NoSQL Database Project Report

TELECOMMUNICATIONS PULSENET DB

BSc's in (Computer Forensics & Security)

Course SE602

A PROJECT REPORT BY

& Support Of

Jabez Dickson Exam No. 20102440

Clodagh Power Lecturer

Ollscoil Teicneolaíochta an Oirdheiscirt

South East Technological University

Table of Contents

1. Introduction

Report Briefing:

Report By: Jabez Jacob Dickson

Reviewed By: South East Technological University (SETU)

The telecommunications industry manages vast amounts of dynamic and diverse data, ranging from customer profiles to billing, call records, and mobile data usage.

This project aims to design and implement a NoSQL database solution for PulseNet, a telecommunications company, to address the need for scalable and flexible data management.

Unlike traditional relational databases, NoSQL databases like MongoDB are better suited for handling the unstructured and semi-structured data that PulseNet requires, ensuring high performance and adaptability to evolving business needs.

The proposed database will include key collections for customers, call records, and billing, enabling PulseNet to efficiently manage customer accounts, monitor call and data usage, and streamline billing processes.

This project seeks to demonstrate the effective application of MongoDB's document-oriented model to solve real-world challenges in telecommunications.

1.1 Objective of Assignment

To design and implement a NoSQL database for a telecommunications company to handle diverse and dynamic data related to customer accounts, billing, call records, and mobile data usage. Highlight the importance of scalability, flexibility, and performance in managing large datasets in the telecom industry.

Key Notes about the Setup

1. **Section 1:** Describes the project proposal and database design.
2. **Section 2:** Outlines the Document Object Model (DOM) and collection details.
3. **Section 3:** Covers implementation steps in relation to creating the collections
4. **Section 4:** Discus's CRUD operations and aggregation pipelines.
5. **Section 5:** Explaining the use of Aggregations.
6. **Section 6:** Concludes with lessons learned and reflections.

1.2 Overview of NoSQL and MongoDB

The NoSQL databases handles large-scale, unstructured data with flexibility and scalability, ideal for dynamic applications. MongoDB, a document-oriented NoSQL database, uses JSON-like documents, enabling schema flexibility and high performance.

NoSQL Databases:

- **Purpose:**
  - Designed to manage unstructured, semi-structured, and large-scale data more efficiently than traditional relational databases.
- **Core Features:**
  - Schema Flexibility: No fixed schema allows dynamic and evolving data models.
  - Scalability: Supports horizontal scaling through sharding, distributing data across multiple servers.
- **Performance:**
  - Optimized for high-speed read and write operations.
  - Handling large datasets with varying structures, such as logs, sensor data, or customer profiles.

Mongo Databases:

- **Purpose:**
  - Designed to manage unstructured, semi-structured, and large-scale data more efficiently than traditional relational databases.
- **Core Features:**
  - Stores data in BSON (Binary JSON) format, enabling nested structures and arrays.
  - Documents resemble JSON objects, making them intuitive for developers.
- **Scalability:**
  - Supports horizontal scaling through sharding, distributing data across clusters.
  - Ensures availability and fault tolerance with replica sets.
- **Aggregation Framework:**
  - Provides tools for data transformation and analysis, enabling real-time insights.
  - Common use cases include data grouping, filtering, and joining collections.

2. Project Proposal

2.1 Telecom DB Statement (PulseNet)

The database is designed to manage customer accounts for both Pay Monthly and Pay As You Go plans in the telecommunications context. It must store flexible information, including customer profiles, plan details, inclusive extras, billing data, and call records.

NoSQL Database Design The database design includes the following key data models:

Customer Accounts: Manages customer information, CTNs (Customer Telephone Numbers), billing, and call records.

Call Records: Logs incoming and outgoing calls, with details like duration, CTN destination, and chargeable status for international calls. This Tracks the current bill, payment history, and plan costs for Pay Monthly accounts.

Credit Class: Determines the type of plan assigned to Pay Monthly customers based on their credit score (Good, Very Good, or Excellent).

Inclusive Extras: Tracks subscription benefits tied to the customer's plan, such as Netflix, Apple TV, or Xbox Game Pass.

Account Status Management Account statuses include:

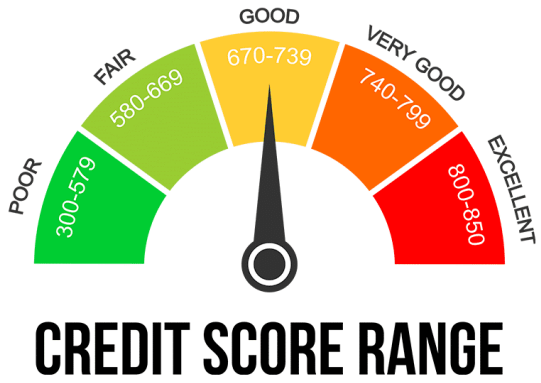
Active: Fully operational account.
Suspended: Temporarily deactivated due to reasons like SIM theft, fraud, or missed payment.
Cancelled: Permanently closed due to unpaid bills, customer death, or request for cancellation.

Account status is critical for managing customer activity and billing cycles.
Suspended accounts have restricted services and may be sent to collections if necessary.
Cancelled accounts are permanently disabled.

Credit Class and Plan Assignment Credit classes dynamically assign plans to customers credit scores:

Good (670–739): Plans below €50.
Very Good (740–799): Plans up to €100+.
Excellent (800–850): Plans €200+.

Integration: The system retrieves a customer's credit score and dynamically assigns the appropriate plan.



Data Flow and Management

Account Creation: Assigns a default "Good" credit class and allocates a plan during account setup.

Real-time Updates: Tracks remaining data, minutes, and texts for Pay As You Go customers.

Call Records: Logs detailed call data for both Pay Monthly and Pay As You Go customers.

Billing Cycles: Pay Monthly customers are billed according to their plan and usage. Pay As You Go customers have balances deducted based on usage.

2.2 Basic Requirements (PulseNet)

Data Management: Support customer accounts with various statuses, billing records, and real-time call data
Scalability: Seamlessly handle growing datasets with horizontal scaling through sharding.
Flexibility: Accommodate evolving data structures without requiring costly migrations.
Analytics: Provide real-time insights into customers billing trends, and data usage using aggregation
Availability: Ensure high uptime and fault tolerance through replication.

3. Database Design

3.1 DOM (Document Object Modelling)

The Document Object Model (DOM) for this project is designed to represent the telecommunications database in a structured and intuitive way, ensuring efficient data storage, retrieval, and relationships.

MongoDB's document-oriented model allows for embedding, referencing, and flexible schemas, which are ideal for handling the diverse datasets required by the telecommunications system.

The DOM includes the following core entities:

Customers: Stores customer details such as name, address, contact information, account status, and credit class.

Customers are stored as individual documents, with fields capturing their personal details, contact information, and account settings.

Embedded sub-documents (e.g., address and plan details) eliminate the need for complex joins, ensuring faster query execution.

Call Records: Maintains logs of incoming and outgoing calls with attributes such as duration, timestamp, and chargeable status.

Each document in this collection represents a call record, with references to the customer_id from the Customers collection.

Fields such as timestamp, duration, and chargeable allow for detailed call tracking and billing calculations.

Billing: Tracks billing details, including payment history, outstanding amounts, and due dates.

Documents in the Billing collection track payment details, referencing customer_id to tie each bill to a specific customer.

