**Evaluation of MongoDB for Restaurant and Cafe POS and Inventory Management System**

KAHNDISM232F-018

KAHNDISM232F-019

KAHNDISM232F-020

KAHNDISM232F-021

**Contents**

1. Introduction
2. Mongodb and justification
3. Need of mongodb
4. Database schema
5. Commands of CRUD operations
6. Reporting and other functionalities
7. Strengths and limitations

1.Introduction

This report evaluates MongoDB as a solution for developing a POS and Inventory Management System for a restaurant and cafe. The analysis covers the database schema design, example queries for reporting, indexing, scaling techniques, and the strengths and limitations of MongoDB.

2.Mongodb and justification

Based on the given requirements, a document store NoSQL database model, such as MongoDB, is the most suitable choice. Here’s the justification:

1. Flexibility and Schema-less Nature

Dynamic Schema: MongoDB's schema-less nature allows for a flexible and dynamic data model. This is beneficial for storing varied item information, different variations, ingredients, and extra options without the need to predefine a rigid schema.

Evolving Requirements: As the restaurant business evolves, new fields can be easily added to documents without altering the existing structure. This is particularly useful for handling new variations, extra options, or additional sales details.

2. Document-oriented Storage:

Complex Data Structures: MongoDB stores data in JSON-like documents, which can easily accommodate nested structures. This is ideal for representing complex entities like food items with variations, ingredients, and extra options within a single document.

Embedded Documents: The ability to embed documents within documents allows for the efficient storage and retrieval of related data, such as customer orders along with item details, extra options, and ingredient information.

3. Aggregation and Rich Query Capabilities:

Aggregation Framework: MongoDB’s powerful aggregation framework can efficiently handle complex queries and transformations, such as calculating the total cost of an order, including extra options and discounts.

Indexing and Performance: MongoDB supports indexing, which improves the performance of read and write operations. This is crucial for managing a large volume of sales transactions and inventory updates.

4. Scalability and Performance:

Horizontal Scalability: MongoDB is designed for horizontal scalability, allowing the system to handle increased loads by distributing data across multiple servers. This is important for a growing business that anticipates an increase in sales transactions and inventory records.

High Performance: MongoDB’s ability to handle high read and write throughput ensures that the POS and inventory management system remains responsive even during peak business hours.

5. User Management and Role-based Access Control:

Built-in Security Features: MongoDB provides robust security features, including role-based access control (RBAC) and the ability to define user roles and permissions. This aligns with the requirement for multiple access levels and administrative control over user permissions.

6. Integration with Other Tools:

Ecosystem: MongoDB integrates well with various tools and technologies used in modern application development, including data analytics platforms, cloud services, and application frameworks. This ensures seamless integration and extensibility for future enhancements.

3.The need of mongodb

The need for a NoSQL database solution for the given scenario is driven by several key factors:

1. Flexibility and Dynamic Schema

Diverse Data Types: The application needs to manage various data types, such as item information, variations, sales records, and purchase details. NoSQL databases like MongoDB allow for a flexible schema, making it easier to store diverse and evolving data structures without predefined schemas.

Schema Evolution: In a dynamic environment like a restaurant, new fields and attributes can emerge frequently (e.g., new item variations, additional customer preferences). NoSQL databases enable seamless schema evolution without the need for costly migrations or schema redesigns.

2. Complex Data Modeling

Nested and Hierarchical Data: Food items can have nested variations (e.g., size, color, flavors) and associated extra options (e.g., extra cheese, extra ice). NoSQL document stores allow for nested documents, making it easier to represent hierarchical data structures within a single document.

Embedded Relationships: The relationships between items, ingredients, and extra options can be embedded within documents. This reduces the need for complex joins and enhances data retrieval performance.

3. Scalability and Performance

Horizontal Scalability: NoSQL databases are designed for horizontal scalability, allowing data to be distributed across multiple servers. This is crucial for handling increasing data volumes as the restaurant grows and processes more sales and inventory records.

