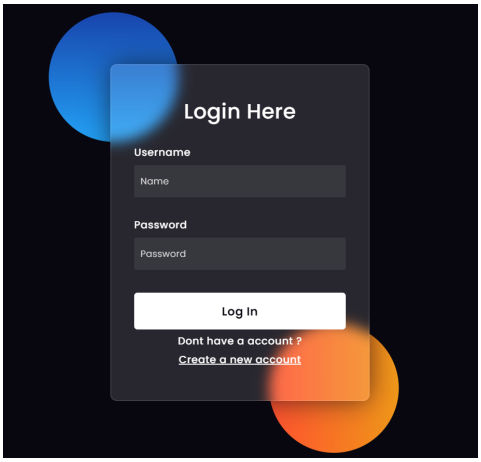
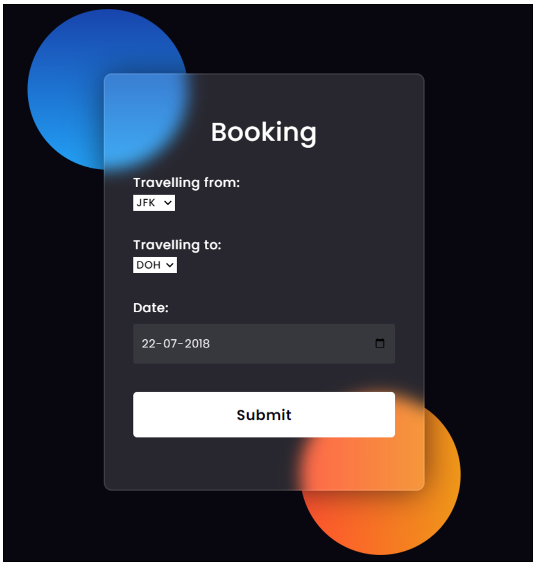


Figure 3: Login Page



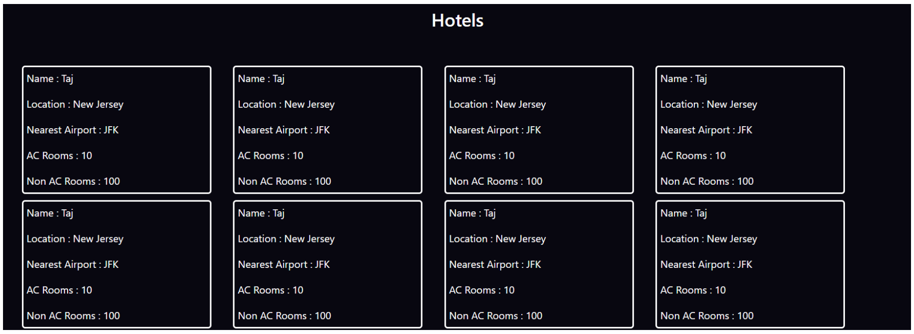
Booking

Travelling from:
JFK

Travelling to:
DOH

Date:
22-07-2018

Figure 4: Booking Page



Hotels

Name : Taj
Location : New Jersey
Nearest Airport : JFK
AC Rooms : 10
Non AC Rooms : 100

Name : Taj
Location : New Jersey
Nearest Airport : JFK
AC Rooms : 10
Non AC Rooms : 100

Name : Taj
Location : New Jersey
Nearest Airport : JFK
AC Rooms : 10
Non AC Rooms : 100

Name : Taj
Location : New Jersey
Nearest Airport : JFK
AC Rooms : 10
Non AC Rooms : 100

