🗄️ Day 12: NoSQL Databases - MongoDB for Modern Data Engineering

📚 What You'll Learn Today (Concept-First Approach)

Primary Focus: Understanding NoSQL principles and document database design for data engineering **Secondary Focus:** Hands-on MongoDB implementation with aggregation pipelines **Dataset for Context:** Amazon Products Dataset from Kaggle for varied schema exploration

🎯 Learning Philosophy for Day 12

"Understand the data structure before choosing the database"

We'll start with NoSQL concepts, explore document modeling principles, understand MongoDB's architecture, and build production-ready data pipelines for flexible, evolving datasets.

🌟 The NoSQL Revolution: Why Document Databases Matter

🤔 The Problem: Rigid Schema Limitations

Scenario: You're building a product catalog system for an e-commerce platform...

With Traditional SQL:

```
Products Table:
+----+-------+-------+--------+----------+
| ID | Name  | Price | Color  | Category |
+----+-------+-------+--------+----------+
| 1  | Phone | 699   | Black  | Tech     |
| 2  | Shirt | 29    | Blue   | Clothing |
+----+-------+-------+--------+----------+

❌ Problems:
- Fixed schema for diverse product types
- NULL values for non-applicable attributes
- Complex changes require schema migrations
- Difficult to handle varying product specifications
- Rigid structure doesn't match real-world data
```

With MongoDB Document Model:

```json
// Flexible product documents
{
  "_id": "product_1",
  "name": "iPhone 15 Pro",
  "price": 999,
  "category": "Electronics",
  "specifications": {
    "storage": "256GB",
    "color": "Deep Purple",
    "camera": "48MP Triple Camera",
    "battery": "3274mAh"
  },
  "reviews": [
    {
      "user": "john_doe",
      "rating": 5,
      "comment": "Amazing phone!",
      "date": "2024-01-15"
    }
  ],
  "variants": [
    {"storage": "128GB", "price": 899},
    {"storage": "512GB", "price": 1199}
  ]
}

{
  "_id": "product_2",
  "name": "Cotton T-Shirt",
  "price": 25,
  "category": "Clothing",
  "specifications": {
    "material": "100% Cotton",
    "fit": "Regular",
    "care": "Machine washable"
  },
  "sizes": ["S", "M", "L", "XL"],
  "colors": ["White", "Black", "Navy"]
}
```

💡 The MongoDB Solution: Documents as Natural Data

Think of MongoDB like this:

- **Traditional SQL:** Filing cabinet with fixed-size folders
- **MongoDB:** Flexible storage where each item can have unique properties

🏗️ Understanding NoSQL Database Types (Visual Approach)

🎨 The NoSQL Landscape

```
┌─────────────────────────────────────────────────────────┐
│                  NOSQL DATABASE TYPES                   │
│                                                         │
│  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐     │
│  │  Document   │  │   Graph     │  │  Key-Value  │     │
│  │ (MongoDB)   │  │  (Neo4j)    │  │   (Redis)   │     │
│  │             │  │             │  │             │     │
│  │ JSON-like   │  │ Nodes &     │  │ Simple      │     │
│  │ Documents   │  │ Relations   │  │ Hash Maps   │     │
│  └─────────────┘  └─────────────┘  └─────────────┘     │
│                                                         │
│         ┌─────────────┐                                 │
│         │ Columnar    │                                 │
│         │ (Cassandra) │                                 │
│         │             │                                 │
│         │ Wide Column │                                 │
│         │ Families    │                                 │
│         └─────────────┘                                 │
└─────────────────────────────────────────────────────────┘
```

🧠 When to Choose NoSQL vs SQL

Choose NoSQL (MongoDB) when:

- ✅ Schema evolves frequently
- ✅ Nested/hierarchical data structures
- ✅ Rapid development cycles
- ✅ Horizontal scaling requirements
- ✅ JSON/document-oriented applications

Choose SQL when:

- ✅ Fixed, well-defined schema

- ✅ Complex relationships and JOINs
- ✅ ACID transactions are critical
- ✅ Strong consistency requirements
- ✅ Mature ecosystem and tooling

🚀 MongoDB Installation and Setup (Visual Learning)

📱 Quick Start with Docker

Step 1: Create MongoDB Environment

```bash
# Create project structure
mkdir mongodb-data-engineering
cd mongodb-data-engineering

# Create directories
mkdir -p data/raw data/processed scripts notebooks
```