High Throughput: NoSQL databases are optimized for high read and write throughput, ensuring that the system can handle peak loads efficiently, such as during busy dining hours or promotional events.

4. Complex Queries and Aggregations

Aggregation Framework: The application requires complex queries and aggregations, such as calculating total sales, tracking inventory usage, and generating detailed sales reports. NoSQL databases like MongoDB provide powerful aggregation frameworks to perform these operations efficiently.

Indexing Capabilities: NoSQL databases offer advanced indexing capabilities to improve query performance, enabling quick retrieval of sales records, inventory levels, and user access logs.

5. Adaptability to Unstructured Data

Varied Data Sources: The application needs to handle data from multiple sources, including sales transactions, online orders, and supplier information. NoSQL databases are well-suited for integrating and managing unstructured or semi-structured data from diverse sources.

Rapid Development: The flexibility of NoSQL databases allows for rapid application development and iteration, enabling the restaurant to quickly adapt to changing business requirements and customer needs.

6. Cost-Effectiveness

Reduced Administrative Overhead: NoSQL databases often require less administrative overhead compared to traditional relational databases, especially in terms of schema management and scaling.

Open-Source Solutions: Many NoSQL databases, such as MongoDB, offer open-source editions, providing cost-effective solutions for small to medium-sized businesses.

7. User Management and Security

Role-Based Access Control: NoSQL databases support role-based access control (RBAC), ensuring that users have appropriate access levels based on their roles. This is essential for managing access to sensitive sales and inventory data.

Audit and Monitoring: NoSQL databases provide features for monitoring and auditing database activities, enhancing security and compliance with internal policies.

4.Datbase schema

Database Schema Design for Restaurant and Cafe POS and Inventory Management System

* Collections and Fields

1. Items Collection

Fields:

* `item\_id`: Unique identifier for the item.
* `name`: Name of the item.
* `unit\_price`: Price per unit of the item.
* `stock`: Current stock level of the item.
* `discount\_rate`: Discount percentage applicable to the item.
* `category`: Category to which the item belongs.
* `variations`: Array of embedded documents for item variations (e.g., size, color, flavor).
* `type`: Type of variation (e.g., size, color).
* `value`: Value of the variation (e.g., Large, Red).
* `extra\_options`: Array of embedded documents for extra options (e.g., extra cheese, extra cream).
* `name`: Name of the extra option.
* `weight\_per\_unit`: Weight of one unit of the extra option.

Rationale:

Embedded Documents for Variations and Extra Options: Embedding variations and extra options within the item document allows for efficient retrieval of item details in a single query. This denormalization optimizes read operations, as it avoids the need for frequent joins between items, variations, and extra options.

2. Users Collection

Fields:

* `user\_id`: Unique identifier for the user.
* `username`: Username of the user.
* `password`: Hashed password for security.
* `role`: Role of the user (e.g., admin, staff).
* `full\_name`: Full name of the user.
* `permissions`: Array of permissions assigned to the user.

Rationale:

Role-Based Access Control: Keeping user details in a separate collection ensures secure management of user credentials and roles. The use of embedded permissions within the user document supports flexible and granular access control.

3. Sales Collection

Fields:

* `sale\_id`: Unique identifier for the sale.
* `invoice\_number`: Invoice number of the sale.
* `date`: Date of the sale.
* `total`: Total amount of the sale.
* `paid\_amount`: Amount paid by the customer.
* `balance\_amount`: Balance amount to be paid by the customer.
* `remark`: Additional remarks for the sale.
* `items`: Array of embedded documents for items purchased in the sale.
* `item\_id`: Reference to the item sold.
* `quantity`: Quantity of the item sold.
* `unit\_price`: Unit price of the item at the time of sale.
* - `extras`: Array of embedded documents for additional options chosen for the item.
* `name`: Name of the extra option (e.g., extra cheese).
* `weight`: Weight or quantity of the extra option added.
* `customer\_type`: Type of customer (e.g., dine-in, takeaway, online).
* `dine\_in\_details`: Embedded document for dine-in customer details (if applicable).
* `table\_number`: Table number for dine-in.
* `staff\_member`: Staff member serving the customer.
* `takeaway\_details`: Embedded document for takeaway customer details (if applicable).
* `token\_number`: Token number for the order.
* `kot\_number`: Kitchen Order Ticket number.
* `online\_order\_details`: Embedded document for online order details (if applicable).
* `order\_reference\_number`: Reference number for the online order.
* `delivery\_address`: Delivery address for the online order.