Fields like amount_due and paid help monitor financial transactions and outstanding balances.

These collections are interconnected using references, allowing relationships between customers and their respective call records or billing details while maintaining scalability and simplicity

3.2 Database Collections

Customer Collection

Field Name	Description	Type		
customer_id	Unique identifier	Integer		
name	Full name	String		
dob	Date of Birth	Date		
address	Sub-document containing street, city, and zip	Object (Sub-document)		
contact	Sub-document containing email and phone	Object (Sub-document)		
plan	Embedded document with type and cost	Object (Embedded)		
account_status	Account status (Active, Suspended, Cancelled)	String (Enum)		
credit_class	Credit classification (Good, Very Good, Excellent)	String (Enum)		

Call Record Collection

Field Name	Description	Type	
record_id	Unique identifier	Integer	
customer_id	Reference to Customers collection	Integer (Foreign Key)	
timestamp	Date and time of call	Date	
type	Call type (Incoming or Outgoing)	String (Enum)	
duration	Call duration in seconds	Integer	
chargeable	Whether the call is chargeable (True/False)	Boolean	

Billing Collection

Field Name	Description	Type
billing_id	Unique identifier	Integer
customer_id	Reference to Customers collection	Integer (Foreign Key)
amount_due	Outstanding amount	Decimal
due_date	Payment due date	Date
paid	Payment status (True/False)	Boolean

Credit Class Collection:

Field Name	Description	Data Type
class	Credit class name	String
min_score	Minimum credit score for the class	Integer
max_score	Maximum credit score for the class	Integer
max_plan_cost	Maximum plan cost allowed	Integer

Example Data

class	min_score	max_score	max_plan_cost
Good	670	739	50
Very Good	740	799	100
Excellent	800	850	200

Inclusive Extras Collection:

Field Name	Description	Data Type
extra_id	Unique identifier for the extra	Integer
name	Name of the subscription benefit	String
description	Description of the benefit	String

Example Data

extra_id	name	description
1	Apple iCloud	Apple’s Storage Service
2	Apple TV	Apple's streaming service
3	Apple Music	Apple’s Music subscription service

4. Implementation

4.1 Setting up MongoDB

This section outlines the installation and initial setup of MongoDB on a Windows operating system, including the use of MongoDB Shell, Visual Studio Code, and MongoDB Playgrounds.

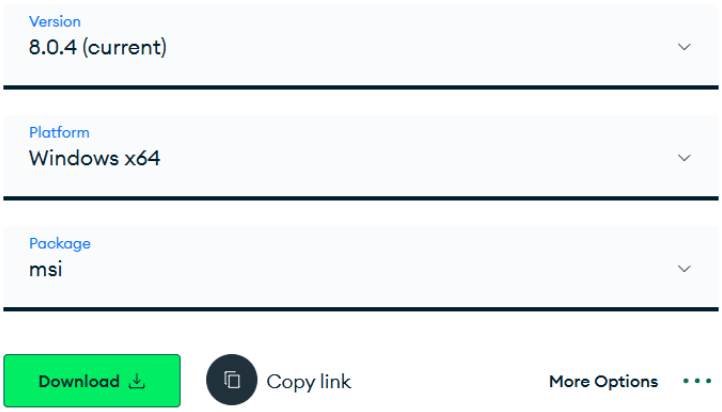
Step 1: Download and Install MongoDB

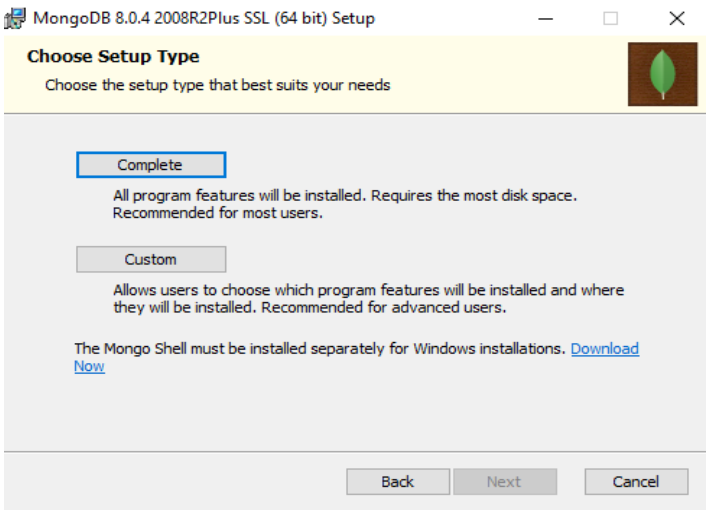
Navigate to the MongoDB Community Server download page: MongoDB Community Server Download.

Select the appropriate installer for your operating system and download the .msi file.

Follow the installation wizard:

1) Choose the following **link:** https://www.mongodb.com/try/download/community
2) Uncheck the "Install MongoDB as a Service" option. Exclude MongoDB Compass from the installation.
3) Add the path to the mongod.exe binary to your PATH environment variable for easier command-line
4) Choose the "Complete" setup type.

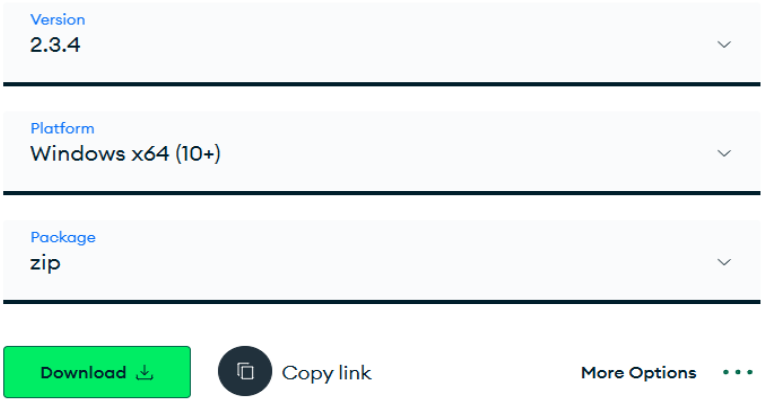


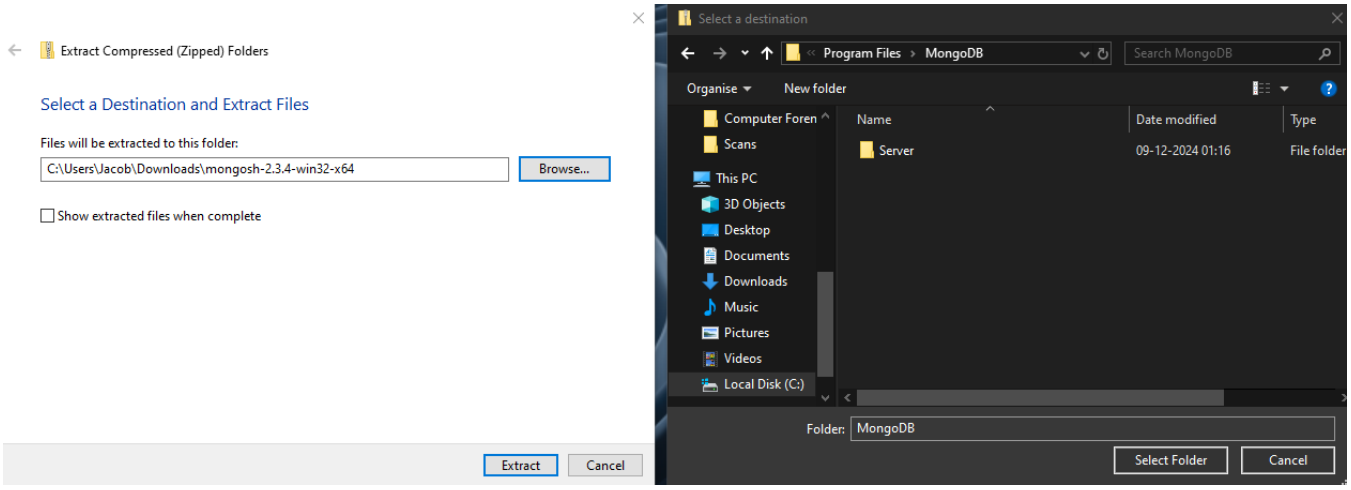


Step 2: Install MongoDB Shell

- Download and install the MongoDB Shell (mongosh).
- Add the path to the mongosh.exe binary to your PATH environment variable