Step 2: Docker Compose Setup

```yaml
version: '3.8'
services:
  mongodb:
    image: mongo:7.0
    container_name: mongodb-dev
    restart: unless-stopped
    ports:
      - "27017:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: password
      MONGO_INITDB_DATABASE: ecommerce
    volumes:
      - mongodb_data:/data/db
      - ./scripts:/docker-entrypoint-initdb.d/

  mongo-express:
    image: mongo-express:latest
    container_name: mongo-express
    restart: unless-stopped
    ports:
      - "8081:8081"
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: admin
      ME_CONFIG_MONGODB_ADMINPASSWORD: password
      ME_CONFIG_MONGODB_URL: mongodb://admin:password@mongodb:27017/
    depends_on:
      - mongodb

volumes:
  mongodb_data:
```

Step 3: Launch MongoDB

```bash
# Start MongoDB and Mongo Express
docker-compose up -d

# Verify MongoDB is running
docker-compose ps

# Access Mongo Express Web UI
# http://localhost:8081
```

🎮 First Look at MongoDB Tools

Main Interface Options:

1. **Mongo Express (Web UI):**
   - Database and collection browser
   - Document viewer and editor
   - Query interface
   - Index management
2. **MongoDB Compass (Desktop):**
   - Visual query builder
   - Schema analysis
   - Performance monitoring
   - Aggregation pipeline builder
3. **MongoDB Shell (Command Line):**
   - Direct database interaction
   - Script execution
   - Administrative tasks

📊 Understanding Documents and Collections

🧩 Document Fundamentals (Visual Learning)

What is a Document?

```json
// A document is like a JSON object with additional data types
{
  "_id": ObjectId("..."),           // Unique identifier
  "product_name": "Laptop",         // String
  "price": 999.99,                  // Number
  "in_stock": true,                 // Boolean
  "launch_date": ISODate("2024-01-01"), // Date
  "categories": ["Electronics", "Computers"], // Array
  "specifications": {               // Embedded document
    "cpu": "Intel i7",
    "ram": "16GB",
    "storage": "512GB SSD"
  },
  "reviews": [                      // Array of documents
    {
      "user": "alice",
      "rating": 5,
      "comment": "Great laptop!"
    }
  ]
}
```

Document Design Patterns:

1. **Embedding Pattern (Denormalization):**

```json
// Good for: Related data accessed together
{
  "_id": "order_123",
  "customer": {
    "name": "John Doe",
    "email": "john@email.com"
  },
  "items": [
    {"product": "Laptop", "price": 999, "qty": 1},
    {"product": "Mouse", "price": 25, "qty": 2}
  ],
  "total": 1049
}
```

2. **Referencing Pattern (Normalization):**

json

```json
// Good for: Large related data, many-to-many relationships
{
  "_id": "order_123",
  "customer_id": "customer_456",
  "item_ids": ["item_789", "item_101"],
  "total": 1049
}
```

📝 Working with Amazon Products Dataset

Step 1: Download and Prepare Data

Dataset Source: Amazon Products Dataset on Kaggle

Files in Dataset:

- `amazon_products.csv` - Product information with nested attributes
- Product categories, prices, ratings, descriptions

Step 2: Data Exploration and Import

Connect to MongoDB:

python

```python
import pymongo
import pandas as pd
import json
from datetime import datetime

# MongoDB connection
client = pymongo.MongoClient("mongodb://admin:password@localhost:27017/")
db = client["ecommerce"]
products_collection = db["products"]
```

Load and Transform Data:

python

```python
# Read the CSV dataset
df = pd.read_csv('data/raw/amazon_products.csv')

# Explore data structure
print("Dataset Info:")
print(f"Rows: {len(df)}")
print(f"Columns: {df.columns.tolist()}")
print(f"Sample data:\n{df.head()}")

# Data transformation for document structure
def transform_product_to_document(row):
    """Transform CSV row to MongoDB document"""
    return {
        "product_id": row.get('product_id'),
        "product_name": row.get('product_name'),
        "category": row.get('category'),
        "discounted_price": float(row.get('discounted_price', 0)),
        "actual_price": float(row.get('actual_price', 0)),
        "discount_percentage": row.get('discount_percentage'),
        "rating": float(row.get('rating', 0)),
        "rating_count": int(row.get('rating_count', 0)),
        "about_product": row.get('about_product'),
        "user_id": row.get('user_id'),
        "user_name": row.get('user_name'),
        "review_id": row.get('review_id'),
        "review_title": row.get('review_title'),
        "review_content": row.get('review_content'),
        "img_link": row.get('img_link'),
        "product_link": row.get('product_link'),
        "created_at": datetime.now()
    }

# Transform and insert data
documents = []
for _, row in df.iterrows():
    doc = transform_product_to_document(row)
    documents.append(doc)

# Insert in batches for better performance
batch_size = 1000
for i in range(0, len(documents), batch_size):
    batch = documents[i:i + batch_size]
    products_collection.insert_many(batch)
```

```python
    print(f"Inserted batch {i//batch_size + 1}")

print(f"Total documents inserted: {products_collection.count_documents({})}")
```