Rationale:

Embedded Documents for Items, Extras, and Customer Details: Embedding items, extras, and customer details within the sale document optimizes retrieval of sales details in a single query. This denormalization supports efficient reporting and analytics on sales data without needing complex joins.

4. Purchases Collection

Fields:

* `purchase\_id`: Unique identifier for the purchase.
* `grn\_id`: Goods Receipt Note (GRN) ID associated with the purchase.
* `supplier\_details`: Details of the supplier.
* `date`: Date of the purchase.
* `total`: Total cost of the purchase.
* `paid`: Amount paid for the purchase.
* `balance`: Balance amount pending for the purchase.
* `items`: Array of embedded documents for items purchased.
* `item\_id`: Reference to the item purchased.
* `quantity`: Quantity of the item purchased.
* `unit\_price`: Unit price of the item at the time of purchase.
* `discount\_received`: Discount received for the item.
* `free\_issues`: Free issues provided by the supplier.

Rationale:

Embedded Documents for Items: Embedding items within the purchase document simplifies the retrieval of purchase details and associated items in a single query. This denormalization improves performance by reducing the need for frequent joins between purchases and items.

**Denormalization and Embedding**

Denormalization: By embedding related data (variations, items in sales and purchases, extra options, customer details), we optimize read performance by minimizing the need for joins. This approach enhances query efficiency and supports faster data retrieval, which is crucial for real-time reporting and analytics in a POS system.

Separate Collections: Keeping users in a separate collection ensures secure management of user credentials and roles, allowing for flexible and granular access control.

Item Collection

{

"$jsonSchema": {

"bsonType": "object",

"required": ["name", "unit\_price", "stock", "category"],

"properties": {

"item\_id": { "bsonType": "objectId" },

"name": { "bsonType": "string" },

"unit\_price": { "bsonType": "number" },

"stock": { "bsonType": "number" },

"discount\_rate": { "bsonType": "number" },

"category": { "bsonType": "string" },

"variations": {

"bsonType": "array",

"items": {

"bsonType": "object",

"properties": {

"type": { "bsonType": "string" },

"value": { "bsonType": "string" }

}

}

},

"ingredients": {

"bsonType": "array",

"items": {

"bsonType": "object",

"properties": {

"ingredient\_name": { "bsonType": "string" },

"quantity": { "bsonType": "number" },

"cost": { "bsonType": "number" }

}

}

},

"extra\_options": {

"bsonType": "array",

"items": {

"bsonType": "object",

"properties": {

"option\_name": { "bsonType": "string" },

"weight": { "bsonType": "number" }

}

}

}

}

}

}

Sales Collection

{

"$jsonSchema": {

"bsonType": "object",

"required": ["invoice\_number", "date", "total", "items", "customer\_type"],

"properties": {

"sales\_id": { "bsonType": "objectId" },

"invoice\_number": { "bsonType": "string" },

"date": { "bsonType": "date" },

"total": { "bsonType": "number" },

"paid\_amount": { "bsonType": "number" },

"balance\_amount": { "bsonType": "number" },

"remark": { "bsonType": "string" },

"items": {

"bsonType": "array",

"items": {

"bsonType": "object",

"properties": {

"item\_id": { "bsonType": "objectId" },

"quantity": { "bsonType": "number" },

"unit\_price": { "bsonType": "number" },

"extra\_options": {

"bsonType": "array",

"items": {

"bsonType": "object",

"properties": {

"option\_name": { "bsonType": "string" },

"quantity": { "bsonType": "number" }

}

}

}

}

}

},

"customer\_type": { "bsonType": "string" }, // "takeaway", "dine-in", "online"