- Install and extract the mongo shell zip file inside C Drive under the Program Files Directory where its labeled MongoDB for executable environment installation.





Step 3: Configure Data Storage

Open a command prompt with administrative privileges. Create the default MongoDB data directory by running the following command: mkdir c:\data\db

Step 4: Install Visual Studio Code:

https://code.visualstudio.com/download

Step 5: Start & Stop MongoDB Server and Shell

Open a terminal and start the MongoDB server by entering: **mongod**

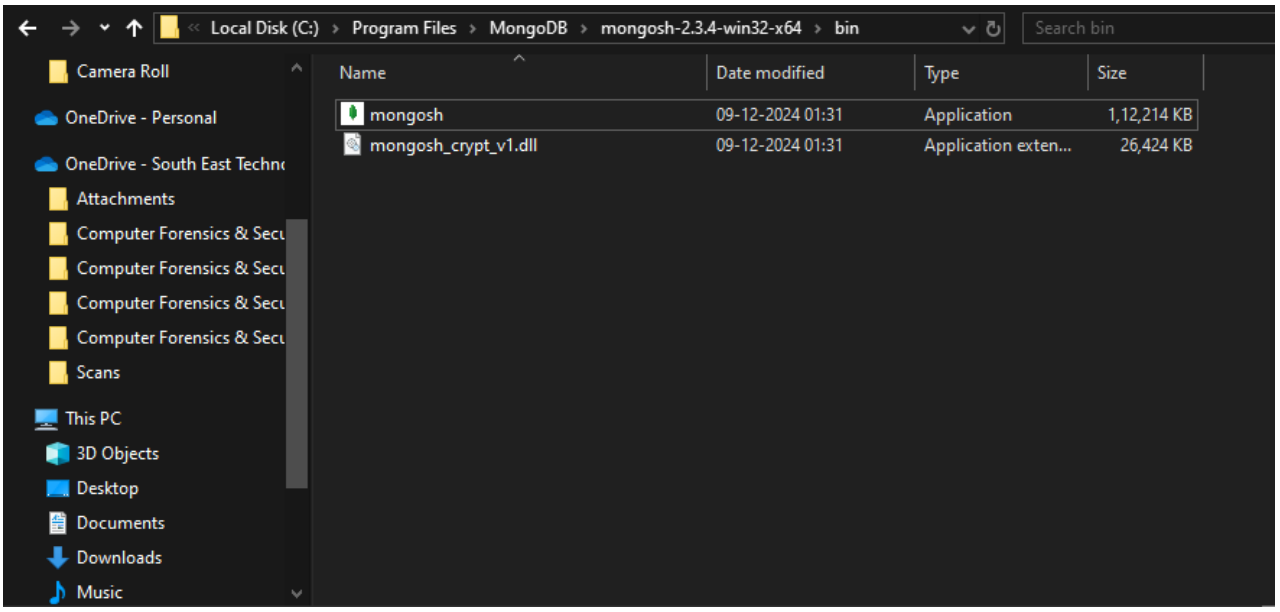
Leave this terminal running in the background. Open a terminal and start the MongoDB Shell with: **mongosh**

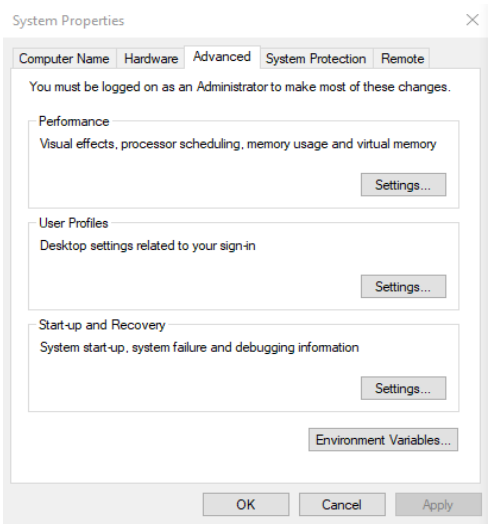
Press CTRL + C to stop the MongoDB Server and MongoDB Shell

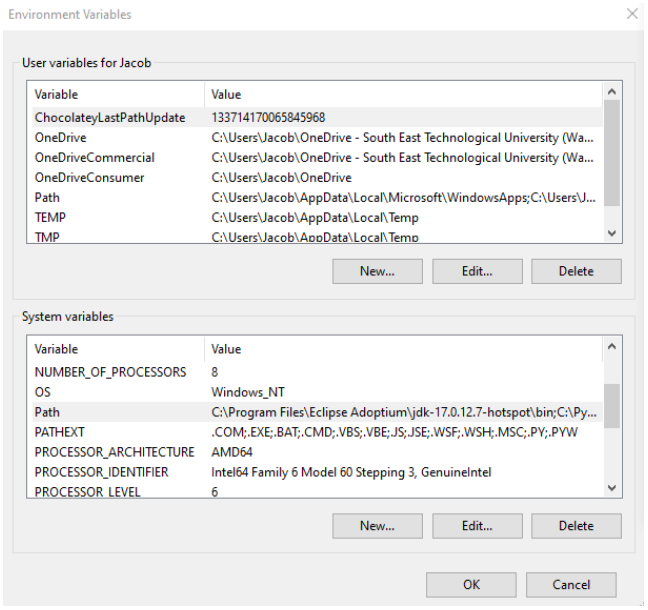
Step 6: Setup Executable Environment Variables

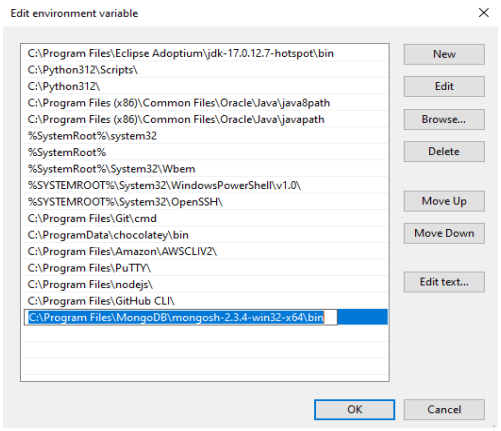
Now locate the mongosh executable file inside the extracted under the bin directory and copy the path which we will need to setup the environment variable.