🔍 MongoDB Queries for Data Engineering

📊 Basic Queries and Data Exploration

1. Document Count and Basic Stats:

python

```python
# Total documents
total_products = products_collection.count_documents({})
print(f"Total products: {total_products}")

# Unique categories
categories = products_collection.distinct("category")
print(f"Product categories: {len(categories)}")

# Sample document structure
sample_doc = products_collection.find_one()
print("Sample document structure:")
for key in sample_doc.keys():
    print(f"  {key}: {type(sample_doc[key])}")
```

2. Filtering and Finding Documents:

python

```python
# Find products in specific price range
expensive_products = products_collection.find({
    "actual_price": {"$gte": 1000, "$lte": 5000}
}).limit(5)

for product in expensive_products:
    print(f"Product: {product['product_name']}, Price: ${product['actual_price']}")

# Find highly rated products
highly_rated = products_collection.find({
    "rating": {"$gte": 4.5},
    "rating_count": {"$gte": 100}
}).sort("rating", -1).limit(10)

print("\nTop rated products:")
for product in highly_rated:
    print(f"{product['product_name']}: {product['rating']} ({product['rating_count']}
```

3. Text Search and Pattern Matching:

python

```python
# Create text index for search
products_collection.create_index([
    ("product_name", "text"),
    ("about_product", "text"),
    ("category", "text")
])

# Search for products
search_results = products_collection.find({
    "$text": {"$search": "wireless bluetooth headphones"}
}).limit(5)

print("Search results for 'wireless bluetooth headphones':")
for product in search_results:
    print(f"- {product['product_name']}")
```

⚡ MongoDB Aggregation Pipeline for Analytics

🔄 Understanding Aggregation Concepts

Aggregation Pipeline Stages:

```
Data Flow: Collection → Stage 1 → Stage 2 → Stage 3 → Result

┌─────────────┐    ┌─────────────┐    ┌─────────────┐    ┌─────────────┐
│             │    │             │    │             │    │             │
│  Documents  │───▶│   $match    │───▶│   $group    │───▶│    $sort    │
│             │    │  (Filter)   │    │ (Aggregate) │    │   (Order)   │
└─────────────┘    └─────────────┘    └─────────────┘    └─────────────┘
```

📊 Business Analytics with Aggregation

1. Category Performance Analysis:

python

```python
# Category-wise product count and average ratings
category_analysis = products_collection.aggregate([
    {
        "$group": {
            "_id": "$category",
            "product_count": {"$sum": 1},
            "avg_rating": {"$avg": "$rating"},
            "avg_price": {"$avg": "$actual_price"},
            "total_reviews": {"$sum": "$rating_count"}
        }
    },
    {
        "$sort": {"product_count": -1}
    },
    {
        "$limit": 10
    }
])

print("Top Categories by Product Count:")
for category in category_analysis:
    print(f"Category: {category['_id']}")
    print(f"  Products: {category['product_count']}")
    print(f"  Avg Rating: {category['avg_rating']:.2f}")
    print(f"  Avg Price: ${category['avg_price']:.2f}")
    print(f"  Total Reviews: {category['total_reviews']}")
    print()
```

