**2 ( B )**

**Problem statement : Aggregation with suitable example using MongoDB**

**Step 1: Create a Sample Database and Collection**

use storeDatabase;

db.orders.drop(); // Clear the collection if it exists

db.orders.insertMany([

{

orderId: 1,

customerId: "C101",

items: [

{ productId: "P1", productName: "Laptop", category: "Electronics", price: 1000, quantity: 1 },

{ productId: "P2", productName: "Mouse", category: "Electronics", price: 50, quantity: 2 }

],

orderDate: new Date("2023-09-10"),

shippingAddress: "123 Main St, City A, 12345",

status: "Shipped"

},

{

orderId: 2,

customerId: "C102",

items: [

{ productId: "P3", productName: "Book", category: "Books", price: 20, quantity: 3 },

{ productId: "P4", productName: "Notebook", category: "Stationery", price: 5, quantity: 5 }

],

orderDate: new Date("2023-09-12"),

shippingAddress: "456 Elm St, City B, 67890",

status: "Delivered"

},

{

orderId: 3,

customerId: "C101",

items: [

{ productId: "P5", productName: "Headphones", category: "Electronics", price: 150, quantity: 1 }

],

orderDate: new Date("2023-09-13"),

shippingAddress: "123 Main St, City A, 12345",

status: "Pending"

}

]);

**Step 2: Aggregation Queries Using All Stages**

1. **$match - Filter documents based on a condition.**
2. **$group - Group documents by a specific field to perform aggregation operations.**
3. **$project - Shape the output, including/excluding specific fields or creating new fields.**
4. **$sort - Sort the documents by specified fields.**
5. **$limit - Limit the number of documents in the output.**
6. **$skip - Skip a specified number of documents.**
7. **$unwind - Deconstruct arrays into separate documents.**
8. **$lookup - Join collections to merge data from different collections.**
9. **$addFields - Add new fields or modify existing fields.**
10. **$out - Write the aggregation results to a new collection.**
11. **$count - Count the number of documents in the aggregation pipeline.**
12. **$facet - Perform multiple aggregations on the same dataset.**

db.orders.aggregate([

// 1. Match orders with status "Shipped" or "Delivered"

{

$match: {

status: { $in: ["Shipped", "Delivered"] }

}

},

// 2. Unwind items array to break out each item into a separate document

{

$unwind: "$items"

},

// 3. Group by customerId to calculate total spending per customer and average item price

{

$group: {

\_id: "$customerId",

totalSpent: { $sum: { $multiply: ["$items.price", "$items.quantity"] } },

averageItemPrice: { $avg: "$items.price" }

}

},

// 4. Add a field that categorizes customers as "High Spender" or "Low Spender" based on totalSpent

{

$addFields: {

spenderCategory: {

$cond: {

if: { $gte: ["$totalSpent", 500] },

then: "High Spender",

else: "Low Spender"

}

}

}

},

// 5. Project only customerId, totalSpent, and spenderCategory fields

{

$project: {

\_id: 0,

customerId: "$\_id",

totalSpent: 1,

spenderCategory: 1

}

},

// 6. Sort the results by totalSpent in descending order

{

$sort: {

totalSpent: -1

}

},

// 7. Limit the output to top 2 customers

{

$limit: 2

}

]);

**Additional Examples Using Other Aggregation Stages**

**Count the Number of Orders for Each Status**

db.orders.aggregate([

{ $group: { \_id: "$status", count: { $sum: 1 } } }

]);

**Facet Example: Get Total Orders per Category and the Average Price of Products per Category**

db.orders.aggregate([

{ $unwind: "$items" },

{

$facet: {

totalOrdersPerCategory: [

{ $group: { \_id: "$items.category", totalOrders: { $sum: 1 } } }

],

averagePricePerCategory: [

{ $group: { \_id: "$items.category", avgPrice: { $avg: "$items.price" } } }

]

}

}

]);

**Output Example: Save Aggregated High-Spenders Data to a New Collection**

db.orders.aggregate([

{ $match: { status: { $in: ["Shipped", "Delivered"] } } },

{ $unwind: "$items" },

{ $group: { \_id: "$customerId", totalSpent: { $sum: { $multiply: ["$items.price", "$items.quantity"] } } } },