In this case (e.g. it will be C:\Program Files\MongoDB\mongosh-2.3.4-win32-x64\bin) which can change based on the mongosh version downloaded









1) Choose Environment Variables
2) Choose Path under System Variables
3) Click New and Save the Copied file path

4.2 Creating Collections (Example)

In MongoDB, collections are created either dynamically by inserting data or explicitly using the `db.createCollection()` command.

1. Switch to a Database

Use the use command to create or switch to a database:

```
use labDB;
```

2. Creating a Collection

Collections are automatically created when you insert documents. For example:

```
db.students.insertOne({
    student_id: 101,
    name: "Alice Smith",
    age: 22,
    course: "Computer Science",
    grades: [85, 90, 78]
});
```

3. Explicitly Create Collections:

Use the `db.createCollection()` command to explicitly create a collection:

```
db.createCollection("courses");
db.createCollection("teachers");
```

4. Insert Multiple Documents

Use insertMany() to populate collections:

```
db.courses.insertMany([
    { course_id: "CS101", name: "Introduction to Programming",
credits: 3 },
    { course_id: "CS102", name: "Data Structures", credits: 4 }
]);
```

4.3 Creating Collections (Example)

Updating collections in MongoDB allows you to modify or replace existing documents using the updateOne, updateMany, and replaceOne commands.

Steps to Update Collections:

1. Update a Single Document:

Modify specific fields in a single document:

```
db.students.updateOne(
    { student_id: 101 },
    { $set: { age: 23 } }
);
```

2. Update Multiple Documents:

Apply changes to all matching documents:

```
db.courses.updateMany(
    { credits: 3 },
    { $set: { credits: 4 } }
);
```

3. Replace Document

Replace the entire document content

```
db.teachers.replaceOne(
    { teacher_id: 201 },
    { teacher_id: 201, name: "Dr John Doe", department: "Mathematics" }
);
```

4. Add New Fields

```
db.students.updateMany(
    {},
    { $set: { graduation_year: 2024 } }
);
```

5. Remove Fields

```
db.students.updateOne(
    { student_id: 101 },
    { $unset: { graduation_year: "" } }
);
```

NoSQL Structure Storage Solution

All the examples provided are **MongoDB NoSQL commands**. They use MongoDB's shell (`mongosh`) syntax for creating, updating, and managing collections and documents. Here's a breakdown of why this is specific to NoSQL:

1. **NoSQL Structure**:

   MongoDB stores data in documents, which are JSON-like objects (BSON internally).

   Collections do not enforce a rigid schema, allowing fields and data types to vary across documents.

2. **Key Features of NoSQL in the Examples**:

   Dynamic Schema: Collections are created dynamically when inserting documents.

   Embedded Documents and Arrays: Example documents contain fields with embedded objects (address) and arrays (grades).

   No Foreign Keys: Relationships are established through document references (if needed), not strict constraints like in relational databases.

   Flexible Commands: The update commands (updateOne, updateMany) and $set, $unset operators are specific to MongoDB's NoSQL architecture.

3. **MongoDB-Specific Commands**

   db.createCollection(): Creates a collection explicitly.

   insertOne() and insertMany(): Add single or multiple documents.

   updateOne() and updateMany(): Update documents selectively.

   $set, $unset: Operators to modify or remove fields.

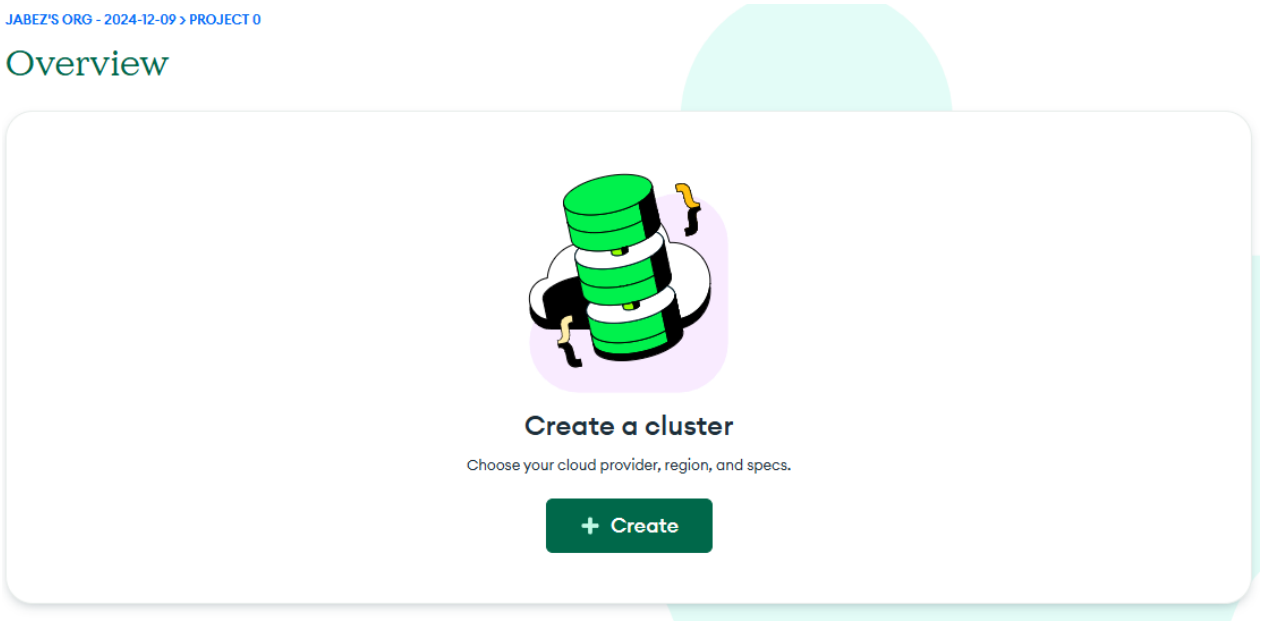
4.4 Setting Up a Free Cluster on Atlas

This section outlines the initial setup of MongoDB connection with an Atlas account which is a cloud-based database service provided by MongoDB that is used for hosting and managing MongoDB databases. It offers a fully managed, globally distributed, and scalable database solution.

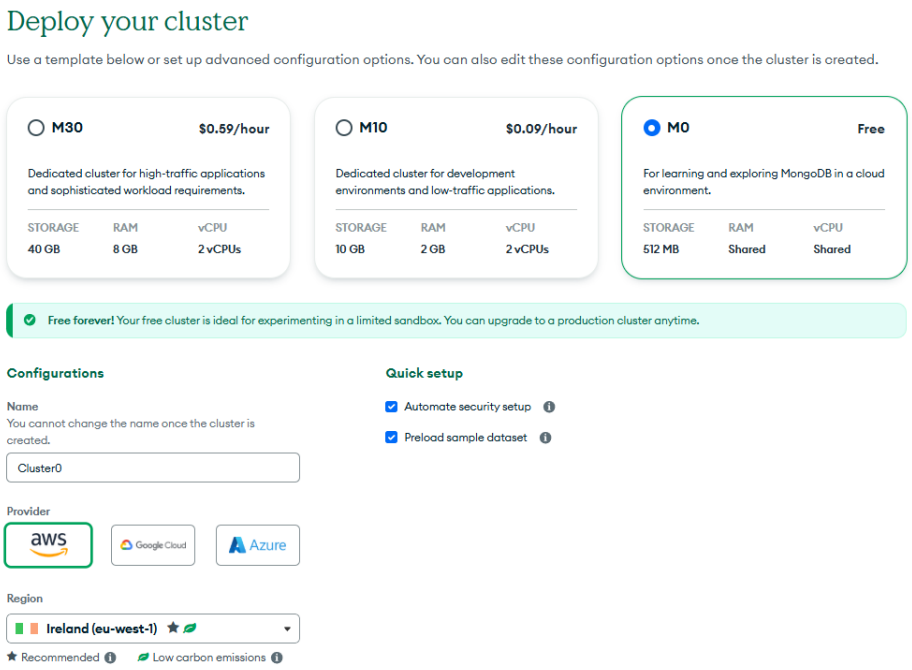
Step 1: Register for an Atlas Account

Navigate to the MongoDB Atlas Sign Up Page here - https://mdb.link/VkXvVOb99g0-register