2. Price Range Distribution:

```python
# Product distribution by price ranges
price_distribution = products_collection.aggregate([
    {
        "$addFields": {
            "price_range": {
                "$switch": {
                    "branches": [
                        {"case": {"$lt": ["$actual_price", 50]}, "then": "Under $50"},
                        {"case": {"$lt": ["$actual_price", 100]}, "then": "$50-$100"},
                        {"case": {"$lt": ["$actual_price", 500]}, "then": "$100-$500"}
                        {"case": {"$lt": ["$actual_price", 1000]}, "then": "$500-$1000
                    ],
                    "default": "Over $1000"
                }
            }
        }
    },
    {
        "$group": {
            "_id": "$price_range",
            "count": {"$sum": 1},
            "avg_rating": {"$avg": "$rating"}
        }
    },
    {
        "$sort": {"count": -1}
    }
])

print("Price Range Distribution:")
for range_data in price_distribution:
    print(f"{range_data['_id']}: {range_data['count']} products (Avg Rating: {range_da
```

3. Advanced Analytics - Customer Sentiment:

python

```python
# Analyze review sentiment patterns
review_analysis = products_collection.aggregate([
    {
        "$match": {
            "review_content": {"$exists": True, "$ne": None}
        }
    },
    {
        "$addFields": {
            "review_length": {"$strLenCP": "$review_content"},
            "sentiment_score": {
                "$cond": {
                    "if": {"$gte": ["$rating", 4]},
                    "then": "positive",
                    "else": {
                        "$cond": {
                            "if": {"$gte": ["$rating", 3]},
                            "then": "neutral",
                            "else": "negative"
                        }
                    }
                }
            }
        }
    },
    {
        "$group": {
            "_id": "$sentiment_score",
            "count": {"$sum": 1},
            "avg_review_length": {"$avg": "$review_length"},
            "avg_rating": {"$avg": "$rating"}
        }
    }
])

print("Review Sentiment Analysis:")
for sentiment in review_analysis:
    print(f"Sentiment: {sentiment['_id']}")
    print(f"  Count: {sentiment['count']}")
    print(f"  Avg Review Length: {sentiment['avg_review_length']:.0f} characters")
    print(f"  Avg Rating: {sentiment['avg_rating']:.2f}")
    print()
```

🎯 Document Design Patterns for Data Engineering

🏗️ Schema Design Strategies

1. Product Catalog with Reviews (Embedding):

```python
# Restructure data with embedded reviews
def create_product_with_reviews():
    # Group reviews by product
    pipeline = [
        {
            "$group": {
                "_id": "$product_id",
                "product_name": {"$first": "$product_name"},
                "category": {"$first": "$category"},
                "actual_price": {"$first": "$actual_price"},
                "discounted_price": {"$first": "$discounted_price"},
                "rating": {"$first": "$rating"},
                "about_product": {"$first": "$about_product"},
                "reviews": {
                    "$push": {
                        "user_id": "$user_id",
                        "user_name": "$user_name",
                        "review_title": "$review_title",
                        "review_content": "$review_content",
                        "rating": "$rating"
                    }
                }
            }
        }
    ]

    # Create new collection with embedded reviews
    db["products_with_reviews"].drop()  # Clean start

    cursor = products_collection.aggregate(pipeline)
    structured_products = list(cursor)

    if structured_products:
        db["products_with_reviews"].insert_many(structured_products)
        print(f"Created {len(structured_products)} products with embedded reviews")

create_product_with_reviews()
```

2. Time-Series Pattern for Analytics:

python

```python
# Create daily product metrics collection
def create_daily_metrics():
    daily_metrics = products_collection.aggregate([
        {
            "$group": {
                "_id": {
                    "date": {"$dateToString": {"format": "%Y-%m-%d", "date": "$created_
                    "category": "$category"
                },
                "products_added": {"$sum": 1},
                "avg_price": {"$avg": "$actual_price"},
                "total_reviews": {"$sum": "$rating_count"}
            }
        },
        {
            "$project": {
                "_id": 0,
                "date": "$_id.date",
                "category": "$_id.category",
                "products_added": 1,
                "avg_price": 1,
                "total_reviews": 1,
                "created_at": datetime.now()
            }
        }
    ])

    # Insert into metrics collection
    db["daily_metrics"].drop()
    db["daily_metrics"].insert_many(list(daily_metrics))
    print("Daily metrics collection created")

create_daily_metrics()
```