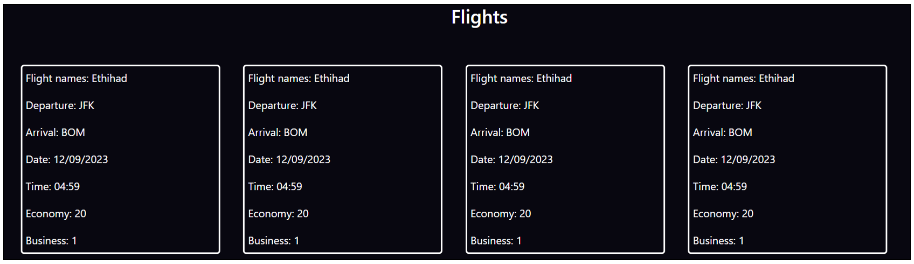
Name : Taj
Location : New Jersey
Nearest Airport : JFK
AC Rooms : 10
Non AC Rooms : 100

Name : Taj
Location : New Jersey
Nearest Airport : JFK
AC Rooms : 10
Non AC Rooms : 100

Name : Taj
Location : New Jersey
Nearest Airport : JFK
AC Rooms : 10
Non AC Rooms : 100

Name : Taj
Location : New Jersey
Nearest Airport : JFK
AC Rooms : 10
Non AC Rooms : 100

Figure 5: Hotel query result



Flights

Flight names: Ethihad
Departure: JFK
Arrival: BOM
Date: 12/09/2023
Time: 04:59
Economy: 20
Business: 1

Flight names: Ethihad
Departure: JFK
Arrival: BOM
Date: 12/09/2023
Time: 04:59
Economy: 20
Business: 1

Flight names: Ethihad
Departure: JFK
Arrival: BOM
Date: 12/09/2023
Time: 04:59
Economy: 20
Business: 1

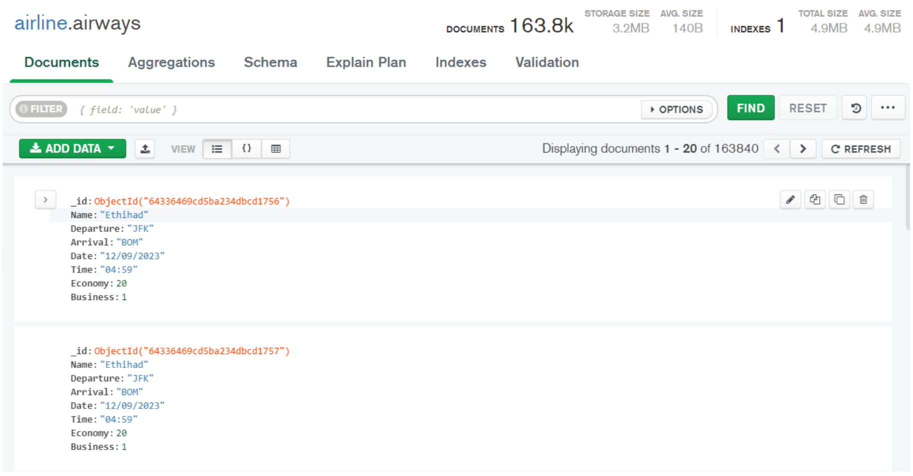
Flight names: Ethihad
Departure: JFK
Arrival: BOM
Date: 12/09/2023
Time: 04:59
Economy: 20
Business: 1

Figure 6: Flight query result

3.3 Databases

3.3.1 Flights Database

The flights' database consists of the details of all the flights with the following attributes: flight name, departure location, arrival location, date, time, and seats (business, first class, economy). Whenever a request arrives from the user machine, the fields traveling from and traveling to from the booking form get matched with the departure location and arrival location of the flights' database. The respective records are sent back to the client machine if such a match is found. The flights' database has a total of 163,840 records. A snapshots of the database is shown in figure 7.