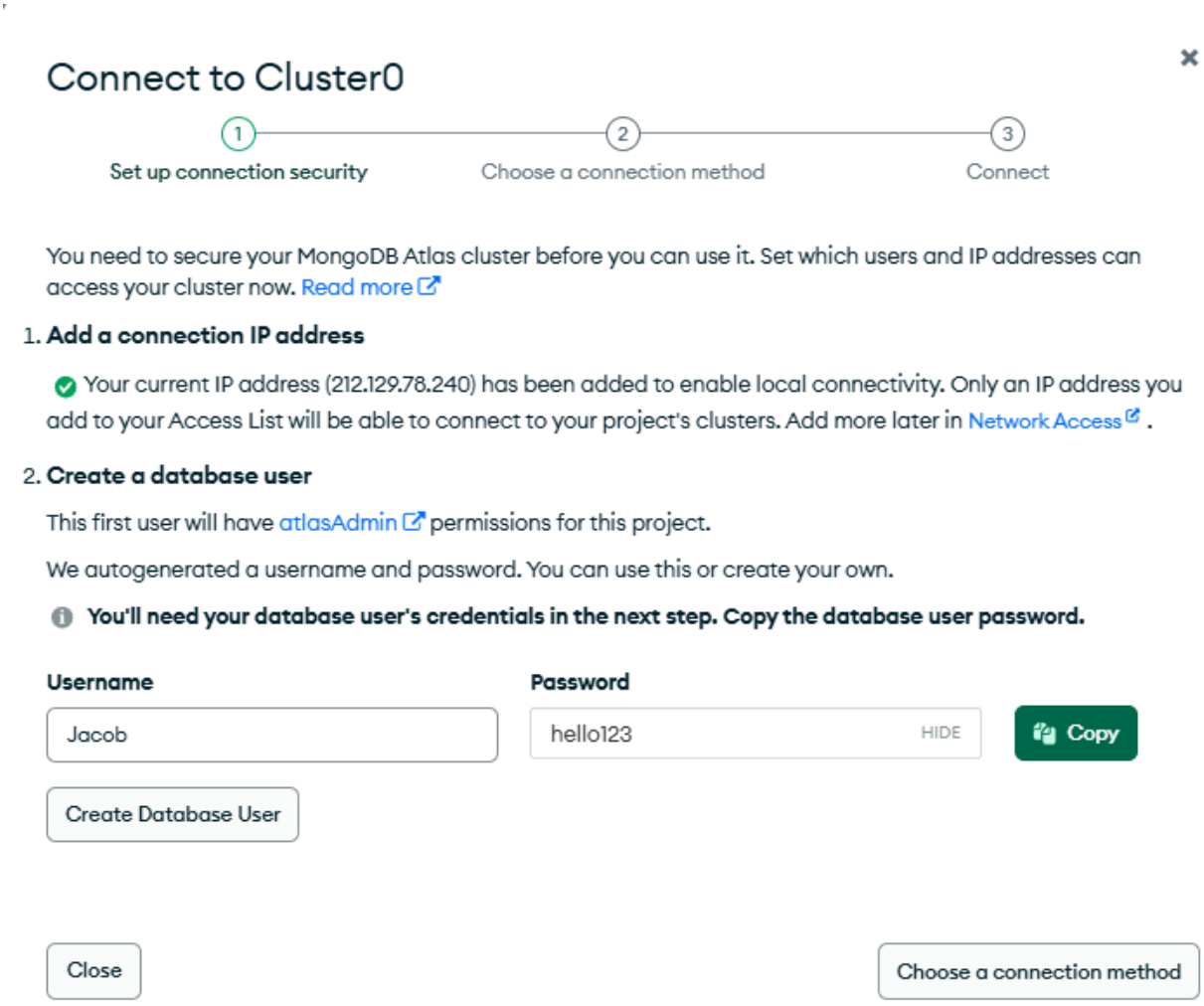
Once the Sign-Up Journey is complete you'll be invited to the introductory page like this:



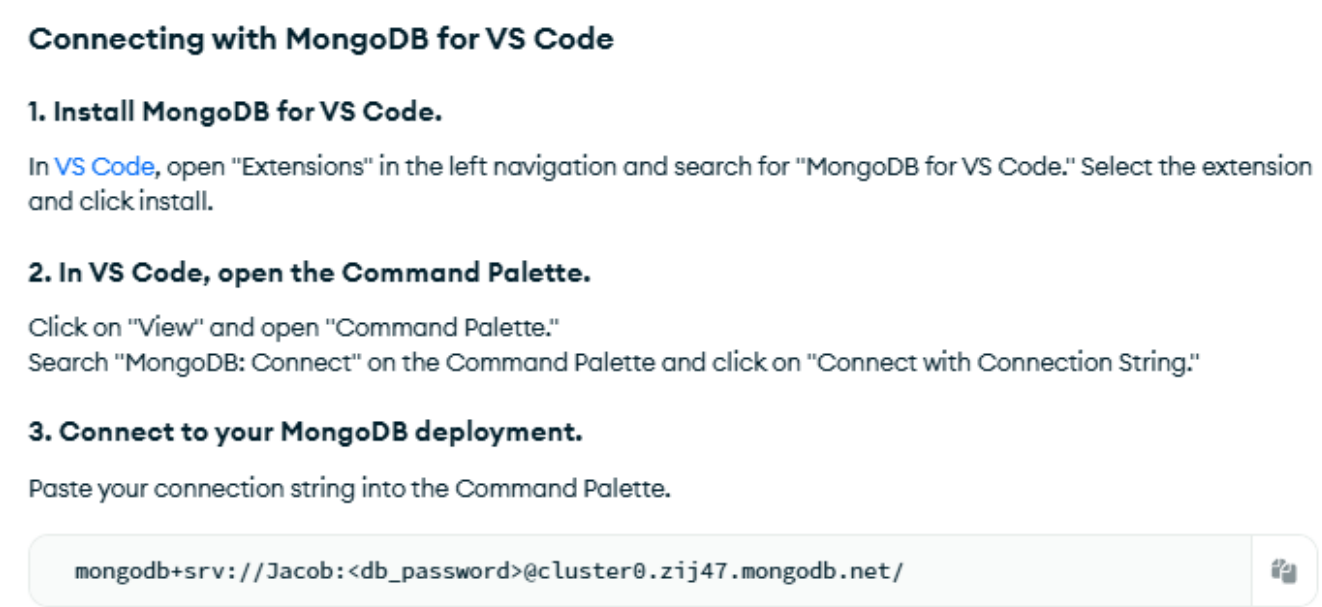
Choose the free tier and choose the default setting and a region based on personal performance for great response time and efficiency ad choose the AWS option and deploy the container