🚀 Performance Optimization and Indexing

📈 Index Strategy for Data Engineering

1. Query Performance Analysis:

```python
# Analyze slow queries
def analyze_query_performance():
    # Enable profiling
    db.set_profiling_level(2)  # Profile all operations

    # Run a complex query
    result = products_collection.find({
        "category": "Electronics",
        "actual_price": {"$gte": 100, "$lte": 1000},
        "rating": {"$gte": 4.0}
    }).sort("rating", -1).limit(20)

    # Get execution stats
    explained = products_collection.find({
        "category": "Electronics",
        "actual_price": {"$gte": 100, "$lte": 1000},
        "rating": {"$gte": 4.0}
    }).explain()

    print("Query execution stats:")
    print(f"Documents examined: {explained['executionStats']['totalDocsExamined']}")
    print(f"Documents returned: {explained['executionStats']['totalDocsReturned']}")
    print(f"Execution time: {explained['executionStats']['executionTimeMillis']}ms")

analyze_query_performance()
```

2. Create Optimized Indexes:

python

```python
# Create compound indexes for common query patterns
def create_performance_indexes():
    indexes_to_create = [
        # Category and price range queries
        [("category", 1), ("actual_price", 1)],

        # Rating-based sorting
        [("rating", -1), ("rating_count", -1)],

        # Price range queries
        [("actual_price", 1)],

        # Category analysis
        [("category", 1), ("rating", 1)],

        # Text search
        [("product_name", "text"), ("about_product", "text")]
    ]

    for index_spec in indexes_to_create:
        try:
            products_collection.create_index(index_spec)
            print(f"Created index: {index_spec}")
        except Exception as e:
            print(f"Index creation failed for {index_spec}: {e}")

create_performance_indexes()

# Verify indexes
indexes = products_collection.list_indexes()
print("\nCurrent indexes:")
for index in indexes:
    print(f"- {index['name']}: {index.get('key', 'N/A')}")
```

🔄 Data Pipeline Patterns with MongoDB

🛠️ ETL Patterns for Document Databases

1. Change Data Capture Pattern:

python

```python
# Monitor collection changes for real-time processing
def monitor_product_changes():
    try:
        # Watch for changes in products collection
        with products_collection.watch() as stream:
            print("Monitoring product changes...")
            for change in stream:
                operation_type = change['operationType']

                if operation_type == 'insert':
                    print(f"New product added: {change['fullDocument']['product_name']}

                elif operation_type == 'update':
                    print(f"Product updated: {change['documentKey']['_id']}")

                elif operation_type == 'delete':
                    print(f"Product deleted: {change['documentKey']['_id']}")

    except KeyboardInterrupt:
        print("Monitoring stopped")

# Note: Change streams require replica set configuration
```

2. Batch Processing Pattern:

python

```python
# Process products in batches for analytics
def batch_process_products():
    batch_size = 1000
    processed_count = 0

    # Process products in batches
    cursor = products_collection.find({}).batch_size(batch_size)

    batch = []
    for product in cursor:
        # Add processing logic here
        processed_product = enrich_product_data(product)
        batch.append(processed_product)

        if len(batch) >= batch_size:
            # Process batch
            process_product_batch(batch)
            processed_count += len(batch)
            batch = []
            print(f"Processed {processed_count} products")

    # Process remaining products
    if batch:
        process_product_batch(batch)
        processed_count += len(batch)

    print(f"Total products processed: {processed_count}")

def enrich_product_data(product):
    """Add computed fields to product"""
    product['discount_amount'] = product['actual_price'] - product['discounted_price']
    product['discount_ratio'] = product['discount_amount'] / product['actual_price']
    return product

def process_product_batch(batch):
    """Process a batch of products"""
    # Could write to another collection, send to API, etc.
    enriched_collection = db["enriched_products"]
    enriched_collection.insert_many(batch)

batch_process_products()
```

📊 MongoDB vs SQL: Practical Comparison

🔍 Query Comparison Examples

SQL Query:

```sql
SELECT
    category,
    COUNT(*) as product_count,
    AVG(rating) as avg_rating,
    AVG(actual_price) as avg_price
FROM products
WHERE rating >= 4.0
    AND actual_price BETWEEN 100 AND 1000
GROUP BY category
ORDER BY product_count DESC
LIMIT 10;
```

MongoDB Equivalent:

python

```python
# MongoDB aggregation pipeline
pipeline = [
    {
        "$match": {
            "rating": {"$gte": 4.0},
            "actual_price": {"$gte": 100, "$lte": 1000}
        }
    },
    {
        "$group": {
            "_id": "$category",
            "product_count": {"$sum": 1},
            "avg_rating": {"$avg": "$rating"},
            "avg_price": {"$avg": "$actual_price"}
        }
    },
    {
        "$sort": {"product_count": -1}
    },
    {
        "$limit": 10
    }
]

result = products_collection.aggregate(pipeline)
```