"token\_number": { "bsonType": "string" },

"KOT\_number": { "bsonType": "string" },

"table\_number": { "bsonType": "string" },

"staff\_member": {

"bsonType": "object",

"properties": {

"staff\_id": { "bsonType": "objectId" },

"name": { "bsonType": "string" }

}

},

"order\_reference\_number": { "bsonType": "string" },

"delivery\_address": { "bsonType": "string" }

}

}

}

Purchase Collection

{

"$jsonSchema": {

"bsonType": "object",

"required": ["GRN\_id", "supplier\_details", "date", "total", "items"],

"properties": {

"purchase\_id": { "bsonType": "objectId" },

"GRN\_id": { "bsonType": "string" },

"supplier\_details": {

"bsonType": "object",

"required": ["supplier\_id", "name", "contact\_info"],

"properties": {

"supplier\_id": { "bsonType": "objectId" },

"name": { "bsonType": "string" },

"contact\_info":

}}

User Collection

{

"$jsonSchema": {

"bsonType": "object",

"required": ["GRN\_id", "supplier\_details", "date", "total", "items"],

"properties": {

"purchase\_id": { "bsonType": "objectId" },

"GRN\_id": { "bsonType": "string" },

"supplier\_details": {

"bsonType": "object",

"required": ["supplier\_id", "name", "contact\_info"],

"properties": {

"supplier\_id": { "bsonType": "objectId" },

"name": { "bsonType": "string" },

"contact\_info": { "bsonType": "string" }

}

},

"date": { "bsonType": "date" },

"total": { "bsonType": "number" },

"paid": { "bsonType": "number" },

"balance": { "bsonType": "number" },

"free\_issues": { "bsonType": "string" },

"items": {

"bsonType": "array",

"items": {

"bsonType": "object",

"required": ["item\_id", "quantity"],

"properties": {

"item\_id": { "bsonType": "objectId" },

"quantity": { "bsonType": "number" },

"discount\_received": { "bsonType": "number" }

}

}

}

}

}

}

5.MongoDB Schema and CRUD Operations

1. Create Collections

// Connect to MongoDB

use restaurant\_cafe\_db;

// Create collections

db.createCollection("items");

db.createCollection("users");

db.createCollection("sales");

db.createCollection("purchases");

2. CRUD Operations

1. Items Collection

Create (Insert), Read, Update, Delete Operations:

Insert an item

let item=({

item\_id: "1",

name: " Milk Coffee",

unit\_price: 500,

stock: 100,

discount\_rate: 0,

category: "Beverage",

variations: [

{ type: "Size", value: "Regular" },

]

});

// Find all items

db.getCollection('items').find();

// Update stock for an item

db.getCollection('items').updateOne(

{ item\_id: "1" },

{ $set: { stock: 90 } }

);

// Delete an item

db.getCollection('items').deleteOne({ item\_id: "1" });

2. Users Collection

let Users=({

user\_id: "user001",

username: "admin",

password: "hashed\_password",

role: "admin",

full\_name: "Oshada"

});

// Find all users

db.getCollection('users').find();

// Find a user by username

db.getCollection('users').find({ username: "admin" });

// Update password for a user

db.getCollection('users').updateOne(

{ username: "admin" },

{ $set: { password: "new\_hashed\_password" } }

);

// Delete a user

db.getCollection('user').deleteOne({ username: "admin" });

3. Sales Collection

Create (Insert), Read, Update, Delete Operations:

// Insert a sale

let sale=({

sale\_id: "2023001",

invoice\_number: "INV2023001",

date: ISODate("2024-04-01"),

total: 10,

paid\_amount: 10,0000,

balance\_amount: 0,

remark: "Paid in full",

items: [

{ item\_id: "1", quantity: 2, unit\_price: 100, extras: [] },

{ item\_id: "2", quantity: 1, unit\_price: 200, extras: [

{ name: "Extra Cheese", weight: "100g" }

] }

]

});

// Find all sales

db.getCollection('sales').find();

// Update paid amount for a sale

db.getCollection('sales').updateOne(

{ sale\_id: "2023001" },

{ $set: { paid\_amount: 5, balance\_amount: 5 } }

);

// Delete a sale

db.getCollection('sales').deleteOne({ sale\_id: "2023001" });

```

4. Purchases Collection

Create (Insert), Read, Update, Delete Operations:

// Insert a purchase

let purchase=({

purchase\_id: "P2023001",

grn\_id: "GRN2023001",

supplier\_details: { name: "Dushyanth", contact: "123-456-7890" },

date: ISODate("2024-06-20"),

total: 50000,

paid: 5000,

balance: 0,

items: [

{ item\_id: "1", quantity: 10, unit\_price: 600, discount\_received: 0, free\_issues: 0 },

{ item\_id: "2", quantity: 5, unit\_price: 100, discount\_received: 0, free\_issues: 1 }

]

});

// Find all purchases

db.getCollection('purchases').find();

// Update paid amount for a purchase

db.getCollection('purchases').updateOne(

{ purchase\_id: "P2023001" },

{ $set: { paid: 40, balance: 10 } }

);

// Delete a purchase

db.getCollection('purchases').deleteOne({ purchase\_id: "P2023001" });

5.Daily Summary Including Sales and Purchases

* Daily Sales Summary

db.sales.aggregate([

{ $match: { date: { $gte: ISODate("2023-06-01T00:00:00Z"), $lt: ISODate("2023-06-02T00:00:00Z") } } },

{ $group: { \_id: null, totalSales: { $sum: "$total" }, totalPaid: { $sum: "$paid\_amount" }, totalBalance: { $sum: "$balance\_amount" } } }

]);

* Daily Summary Including Sales and Purchases

const salesSummary = db.sales.aggregate([

{ $match: { date: { $gte: ISODate("2023-06-01T00:00:00Z"), $lt: ISODate("2023-06-02T00:00:00Z") } } },

{ $group: { \_id: null, totalSales: { $sum: "$total" }, totalPaid: { $sum: "$paid\_amount" }, totalBalance: { $sum: "$balance\_amount" } } }

]);

const purchaseSummary = db.purchases.aggregate([

{ $match: { date: { $gte: ISODate("2023-06-01T00:00:00Z"), $lt: ISODate("2023-06-02T00:00:00Z") } } },

{ $group: { \_id: null, totalPurchases: { $sum: "$total" }, totalPaid: { $sum: "$paid" }, totalBalance: { $sum: "$balance" } } }

]);

const dailySummary = {

salesSummary: salesSummary.toArray(),

purchaseSummary: purchaseSummary.toArray()

};

printjson(dailySummary);

* Item-Based Sales Report

db.sales.aggregate([

{ $unwind: "$items" },

{ $group: { \_id: "$items.item\_id", totalQuantitySold: { $sum: "$items.quantity" }, totalRevenue: { $sum: { $multiply: ["$items.quantity", "$items.unit\_price"] } } } }

]);

* Indexing and Scaling Techniques

Indexing

Single Field Index: Create indexes on fields that are frequently queried, such as date, item\_id, and user\_id.

db.sales.createIndex({ date: 1 });

db.items.createIndex({ item\_id: 1 });

db.users.createIndex({ user\_id: 1 });

Compound Index: Create compound indexes for fields that are often queried together.

db.sales.createIndex({ date: 1, total: 1 });

Text Index: For fields that contain text and need to support text search.

db.items.createIndex({ name: "text" });

* Scaling

Sharding: Distribute data across multiple servers to support horizontal scaling. Shard collections on a field that ensures even data distribution, such as sale\_id.

sh.enableSharding("restaurant\_cafe\_db");

sh.shardCollection("restaurant\_cafe\_db.sales", { sale\_id: "hashed" });

Replication: Use replica sets to ensure high availability and fault tolerance.

rs.initiate({

\_id: "rs0",

members: [

{ \_id: 0, host: "mongodb0.example.net:27017" },

{ \_id: 1, host: "mongodb1.example.net:27017" },

{ \_id: 2, host: "mongodb2.example.net:27017" }

]

});

6.Strengths and solutions

1. Flexibility and Schema Design:

Schema-less Nature: MongoDB's schema-less nature allows for easy and dynamic handling of various data structures, making it ideal for the evolving needs of a restaurant and cafe's inventory and sales data.

Embedded Documents: The use of embedded documents (denormalization) simplifies data retrieval, reducing the need for complex joins and enhancing query performance.

2. Scalability:

Horizontal Scaling: MongoDB supports horizontal scaling through sharding, which allows the database to handle large volumes of data and high-traffic loads efficiently.

Replication: MongoDB's replica sets provide high availability and data redundancy, ensuring business continuity and data durability.

3. Performance:

Indexing: Creating indexes on frequently queried fields improves read performance significantly.

Aggregation Framework: MongoDB’s powerful aggregation framework supports complex data processing and reporting tasks, which is crucial for generating business insights.

4. Operational Efficiency:

Document-Oriented Storage: MongoDB's document-oriented storage aligns well with the hierarchical and nested data structures often used in POS and inventory systems.

Ease of Use: MongoDB's query language is user-friendly and similar to JSON, making it easy to implement and manage for developers.

5. Real-Time Analytics:

Real-Time Data Processing: MongoDB’s capabilities for real-time data processing and analytics support timely decision-making, which is essential for managing inventory, sales, and purchases in a restaurant and cafe setting.

**Limitations of the MongoDB Solution**

1. Data Consistency:

Eventual Consistency: MongoDB follows an eventual consistency model in distributed setups, which might not be suitable for applications requiring strict transactional consistency.

Lack of ACID Transactions: Although MongoDB supports multi-document ACID transactions, they are not as robust as those in traditional RDBMS systems, potentially leading to issues in scenarios requiring complex transactional operations.

2. Memory Usage:

High Memory Consumption: MongoDB can consume a significant amount of memory, especially with large datasets and extensive indexing. This might lead to higher infrastructure costs.

Denormalization Trade-Offs: While denormalization improves read performance, it can lead to data redundancy, increased storage requirements, and complexity in data updates.

3. Complex Querying:

Limited Support for Complex Joins: MongoDB does not natively support complex joins as efficiently as relational databases, which might complicate certain querying scenarios.

Aggregation Framework Limitations: Although powerful, the aggregation framework can become complex and resource-intensive for very large datasets or highly intricate queries.

4. Tooling and Ecosystem:

Limited Ecosystem Compared to RDBMS: The ecosystem of tools and extensions for MongoDB, while growing, is not as mature or extensive as that for traditional RDBMS systems.

Learning Curve: Developers familiar with SQL-based systems may face a learning curve when transitioning to MongoDB’s document-based query language and operational model.

5. Indexing and Sharding Complexity:

Index Management: Proper index management is crucial but can become complex as the dataset grows. Poorly designed indexes can degrade performance instead of improving it.

Sharding Challenges: Implementing and managing sharding can be complex and requires careful planning to avoid issues like hotspotting or uneven data distribution.

7.Conclusion

While MongoDB offers significant strengths such as flexibility, scalability, performance, and operational efficiency, it also presents limitations related to data consistency, memory usage, complex querying, tooling ecosystem, and the complexity of indexing and sharding. These factors should be carefully considered in the context of the specific requirements and constraints of the restaurant and cafe's POS and inventory management system.