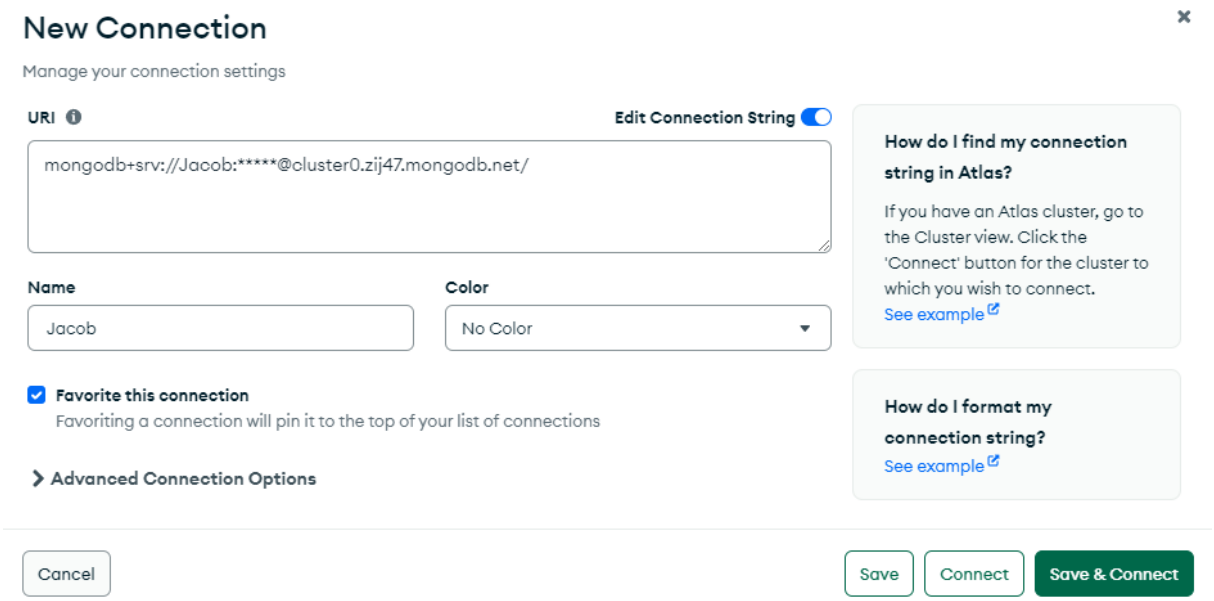
Step 2: Obtain the details to the Cloud Connection



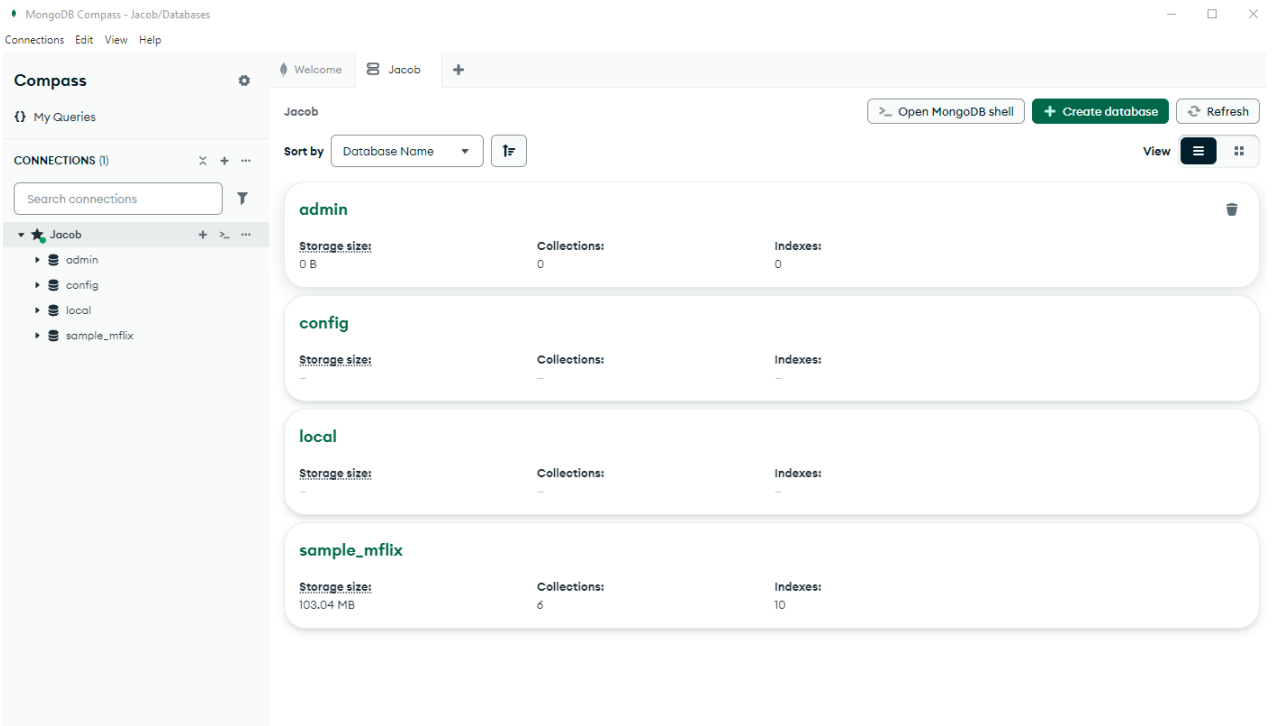
Step 3: Choose MongoDB for VS Code



Step 4: MongoDB Compass Pairing Setup (Alternative Method)



A successful connection will now be established giving the user access to the Atlas account with the running server on the AWS cloud system hosted on Mongo systems



5. CRUD Operations

5.1 Create, Read, Update & Delete Operations

Create Customer Collection

```
db.customers.insertOne({
    customer_id: 101,
    name: "John Doe",
    dob: ISODate("1990-01-01"),
    address: { street: "123 Elm St", city: "Waterford", zip: "X91P3K" },
    contact: { email: "john.doe@example.com", phone: "087-1234567" },
    plan: { type: "Pay Monthly", cost: 50 },
    account_status: "Active",
    credit_class: "Good"
    inclusive_extras: [1, 2]

});
```

Read Customer Collection

```
// Retrieve all customers
db.customers.find().pretty();

// Find a specific customer by ID
db.customers.find({ customer_id: 101 }).pretty();

// Retrieve customers with an "Active" account status
db.customers.find({ account_status: "Active" }).pretty();
```

Update Customer Collection

```
// Update account status for a specific customer
db.customers.updateOne(
    { customer_id: 101 },
    { $set: { account_status: "Suspended" } }
);

// Add a new field for loyalty points
db.customers.updateMany(
    {},
    { $set: { loyalty_points: 0 } }
);
```

Delete Customer Collection

```
// Delete a specific customer
db.customers.deleteOne({ customer_id: 101 });
```

Create Call Record Collection

```
db.call_records.insertMany([
    {
        record_id: 201,
        customer_id: 101,
        timestamp: ISODate("2024-01-01T10:00:00Z"),
        type: "Outgoing",
        duration: 300,
        chargeable: true
    },
    {
        record_id: 202,
        customer_id: 101,
        timestamp: ISODate("2024-01-02T12:30:00Z"),
        type: "Incoming",
        duration: 180,
        chargeable: false
    }
]);
```