📈 Performance Comparison

Scenario: Product Search and Analytics

Aspect	SQL (PostgreSQL)	MongoDB
Schema Flexibility	Fixed schema, migrations needed	Dynamic schema, instant changes
Nested Data	Complex JOINs, multiple tables	Natural document structure
Horizontal Scaling	Challenging, requires sharding	Built-in sharding support
Query Complexity	SQL joins can be complex	Aggregation pipelines intuitive
ACID Transactions	Full ACID compliance	Limited to document level
Development Speed	Slower for varying schemas	Faster for evolving requirements

🛡️ MongoDB Best Practices for Data Engineering

🎯 Document Design Guidelines

1. Embedding vs Referencing Decision Tree:

```
┌─────────────────────────────────────────┐
│         Document Design Decision        │
└─────────────────┬───────────────────────┘
                  │
         ┌────────▼─────────┐
         │ Data accessed    │
         │ together?        │
         └──┬───────────────┬┘
            │ Yes           │ No
        ┌───▼──────┐   ┌────▼─────┐
        │  Embed   │   │Reference │
        │documents │   │documents │
        └──────────┘   └──────────┘
```

2. Collection Naming Conventions:

```python
# Good naming practices
collections = {
    "products": "Main product catalog",
    "product_reviews": "Product reviews (if separate)",
    "daily_metrics": "Aggregated daily statistics",
    "user_sessions": "User activity tracking",
    "order_history": "Historical order data"
}

# Avoid
bad_names = [
    "data",          # Too generic
    "Products",      # Inconsistent casing
    "product-data",  # Hyphens in names
    "tbl_products"   # SQL-style naming
]
```

3. Index Strategy:

```python
def create_production_indexes():
    """Create indexes for production workloads"""

    # Frequently queried fields
    products_collection.create_index("category")
    products_collection.create_index("rating")

    # Compound indexes for common query patterns
    products_collection.create_index([
        ("category", 1),
        ("rating", -1),
        ("actual_price", 1)
    ])

    # Text search
    products_collection.create_index([
        ("product_name", "text"),
        ("about_product", "text")
    ])

    # Sparse indexes for optional fields
    products_collection.create_index(
        "review_content",
        sparse=True  # Only index documents with this field
    )

    print("Production indexes created successfully")

create_production_indexes()
```

🔄 Integration with Data Engineering Stack

🛠️ MongoDB with Apache Airflow

DAG for MongoDB ETL Pipeline:

python

```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta
import pymongo

default_args = {
    'owner': 'data-team',
    'depends_on_past': False,
    'start_date': datetime(2024, 1, 1),
    'retries': 2,
    'retry_delay': timedelta(minutes=5)
}

def extract_and_transform_products(**context):
    """Extract products and perform transformations"""
    client = pymongo.MongoClient("mongodb://admin:password@mongodb:27017/")
    db = client["ecommerce"]

    # Extract products needing processing
    products_to_process = db.products.find({
        "processed": {"$ne": True}
    }).limit(1000)

    processed_products = []
    for product in products_to_process:
        # Transform product data
        transformed_product = {
            **product,
            "price_category": categorize_price(product['actual_price']),
            "sentiment_score": analyze_sentiment(product.get('review_content')),
            "processed": True,
            "processed_at": datetime.now()
        }
        processed_products.append(transformed_product)

    # Update processed products
    for product in processed_products:
        db.products.update_one(
            {"_id": product["_id"]},
            {"$set": product}
        )

    return len(processed_products)
```

```python
def categorize_price(price):
    """Categorize products by price range"""
    if price < 50:
        return "budget"
    elif price < 200:
        return "mid-range"
    elif price < 1000:
        return "premium"
    else:
        return "luxury"

def analyze_sentiment(review_text):
    """Simple sentiment analysis"""
    if not review_text:
        return "neutral"

    positive_words = ["good", "great", "excellent", "amazing", "love"]
    negative_words = ["bad", "terrible", "awful", "hate", "worst"]

    text_lower = review_text.lower()
    positive_count = sum(1 for word in positive_words if word in text_lower)
    negative_count = sum(1 for word in negative_words if word in text_lower)

    if positive_count > negative_count:
        return "positive"
    elif negative_count > positive_count:
        return "negative"
    else:
        return "neutral"

def generate_daily_analytics(**context):
    """Generate daily analytics reports"""
    client = pymongo.MongoClient("mongodb://admin:password@mongodb:27017/")
    db = client["ecommerce"]

    # Create daily summary
    today = datetime.now().strftime("%Y-%m-%d")

    analytics = db.products.aggregate([
        {
            "$group": {
                "_id": None,
                "total_products": {"$sum": 1},
```

```
                "avg_rating": {"$avg": "$rating"},
                "categories": {"$addToSet": "$category"},
                "price_ranges": {
                    "$push": {
                        "$switch": {
                            "branches": [
                                {"case": {"$lt": ["$actual_price", 50]}, "then": "budg
                                {"case": {"$lt": ["$actual_price", 200]}, "then": "mid
                                {"case": {"$lt": ["$actual_price", 1000]}, "then": "pr
                            ],
                            "default": "luxury"
                        }
                    }
                }
            }
        }
    ])

    result = list(analytics)[0] if analytics else {}

    # Store daily summary
    daily_summary = {
        "date": today,
        "metrics": result,
        "created_at": datetime.now()
    }

    db.daily_summaries.insert_one(daily_summary)
    return result

# Create DAG
dag = DAG(
    'mongodb_product_analytics',
    default_args=default_args,
    description='MongoDB product analytics pipeline',
    schedule_interval='0 2 * * *',  # Daily at 2 AM
    catchup=False
)

# Define tasks
extract_transform_task = PythonOperator(
    task_id='extract_transform_products',
    python_callable=extract_and_transform_products,
    dag=dag
```

```python
)

analytics_task = PythonOperator(
    task_id='generate_daily_analytics',
    python_callable=generate_daily_analytics,
    dag=dag
)

# Set dependencies
extract_transform_task >> analytics_task
```

