E-commerce Catalog with Nested Document Structure in MongoDB

Aim

To design and implement an E-commerce product catalog using MongoDB's nested document (embedded document) capabilities to model products, variants, categories, inventory, and reviews for efficient querying and scalability.

Objectives

- Understand when to use embedded (nested) documents vs references in MongoDB.
- Design a flexible product schema that supports variants, attributes, pricing, and stock information.
- Implement CRUD operations and common queries (filters, sorting, aggregations) on nested documents.
- Demonstrate indexing strategies and aggregation pipelines for reporting and product discovery.

Theory

MongoDB is a document-oriented NoSQL database that stores JSON-like documents (BSON). Nested documents (embedding) are a core feature that allows related data to be stored within the same document. Embedding is ideal when:
- The related data is frequently accessed together (e.g., product with its variants).
- The embedded array won't grow without bound (e.g., a finite number of variants or images).

Referencing (normalization) is preferred when:
- Related data is large, unbounded, or frequently updated independently (e.g., detailed user histories).

Trade-offs:
- Embedding improves read performance and atomic updates for the parent document but may increase document size.
- Proper indexing, projection, and aggregation are important for performance when working with nested structures.

Procedure

1. Identify the access patterns (search by category, filter by price/attributes, show product details + variants, update stock).
2. Design schema with embedded subdocuments for parts that are accessed together (variants, images, reviews preview) and references for large or shared collections (full review collection or global category tree if very deep).
3. Create indexes on frequently queried fields (e.g., `slug`, `categories`, `variants.sku`, `variants.price`, `attributes.key`).

4. Implement CRUD operations using a driver (Node.js + Mongoose shown below).
5. Use aggregation pipelines for advanced queries: flatten variants for search, compute average rating, and group by category for reports.
6. Test with sample documents and tune indexes.

Code (Example in Node.js with Mongoose)

```
// model/Product.js
const mongoose = require('mongoose');

const VariantSchema = new mongoose.Schema({
  sku: { type: String, required: true },
  title: String,
  price: Number,
  compareAtPrice: Number,
  inventory: {
    qty: Number,
    warehouseId: String
  },
  attributes: [{ key: String, value: String }] // e.g., color, size
}, { _id: false });

const ReviewPreviewSchema = new mongoose.Schema({
  userId: mongoose.Schema.Types.ObjectId,
  rating: Number,
  comment: String,
  createdAt: { type: Date, default: Date.now }
}, { _id: false });

const ProductSchema = new mongoose.Schema({
  title: { type: String, required: true },
  slug: { type: String, required: true, unique: true, index: true },
  description: String,
  brand: String,
  categories: [{ type: String, index: true }], // simple strings or ObjectId refs
  images: [String],
  variants: [VariantSchema], // embedded variants
  reviewPreview: [ReviewPreviewSchema], // small preview embedded
  metadata: {
    weight: Number,
    dimensions: { l: Number, w: Number, h: Number }
  },
```

```
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('Product', ProductSchema);

// controller/productController.js (selected operations)
const Product = require('../model/Product');

// Create product
exports.createProduct = async (req, res) => {
  const p = new Product(req.body);
  await p.save();
  res.json({ message: 'Product created', id: p._id });
};

// Get product by slug (project only needed fields)
exports.getProductBySlug = async (req, res) => {
  const slug = req.params.slug;
  const product = await Product.findOne({ slug }).select('title description images variants
reviewPreview');
  res.json(product);
};

// Search products by category + attribute + price range (simplified)
exports.search = async (req, res) => {
  const { category, attrKey, attrValue, minPrice, maxPrice } = req.query;
  const filter = {};
  if (category) filter.categories = category;
  if (attrKey && attrValue) filter['variants.attributes'] = { $elemMatch: { key: attrKey, value:
attrValue } };
  if (minPrice || maxPrice) {
    filter['variants.price'] = {};
    if (minPrice) filter['variants.price'].$gte = Number(minPrice);
    if (maxPrice) filter['variants.price'].$lte = Number(maxPrice);
  }
  // unwind + match + group using aggregation would be better; this is a simple find-based
approach
  const results = await Product.find(filter).select('title slug variants');
  res.json(results);
};

// Update variant inventory atomically (use positional operator)
```

```
exports.updateInventory = async (req, res) => {
  const { sku } = req.params;
  const { qty } = req.body;
  const result = await Product.updateOne(
    { 'variants.sku': sku },
    { $set: { 'variants.$.inventory.qty': qty } }
  );
  res.json(result);
};

// Aggregation example: compute average rating for products in a category
exports.avgRatingByCategory = async (req, res) => {
  const { category } = req.query;
  const pipeline = [
    { $match: { categories: category } },
    { $unwind: '$reviewPreview' },
    { $group: { _id: '$_id', avgRating: { $avg: '$reviewPreview.rating' }, title: { $first: '$title' } }
},
    { $sort: { avgRating: -1 } }
  ];
  const agg = await Product.aggregate(pipeline);
  res.json(agg);
};
```

Sample Document (Product)

```
{
  _id: ObjectId('...'),
  title: 'Classic White T-Shirt',
  slug: 'classic-white-tshirt',
  description: 'Comfortable cotton t-shirt',
  brand: 'Acme',
  categories: ['Apparel', 'T-Shirts'],
  images: ['img1.jpg','img2.jpg'],
  variants: [
    { sku: 'TSHIRT-WHT-S', title: 'Small', price: 299, inventory: { qty: 120 }, attributes:
[{key:'color', value:'white'}, {key:'size', value:'S'}] },
    { sku: 'TSHIRT-WHT-M', title: 'Medium', price: 299, inventory: { qty: 80 }, attributes:
[{key:'color', value:'white'}, {key:'size', value:'M'}] }
  ],
  reviewPreview: [ { userId: ObjectId('...'), rating: 4, comment: 'Good quality' } ]
}
```

Common Queries & Aggregations

1) Find product by sku (search inside embedded array):

```
db.products.find({ 'variants.sku': 'TSHIRT-WHT-S' })
```

2) Get all variants with price <= 500 using aggregation unwind:

```
db.products.aggregate([
  { $unwind: '$variants' },
  { $match: { 'variants.price': { $lte: 500 } } },
  { $project: { title: 1, 'variants.sku': 1, 'variants.price': 1 } }
])
```

3) Compute average rating per product:

```
db.products.aggregate([
  { $unwind: '$reviewPreview' },
  { $group: { _id: '$_id', avgRating: { $avg: '$reviewPreview.rating' }, title: { $first: '$title' } }
}
])
```

Output (Sample)

- Product created with id: 64b9f7a2e4c2f1a0d5d4a2c3
- Search results: [{ title: 'Classic White T-Shirt', slug: 'classic-white-tshirt', variants: [...] }]
- Aggregation avgRating result: [{ _id: ObjectId('...'), avgRating: 4.2, title: 'Classic White T-Shirt' }]

Learning Outcomes

1. Learn to model e-commerce products using MongoDB's embedded documents and understand the trade-offs with referencing.
2. Gain practical experience writing queries and aggregation pipelines for nested structures and product search scenarios.
3. Understand indexing strategies and update patterns (atomic updates on nested arrays) to maintain performance and data integrity.