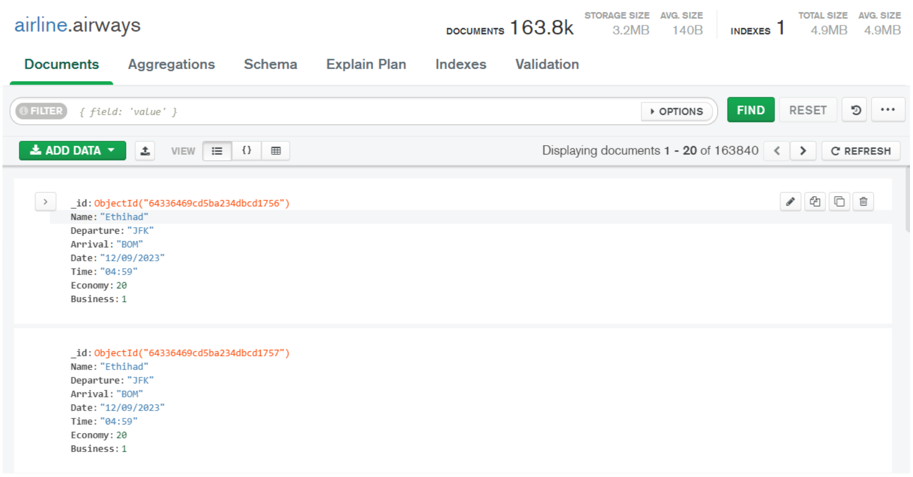
```
_id: ObjectId("64336469cd5ba234dbcd1756")
Name: "Ethihad"
Departure: "JFK"
Arrival: "BOM"
Date: "12/09/2023"
Time: "04:59"
Economy: 20
Business: 1

_id: ObjectId("64336469cd5ba234dbcd1757")
Name: "Ethihad"
Departure: "JFK"
Arrival: "BOM"
Date: "12/09/2023"
Time: "04:59"
Economy: 20
Business: 1
```

Figure 7: Flight's Database entries

3.3.2 Hotel Database

The Hotel database consists of the details of all the hotels with the following attributes: Hotel name, location, nearest airport, ac rooms, non ac rooms. When a request is received at this database, the travelling to field value from the booking form at the client side gets matched with the nearest airport field of Hotel database. The respective hotel records are returned to the client if a match is found. The Hotels' database too has a total of 163,840 records.A snapshots of the database is shown in figure 8.



Displaying documents 1 - 20 of 163840

```
_id: ObjectId("64336469cd5ba234dbcd1756")
Name: "Ethihad"
Departure: "JFK"
Arrival: "BOM"
Date: "12/09/2023"
Time: "04:59"
Economy: 20
Business: 1

_id: ObjectId("64336469cd5ba234dbcd1757")
Name: "Ethihad"
Departure: "JFK"
Arrival: "BOM"
Date: "12/09/2023"
Time: "04:59"
Economy: 20
Business: 1
```

Figure 8: Hotel's Database entries

3.3.3 User Database

This database is of the website which is providing the reservation service to the user. Its task is to store the users' login credentials using fields such as username and password.

3.4 Backend Working

The backend of the website is created using NodeJs. The reason behind this is that it has the worker threads library which can be utilized to implement multithreading. In our case, we are making use of two threads to handle the information retrieval from the flights and hotels databases. There are two separate thread code blocks that take in the request from the user and process it parallelly.

The first worker thread gets the values from the booking form and converts it into a MongoDB query which is forwarded to the flights' database for record fetching. When the thread gets the response from the database, it stores the result in a shared buffer.

Similarly, the second thread is responsible for handling the communication between the client and the hotel database. When this worker thread gets the values from the booking form, the thread converts it into a MongoDB query and passes it to the hotel database. Once it gets the result, the thread stores it in a shared buffer.

Note that the system retrieves data from both databases simultaneously and stores it in a shared buffer that is accessible by any threads created by the process. The shared buffer facilitates the simultaneous storage and retrieval of data, which results in reduced system response time.

Also when we want to pass on the shared buffer from the thread code block to the main code block, the shared buffer needs to be encoded at the thread code block before sending and decoded at the main code block to access the contents within it. Once the buffer is decoded, its contents are displayed to the user on the screen as shown in figure 5 and 6.