🐍 MongoDB with Apache Spark

Spark MongoDB Connector:

python

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

# Initialize Spark with MongoDB connector
spark = SparkSession.builder \
    .appName("MongoDBAnalytics") \
    .config("spark.mongodb.input.uri", "mongodb://admin:password@mongodb:27017/ecommer
    .config("spark.mongodb.output.uri", "mongodb://admin:password@mongodb:27017/ecomme
    .config("spark.jars.packages", "org.mongodb.spark:mongo-spark-connector_2.12:3.0.1
    .getOrCreate()

# Read from MongoDB
df = spark.read \
    .format("mongo") \
    .option("database", "ecommerce") \
    .option("collection", "products") \
    .load()

# Perform analytics
category_analysis = df.groupBy("category") \
    .agg(
        count("*").alias("product_count"),
        avg("rating").alias("avg_rating"),
        avg("actual_price").alias("avg_price"),
        sum("rating_count").alias("total_reviews")
    ) \
    .orderBy(desc("product_count"))

# Write results back to MongoDB
category_analysis.write \
    .format("mongo") \
    .option("database", "ecommerce") \
    .option("collection", "category_analytics") \
    .mode("overwrite") \
    .save()

print("Spark MongoDB analytics completed")
```

📊 Real-Time Analytics Patterns

🔄 Change Streams for Real-Time Processing

Real-Time Product Monitoring:

python

```python
import asyncio
from motor.motor_asyncio import AsyncIOMotorClient
from datetime import datetime

async def real_time_product_monitor():
    """Monitor product changes in real-time"""

    # Async MongoDB client
    client = AsyncIOMotorClient("mongodb://admin:password@localhost:27017/")
    db = client["ecommerce"]
    collection = db["products"]

    print("Starting real-time product monitoring...")

    # Watch for changes
    async with collection.watch() as stream:
        async for change in stream:
            await process_product_change(change, db)

async def process_product_change(change, db):
    """Process individual product changes"""
    operation = change['operationType']

    if operation == 'insert':
        # New product added
        product = change['fullDocument']
        await handle_new_product(product, db)

    elif operation == 'update':
        # Product updated
        product_id = change['documentKey']['_id']
        updated_fields = change.get('updateDescription', {}).get('updatedFields', {})
        await handle_product_update(product_id, updated_fields, db)

async def handle_new_product(product, db):
    """Handle new product insertion"""
    print(f"New product: {product['product_name']} in {product['category']}")

    # Update category statistics
    await db.category_stats.update_one(
        {"category": product['category']},
        {
            "$inc": {"product_count": 1},
```