{ $match: { totalSpent: { $gte: 500 } } }, // Filter only high spenders

{ $out: "highSpenders" } // Save results to new collection

]);

// Check the results in the "highSpenders" collection

db.highSpenders.find().pretty();

**2 ( C )**

**Problem statement: Map-reduce with suitable example using MongoDB. Objectives: To understand the basic map-reduce of MongoDB**

**Step 1: Create Sample Database and Collection**

use storeDatabase;

db.orders.drop(); // Clear collection if it exists

db.orders.insertMany([

{

orderId: 1,

customerId: "C101",

items: [

{ productId: "P1", productName: "Laptop", category: "Electronics", price: 1000, quantity: 1 },

{ productId: "P2", productName: "Mouse", category: "Electronics", price: 50, quantity: 2 }

],

orderDate: new Date("2023-09-10"),

status: "Shipped"

},

{

orderId: 2,

customerId: "C102",

items: [

{ productId: "P3", productName: "Book", category: "Books", price: 20, quantity: 3 },

{ productId: "P4", productName: "Notebook", category: "Stationery", price: 5, quantity: 5 }

],

orderDate: new Date("2023-09-12"),

status: "Delivered"

},

{

orderId: 3,

customerId: "C101",

items: [

{ productId: "P5", productName: "Headphones", category: "Electronics", price: 150, quantity: 1 }

],

orderDate: new Date("2023-09-13"),

status: "Pending"

}

]);

**Step 2: MapReduce Examples**

**Objective 1: Calculate Total Amount Spent Per Customer**

1. **Map Function**: Emit each item’s total price (price × quantity) for each customer.
2. **Reduce Function**: Sum up the total amounts for each customer.

// Map Function

const mapFunctionTotalSpent = function() {

this.items.forEach(item => {

emit(this.customerId, item.price \* item.quantity);

});

};

// Reduce Function

const reduceFunctionTotalSpent = function(customerId, amounts) {

return Array.sum(amounts);

};

// Execute MapReduce

db.orders.mapReduce(

mapFunctionTotalSpent,

reduceFunctionTotalSpent,

{ out: "totalSpentPerCustomer" }

);

// View Results

db.totalSpentPerCustomer.find().pretty();

**Objective 2: Find the Most Frequently Purchased Product Category**

1. **Map Function: Emit the category of each item purchased.**
2. **Reduce Function: Count the occurrences of each category.**
3. **Finalize Function: Not needed, but could be used for any further adjustments.**

// Map Function

const mapFunctionCategoryCount = function() {

this.items.forEach(item => {

emit(item.category, 1);

});

};

// Reduce Function

const reduceFunctionCategoryCount = function(category, counts) {

return Array.sum(counts);

};

// Execute MapReduce

db.orders.mapReduce(

mapFunctionCategoryCount,

reduceFunctionCategoryCount,

{ out: "categoryCounts" }

);

// View Results

db.categoryCounts.find().sort({ value: -1 }).limit(1).pretty();

**Objective 3: Find Highest and Lowest Purchase Amounts for Each Customer**

1. **Map Function: Emit each item’s total price (price × quantity) for each customer.**
2. **Reduce Function: Calculate the highest and lowest values for each customer.**

// Map Function

const mapFunctionMaxMinPurchase = function() {

this.items.forEach(item => {

emit(this.customerId, { total: item.price \* item.quantity });

});

};

// Reduce Function

const reduceFunctionMaxMinPurchase = function(customerId, values) {

let maxPurchase = -Infinity;

let minPurchase = Infinity;

values.forEach(value => {

maxPurchase = Math.max(maxPurchase, value.total);

minPurchase = Math.min(minPurchase, value.total);

});

return { max: maxPurchase, min: minPurchase };

};

// Execute MapReduce

db.orders.mapReduce(

mapFunctionMaxMinPurchase,

reduceFunctionMaxMinPurchase,

{ out: "customerMaxMinPurchase" }

);

// View Results

db.customerMaxMinPurchase.find().pretty();