EXPERIMENT 4.3

E-commerce Catalog with Nested Document Structure in MongoDB

CODE:-

Data Model Design

Schema Structure
```javascript
{
 _id: ObjectId,
 name: String,
 price: Number,
 category: String,
 description: String,
 variants: [
   {
     _id: String,
     color: String,
     size: String,
     stock: Number,
     sku: String
   }
 ],
 __v: Number
}
```

MongoDB Shell Commands

1. Database Setup
```javascript
// Connect to MongoDB and use the ecommerce database
use ecommerce

// Create the products collection (MongoDB creates it automatically when first document is inserted)
```

2. Sample Data Insertion

```javascript
// Insert sample products with nested variants
db.products.insertMany([
  {
    name: "Running Shoes",
    price: 120,
    category: "Footwear",
    description: "High-performance running shoes for athletes",
    variants: [
      {
```

```
      _id: "686f68ed2bf5384209b236af",
      color: "Red",
      size: "M",
      stock: 10,
      sku: "RS-RED-M-001"
    },
    {
      _id: "686f68ed2bf5384209b236b0",
      color: "Blue",
      size: "L",
      stock: 5,
      sku: "RS-BLU-L-001"
    }
  ],
  __v: 0
},
{
  name: "Smartphone",
  price: 699,
  category: "Electronics",
  description: "Latest smartphone with advanced features",
  variants: [
    {
      _id: "686f63eb90ac2728b3f11082",
      color: "Black",
      size: "128GB",
      stock: 15,
      sku: "SP-BLK-128-001"
    },
    {
      _id: "686f63eb90ac2728b3f11083",
      color: "White",
      size: "256GB",
      stock: 8,
      sku: "SP-WHT-256-001"
    }
  ],
  __v: 0
},
{
  name: "Winter Jacket",
  price: 200,
  category: "Apparel",
  description: "Warm winter jacket for cold weather",
  variants: [
    {
      _id: "686f68ed2bf5384209b236b3",
      color: "Black",
      size: "S",
      stock: 8,
      sku: "WJ-BLK-S-001"
    },
    {
      _id: "686f68ed2bf5384209b236b4",
      color: "Gray",
        size: "M",
        stock: 12,
        sku: "WJ-GRY-M-001"
      }
    ],
    __v: 0
  },
  {
    name: "Gaming Laptop",
    price: 1299,
    category: "Electronics",
    description: "High-performance gaming laptop",
    variants: [
      {
        _id: "686f63eb90ac2728b3f11084",
        color: "Black",
        size: "16GB RAM",
        stock: 3,
        sku: "GL-BLK-16-001"
      },
      {
        _id: "686f63eb90ac2728b3f11085",
        color: "Silver",
        size: "32GB RAM",
        stock: 2,
        sku: "GL-SLV-32-001"
      }
    ],
    __v: 0
  }
])
```

3. Basic Query Operations

Retrieve All Products
```javascript
// Get all products
db.products.find().pretty()

// Get all products with only specific fields
db.products.find({}, { name: 1, price: 1, category: 1 }).pretty()
```

Filter Products by Category
```javascript
// Get all Electronics products
db.products.find({ category: "Electronics" }).pretty()

// Get all Footwear products
db.products.find({ category: "Footwear" }).pretty()

// Get products from multiple categories
db.products.find({ category: { $in: ["Electronics", "Apparel"] } }).pretty()
```

Filter Products by Price Range
```javascript
// Products under $500
db.products.find({ price: { $lt: 500 } }).pretty()

// Products between $100 and $800
db.products.find({ price: { $gte: 100, $lte: 800 } }).pretty()
```

4. Working with Nested Variants

Query Products by Variant Color
```javascript
// Find products with Red variants
db.products.find({ "variants.color": "Red" }).pretty()

// Find products with Black variants
db.products.find({ "variants.color": "Black" }).pretty()
```

Query Products by Stock Availability
```javascript
// Find products with variants having stock > 10
db.products.find({ "variants.stock": { $gt: 10 } }).pretty()

// Find products with any variant out of stock
db.products.find({ "variants.stock": 0 }).pretty()
```

Complex Variant Queries
```javascript
// Find products with Blue variants in size L
db.products.find({
  "variants": {
    $elemMatch: {
      color: "Blue",
      size: "L"
    }
  }
}).pretty()

// Find Electronics with variants having stock > 5
db.products.find({
  category: "Electronics",
  "variants.stock": { $gt: 5 }
}).pretty()
```