Read Call Record Collection

```
// Retrieve all call records
db.call_records.find().pretty();

// Find all calls for a specific customer
db.call_records.find({ customer_id: 101 }).pretty();

// Retrieve chargeable calls
db.call_records.find({ chargeable: true }).pretty();
```

Update Call Record Collection

```
// Update the duration of a specific call
db.call_records.updateOne(
    { record_id: 201 },
    { $set: { duration: 360 } }
);
```

Delete Call Record Collection

```
// Delete all non-chargeable call records
db.call_records.deleteMany({ chargeable: false });
```

Create Inclusive Extra Collection

```
// Insert multiple inclusive extras into the collection
db.inclusive_extras.insertMany([
    {
        extra_id: 1,
        name: "Apple iCloud",
        description: "Apple’s Storage Service"
    },
    {
        extra_id: 2,
        name: "Apple TV",
        description: "Apple's Streaming Service"
    },
    {
        extra_id: 3,
        name: "Apple Music",
        description: "Apple’s Music Subscription Service"
    }
]);
```

Read Inclusive Extra Collection

```
// Retrieve all documents in the inclusive_extras collection
db.inclusive_extras.find().pretty();

// Find a specific extra by its name
db.inclusive_extras.find({ name: "Apple TV" }).pretty();
```

Update Inclusive Extra Collection

```
// Update the description of an inclusive extra by its ID
db.inclusive_extras.updateOne(
    { extra_id: 1 }, // Match the document where extra_id is 1
    { $set: { description: "Apple’s Cloud Storage Service" } } // Update the
description field
);
```

Delete Inclusive Extra Collection

```
// Delete a specific inclusive extra by its ID
db.inclusive_extras.deleteOne({ extra_id: 3 });
```

Create Billing Collection

```
db.billing.insertOne({
    billing_id: 301,
    customer_id: 101,
    amount_due: 50.00,
    due_date: ISODate("2024-02-01"),
    paid: false
});
```

Read Billing Collection

```
// Retrieve all billing records
db.billing.find().pretty();

// Find unpaid bills
db.billing.find({ paid: false }).pretty();

// Retrieve billing information for a specific customer
db.billing.find({ customer_id: 101 }).pretty();
```

Update Billing Collection

```
// Mark a bill as paid
db.billing.updateOne(
    { billing_id: 301 },
    { $set: { paid: true } }
);
```

Delete Billing Collection

```
// Delete all paid bills
db.billing.deleteMany({ paid: true });
```

Create Credit Class Collection

```
db.credit_classes.insertMany([
    { class: "Good", min_score: 670, max_score: 739, max_plan_cost: 50 },
    { class: "Very Good", min_score: 740, max_score: 799, max_plan_cost: 100 },
    { class: "Excellent", min_score: 800, max_score: 850, max_plan_cost: 200 }
]);
```

Read Credit Class Collection

```
// Retrieve all credit classes
db.credit_classes.find().pretty();

// Find the credit class for a specific score
db.credit_classes.find({ min_score: { $lte: 750 }, max_score: { $gte: 750 }
}).pretty();
```

Update Credit Class Collection

```
// Update the max_plan_cost for a specific class
db.credit_classes.updateOne(
    { class: "Good" },
    { $set: { max_plan_cost: 60 } }
);
```

Delete Credit Class Collection

```
// Delete a specific credit class
db.credit_classes.deleteOne({ class: "Excellent" });
```

Extra Functionality to Restore and Backup the Database

```
# Backup command
mongodump --db PulseNetDB --out /backup/directory

# Restore command
mongorestore --db PulseNetDB /backup/directory/PulseNetDB
```

6. Aggregation Pipeline (Integration)

Overview

Total Call Duration per Customer: Helps customers track usage and billing agents verify call duration for bills.

Monthly Revenue by Customer: Tracks revenue and outstanding balances for each customer during a billing cycle.

Top Data Users by Plan: Identifies high-value customers for personalized plan recommendations or marketing campaigns.

6.1 Total Call Duration per Customer

```
db.call_records.aggregate([
    // Step 1: Join with the "customers" collection to get customer details and plans
    {
        $lookup: {
            from: "customers",
            localField: "customer_id",
            foreignField: "customer_id",
            as: "customer_info"
        }
    },
    // Step 2: Flatten the "customer_info" array to access individual fields
    {
        $unwind: "$customer_info"
    },
    // Step 3: Group by customer_id to calculate total call duration for each customer
    {
        $group: {
            _id: "$customer_id",
            name: { $first: "$customer_info.name" },
            plan_type: { $first: "$customer_info.plan.type" },
            total_duration: { $sum: "$duration" }
        }
    },
    // Step 4: Sort customers by total call duration in descending order
    {
        $sort: { total_duration: -1 }
    },
    // Step 5: Format the output, including customer name, plan type, and total duration
    {
        $project: {
            _id: 0,
            customer_id: "$_id",
            name: 1,
            plan_type: 1,
            total_duration: 1
        }
    }
]);
```

6.2 Monthly Revenue of Customer

This pipeline calculates the total revenue generated by each customer for a specific billing month. This is valuable for billing agents to track customer payments and monitor overdue balances.

```
db.billing.aggregate([
    // Step 1: Match bills due within the specified month
    {
        $match: {
            due_date: {
                $gte: ISODate("2024-01-01"), // Start of the month
                $lt: ISODate("2024-02-01")  // End of the month
            }
        }
    },
    // Step 2: Group by customer_id to calculate total revenue
    {
        $group: {
            _id: "$customer_id",
            total_revenue: { $sum: "$amount_due" },
            total_paid: { $sum: { $cond: [{ $eq: ["$paid", true] },
"$amount_due", 0] } },
            total_outstanding: { $sum: { $cond: [{ $eq: ["$paid", false] },
"$amount_due", 0] } }
        }
    },
    // Step 3: Join with the "customers" collection to get customer details
    {
        $lookup: {
            from: "customers",
            localField: "_id", // Match on customer_id
            foreignField: "customer_id",
            as: "customer_info"
        }
    },
    // Step 4: Format the output
    {
        $project: {
            _id: 0,
            customer_id: "$_id",
            name: { $arrayElemAt: ["$customer_info.name", 0] },
            total_revenue: 1,
            total_paid: 1,
            total_outstanding: 1
        }
    },
    // Step 5: Sort customers by total revenue in descending order
    {
        $sort: { total_revenue: -1 }
    }
]);
```

6.3 Top Data Users by Plan

This pipeline identifies customers who used the most data in a given period. It helps telecom agents spot high-usage customers for targeted marketing or plan upsell opportunities.

```
db.call_records.aggregate([
    // Step 1: Join with the "customers" collection to get customer details and
plans
    {
        $lookup: {
            from: "customers",
            localField: "customer_id", // Field in `call_records`
            foreignField: "customer_id", // Field in `customers`
            as: "customer_info"
        }
    },
    // Step 2: Flatten the "customer_info" array to access individual fields
    {
        $unwind: "$customer_info"
    },
    // Step 3: Group by customer_id to calculate total call duration for each
customer
    {
        $group: {
            _id: "$customer_id",
            name: { $first: "$customer_info.name" },
            plan_type: { $first: "$customer_info.plan.type" },
            total_duration: { $sum: "$duration" }
        }
    },
    // Step 4: Sort customers by total call duration in descending order
    {
        $sort: { total_duration: -1 }
    },
    // Step 5: Limit the result to the top 5 high data users
    {
        $limit: 5
    },
    // Step 6: Format the output
    {
        $project: {
            _id: 0,
            customer_id: "$_id",
            name: 1,
            plan_type: 1,
            total_duration: 1
        }
    }
]);
```