We need to encode because the sharedbuffer created is of the type Uint8Array i.e it represents an array of 8-bit unsigned intergers. The results retrieved from the database are in JSON format. So we need to encode the JSON message recieved from the database in order to store the data in the shared buffer, which gets decoded back at the main thread.

4 Results

We experimented with two different scripts for accessing two databases, each containing 163840 documents. One script utilized multi-threading, while the other accessed the database sequentially. To limit the number of documents fetched, we utilized the limit function provided by the MongoDB library. Our focus was to measure the time taken by each script for data fetched, while gradually doubling the amount of data for each test. By comparing the results, we aimed to determine the most efficient way to access and fetch data from large databases. The statistics collected by us during the experimentation is show in table 1 and its corresponding graph is show in figure 9.

Records fetching			
Index	Documents Fetched from Database	Multi-threading Time Taken (sec)	Sequential Access Time Taken (sec)
1	20	0.023156	0.034339
2	40	0.027985	0.038538
3	80	0.03709	0.063974
4	160	0.029729	0.049847
5	320	0.053938	0.123961
6	640	0.093299	0.256861
7	1280	0.132452	0.165548
8	2560	0.184271	0.394815
9	5120	0.305893	0.490104
10	10240	0.528062	1.188434
11	20480	0.824746	1.330169
12	40960	1.621	3.57
13	81920	3.124	7.699
14	163840	6.819	11.593
15	327680	12.687	22.988

Table 1: Experimentaion results

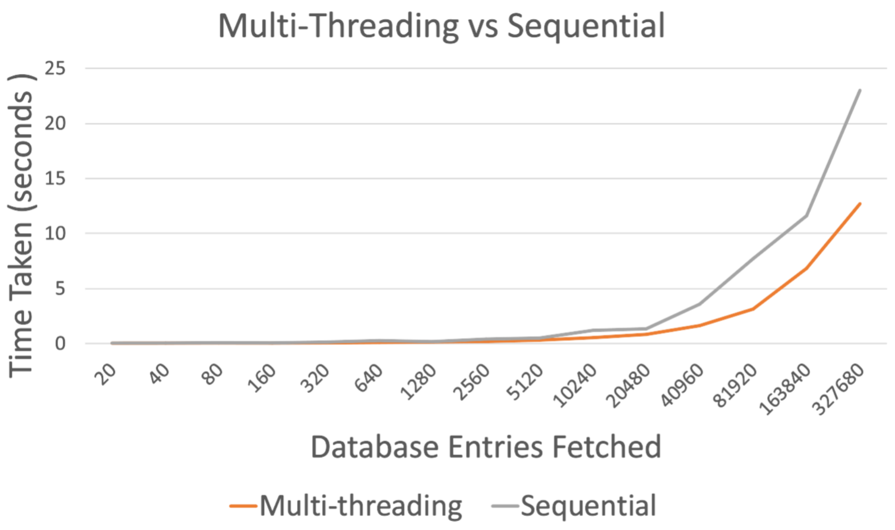


Figure 9: Visual comparison of Table 1

5 Conclusion

The use of multi-threading in an airline reservation system has proven to be an effective way of improving the data transfer process. The research shows that multi-threading significantly increases data transfer rates, with an 81.19% improvement when fetching 327680 documents, shown in figure 9 and table 1. This demonstrates the potential for multi-threading to be used in other systems that involve large amounts of data transfer. As such, further exploration into the use of multi-threading in other systems is necessary to fully realize its benefits.

References

[1] Ulili, S.: How To Use Multithreading in Node.js. https://www.digitalocean.com/community/tutorials/how-to-use-multithreading-in-node-js (2022)

[2] Rahman, K.M.: Handlebars and MongoDB with Express 4. https://medium.com/@iamcrypticcoder/handlebars-and-mongodb-with-express-4-2aeb808212d0 (2017)

[3] Sufiyan, T.: Getting Started With NodeJs MongoDB. https://www.simplilearn.com/tutorials/nodejs-tutorial/nodejs-mongodb (2023)