5. Projection Queries

Project Specific Variant Details
```javascript
// Get only variant colors and stock for all products
db.products.find({}, {
  name: 1,
  "variants.color": 1,
  "variants.stock": 1
}).pretty()

// Get products with only in-stock variants
db.products.aggregate([
  {
    $project: {
      name: 1,
      price: 1,
      category: 1,
      variants: {
        $filter: {
          input: "$variants",
          cond: { $gt: ["$$this.stock", 0] }
        }
      }
    }
  }
])
```

6. Aggregation Pipeline Examples

Count Products by Category
```javascript
db.products.aggregate([
  {
    $group: {
      _id: "$category",
      count: { $sum: 1 },
      avgPrice: { $avg: "$price" }
    }
  }
])
```

Total Stock by Color
```javascript
db.products.aggregate([
  { $unwind: "$variants" },
  {
    $group: {
      _id: "$variants.color",
      totalStock: { $sum: "$variants.stock" },
      productCount: { $sum: 1 }
    }
  },
  { $sort: { totalStock: -1 } }
])
```

Products with Low Stock Variants
```javascript
```

```
db.products.aggregate([
  {
    $match: {
      "variants.stock": { $lt: 5 }
    }
  },
  {
    $project: {
      name: 1,
      category: 1,
      lowStockVariants: {
        $filter: {
          input: "$variants",
          cond: { $lt: ["$$this.stock", 5] }
        }
      }
    }
  }
])
```

7. Update Operations

Update Product Price
```javascript
// Update price for a specific product
db.products.updateOne(
  { name: "Running Shoes" },
  { $set: { price: 130 } }
)
```

Update Variant Stock
```javascript
// Update stock for a specific variant
db.products.updateOne(
  { "variants._id": "686f68ed2bf5384209b236af" },
  { $set: { "variants.$.stock": 15 } }
)

// Decrease stock for a variant (simulating a purchase)
db.products.updateOne(
  { "variants._id": "686f68ed2bf5384209b236af" },
  { $inc: { "variants.$.stock": -1 } }
)
```

Add New Variant to Product
```javascript
// Add a new variant to an existing product
db.products.updateOne(
  { name: "Running Shoes" },
  {
    $push: {
      variants: {
        _id: "686f68ed2bf5384209b236b5",
        color: "Green",
        size: "XL",
        stock: 7,
        sku: "RS-GRN-XL-001"
      }
    }
  }
)
```

Mongoose Implementation

If you're using Node.js with Mongoose, here's the corresponding schema and operations:

Schema Definition
```javascript
const mongoose = require('mongoose');

const variantSchema = new mongoose.Schema({
  _id: { type: String, required: true },
  color: { type: String, required: true },
  size: { type: String, required: true },
  stock: { type: Number, required: true, min: 0 },
  sku: { type: String, required: true, unique: true }
});

const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  price: { type: Number, required: true, min: 0 },
  category: { type: String, required: true },
  description: String,
  variants: [variantSchema]
});

const Product = mongoose.model('Product', productSchema);
```

Mongoose Query Examples
```javascript
// Find all products
const allProducts = await Product.find();

// Find products by category
const electronicsProducts = await Product.find({ category: 'Electronics' });

// Find products with specific variant color
const redProducts = await Product.find({ 'variants.color': 'Red' });

// Find products with low stock variants
const lowStockProducts = await Product.find({ 'variants.stock': { $lt: 5 } });

// Project specific fields
const productSummary = await Product.find({}, 'name price category variants.color
```

variants.stock');

Best Practices

1. **Index Strategy** : Create indexes on frequently queried fields
```javascript
db.products.createIndex({ category: 1 })
db.products.createIndex({ "variants.color": 1 })
db.products.createIndex({ "variants.stock": 1 })
```

2. **Validation** : Use MongoDB schema validation or Mongoose schemas to ensure data integrity

3. **Limit Nesting** : Keep nested arrays reasonably sized (MongoDB has a 16MB document limit)

4. **Use $elemMatch** : For complex queries on array elements with multiple conditions

5. **Consider Denormalization**: For frequently accessed data, consider duplicating information to avoid complex joins

This implementation demonstrates MongoDB's flexibility in handling complex, nested data structures while maintaining query performance and data integrity.