Monogosh Terminal Screenshots

Adding Inclusive Extras to Collection

```
> use PulseNetDB
< already on db PulseNetDB
> db.inclusive_extras.insertMany([
    { extra_id: 1, name: "Apple TV", description: "Apple's streaming service" },
    { extra_id: 2, name: "Netflix", description: "Netflix streaming service" },
    { extra_id: 3, name: "Spotify", description: "Spotify music streaming" },
    { extra_id: 4, name: "Xbox Game Pass", description: "Gaming subscription service" },
    { extra_id: 5, name: "Disney+", description: "Disney's streaming service" }
  ]);
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('675735bd536bf3218afe9fd7'),
      '1': ObjectId('675735bd536bf3218afe9fd8'),
      '2': ObjectId('675735bd536bf3218afe9fd9'),
      '3': ObjectId('675735bd536bf3218afe9fda'),
      '4': ObjectId('675735bd536bf3218afe9fdb')
    }
  }
```

Adding Customers to Collection

```
  {
    customer_id: 4,
    name: "Michael Brown",
    dob: ISODate("1978-12-05"),
    address: { street: "321 Maple Blvd", city: "Galway", zip: "H91R3K" },
    contact: { email: "michael.brown@example.com", phone: "087-2233445" },
    plan: { type: "Pay Monthly", cost: 70 },
    account_status: "Suspended",
    credit_class: "Good",
    inclusive_extras: [4] // Xbox Game Pass
  },
  {
    customer_id: 5,
    name: "Sarah Lee",
    dob: ISODate("1988-06-22"),
    address: { street: "654 Cedar St", city: "Limerick", zip: "V94F3K" },
    contact: { email: "sarah.lee@example.com", phone: "087-3344556" },
    plan: { type: "Pay As You Go", cost: 30 },
    account_status: "Cancelled",
    credit_class: "Very Good",
    inclusive_extras: [5] // Disney+
  }
]);
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('675735c7536bf3218afe9fdc'),
    '1': ObjectId('675735c7536bf3218afe9fdd'),
    '2': ObjectId('675735c7536bf3218afe9fde'),
    '3': ObjectId('675735c7536bf3218afe9fdf'),
    '4': ObjectId('675735c7536bf3218afe9fe0')
  }
}
```

Adding Billing to Collection

```
> db.billing.insertMany([
    { billing_id: 101, customer_id: 1, amount_due: 50, due_date: ISODate("2024-01-01"), paid: false },
    { billing_id: 102, customer_id: 2, amount_due: 20, due_date: ISODate("2024-01-10"), paid: true },
    { billing_id: 103, customer_id: 3, amount_due: 100, due_date: ISODate("2024-01-15"), paid: false },
    { billing_id: 104, customer_id: 4, amount_due: 70, due_date: ISODate("2024-01-20"), paid: true },
    { billing_id: 105, customer_id: 5, amount_due: 30, due_date: ISODate("2024-01-25"), paid: true }
  ]);
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('675735d9536bf3218afe9fe1'),
      '1': ObjectId('675735d9536bf3218afe9fe2'),
      '2': ObjectId('675735d9536bf3218afe9fe3'),
      '3': ObjectId('675735d9536bf3218afe9fe4'),
      '4': ObjectId('675735d9536bf3218afe9fe5')
    }
  }
```

Adding Call Records to Collection

```
> db.call_records.insertMany([
    { record_id: 201, customer_id: 1, timestamp: ISODate("2024-01-02T10:00:00Z"), type: "Outgoing", duration: 300, chargeable: true },
    { record_id: 202, customer_id: 2, timestamp: ISODate("2024-01-03T11:30:00Z"), type: "Incoming", duration: 180, chargeable: false },
    { record_id: 203, customer_id: 3, timestamp: ISODate("2024-01-04T12:00:00Z"), type: "Outgoing", duration: 240, chargeable: true },
    { record_id: 204, customer_id: 4, timestamp: ISODate("2024-01-05T14:00:00Z"), type: "Outgoing", duration: 60, chargeable: true },
    { record_id: 205, customer_id: 5, timestamp: ISODate("2024-01-06T16:00:00Z"), type: "Incoming", duration: 150, chargeable: false }
  ]);
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('675735e3536bf3218afe9fe6'),
      '1': ObjectId('675735e3536bf3218afe9fe7'),
      '2': ObjectId('675735e3536bf3218afe9fe8'),
      '3': ObjectId('675735e3536bf3218afe9fe9'),
      '4': ObjectId('675735e3536bf3218afe9fea')
    }
  }
```

Adding Credit Class Collection

```
> db.credit_classes.insertMany([
    { class: "Good", min_score: 670, max_score: 739, max_plan_cost: 50 },
    { class: "Very Good", min_score: 740, max_score: 799, max_plan_cost: 100 },
    { class: "Excellent", min_score: 800, max_score: 850, max_plan_cost: 200 }
  ]);
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('675735ec536bf3218afe9feb'),
      '1': ObjectId('675735ec536bf3218afe9fec'),
      '2': ObjectId('675735ec536bf3218afe9fed')
    }
  }
```

References

MongoDB Documentation: Comprehensive guides and resources on MongoDB's features and functionalities.
Link: https://www.mongodb.com/docs/

MongoDB Atlas Documentation: Detailed information on deploying and managing databases using MongoDB's cloud platform.
Link: https://www.mongodb.com/docs/atlas/

Getting Started with MongoDB Atlas: A step-by-step tutorial for creating an Atlas cluster, connecting to it, and loading sample data.
Link: https://www.mongodb.com/docs/atlas/getting-started/

MongoDB CRUD Operations: An overview of create, read, update, and delete operations in MongoDB.
Link: https://www.mongodb.com/docs/manual/crud/

MongoDB Manual Contents: The official manual covering core MongoDB concepts, including data modelling, querying, and aggregation.
Link: https://www.mongodb.com/docs/manual/contents/

MongoDB Atlas Search: Information on building full-text search capabilities on top of your data using Atlas Search.
Link: https://www.mongodb.com/docs/atlas/full-text-search/

MongoDB Atlas Data Lake: Details on integrating and analysing data across different sources using Atlas Data Lake.
Link: https://www.mongodb.com/docs/atlas/data-lake/

MongoDB Aggregation Framework: Documentation on performing aggregation operations in MongoDB.
Link: https://www.mongodb.com/docs/manual/aggregation/

MongoDB Indexes: Guidance on creating and managing indexes to optimize query performance.
Link: https://www.mongodb.com/docs/manual/indexes/

MongoDB Transactions: Explanation of multi-document ACID transactions in MongoDB.
Link: https://www.mongodb.com/docs/manual/transactions/

Visual Studio Code MongoDB Integration: Learn how to integrate MongoDB with Visual Studio Code using the MongoDB extension.
Link: https://marketplace.visualstudio.com/items?itemName=mongodb.mongodb-vscode