```python
            "$push": {
                "recent_products": {
                    "name": product['product_name'],
                    "price": product['actual_price'],
                    "added_at": datetime.now()
                }
            }
        },
        upsert=True
    )

async def handle_product_update(product_id, updated_fields, db):
    """Handle product updates"""
    if 'rating' in updated_fields:
        print(f"Product {product_id} rating updated to {updated_fields['rating']}")

        # Update rating analytics
        await db.rating_updates.insert_one({
            "product_id": product_id,
            "new_rating": updated_fields['rating'],
            "updated_at": datetime.now()
        })

# Run the real-time monitor
# asyncio.run(real_time_product_monitor())
```

📈 Aggregation Pipeline for Business Intelligence

Advanced Analytics Queries:

python

```python
def create_business_intelligence_views():
    """Create MongoDB views for business intelligence"""

    # 1. Product Performance Dashboard
    product_performance_pipeline = [
        {
            "$addFields": {
                "revenue_estimate": {
                    "$multiply": ["$discounted_price", "$rating_count"]
                },
                "discount_impact": {
                    "$divide": [
                        {"$subtract": ["$actual_price", "$discounted_price"]},
                        "$actual_price"
                    ]
                }
            }
        },
        {
            "$group": {
                "_id": "$category",
                "total_products": {"$sum": 1},
                "avg_rating": {"$avg": "$rating"},
                "total_revenue_estimate": {"$sum": "$revenue_estimate"},
                "avg_discount": {"$avg": "$discount_impact"},
                "top_product": {
                    "$max": {
                        "rating": "$rating",
                        "name": "$product_name",
                        "price": "$discounted_price"
                    }
                }
            }
        },
        {
            "$sort": {"total_revenue_estimate": -1}
        }
    ]

    # Create view
    db.create_collection("product_performance_view", viewOn="products", pipeline=produ

    # 2. Customer Sentiment Analysis
```

```
sentiment_pipeline = [
    {
        "$match": {
            "review_content": {"$exists": True, "$ne": None}
        }
    },
    {
        "$addFields": {
            "sentiment": {
                "$switch": {
                    "branches": [
                        {
                            "case": {"$gte": ["$rating", 4]},
                            "then": "positive"
                        },
                        {
                            "case": {"$gte": ["$rating", 3]},
                            "then": "neutral"
                        }
                    ],
                    "default": "negative"
                }
            },
            "review_length_category": {
                "$switch": {
                    "branches": [
                        {
                            "case": {"$lt": [{"$strLenCP": "$review_content"}, 50]
                            "then": "short"
                        },
                        {
                            "case": {"$lt": [{"$strLenCP": "$review_content"}, 200
                            "then": "medium"
                        }
                    ],
                    "default": "long"
                }
            }
        }
    },
    {
        "$group": {
            "_id": {
                "category": "$category",
```

```python
                    "sentiment": "$sentiment",
                    "review_length": "$review_length_category"
                },
                "count": {"$sum": 1},
                "avg_rating": {"$avg": "$rating"}
            }
        }
    ]

    db.create_collection("sentiment_analysis_view", viewOn="products", pipeline=sentim

    print("Business intelligence views created successfully")

create_business_intelligence_views()

# Query the views
def query_business_intelligence():
    """Query the created BI views"""

    print("=== Product Performance by Category ===")
    for doc in db.product_performance_view.find().limit(5):
        print(f"Category: {doc['_id']}")
        print(f"  Products: {doc['total_products']}")
        print(f"  Avg Rating: {doc['avg_rating']:.2f}")
        print(f"  Revenue Estimate: ${doc['total_revenue_estimate']:,.2f}")
        print(f"  Avg Discount: {doc['avg_discount']:.1%}")
        print()

    print("=== Sentiment Analysis ===")
    sentiment_results = db.sentiment_analysis_view.aggregate([
        {
            "$group": {
                "_id": "$_id.sentiment",
                "total_reviews": {"$sum": "$count"},
                "categories": {"$addToSet": "$_id.category"}
            }
        },
        {"$sort": {"total_reviews": -1}}
    ])

    for sentiment in sentiment_results:
        print(f"Sentiment: {sentiment['_id']}")
        print(f"  Total Reviews: {sentiment['total_reviews']}")
        print(f"  Categories: {len(sentiment['categories'])}")
```

```python
        print()

query_business_intelligence()
```

🔧 MongoDB Administration for Data Engineers

🛡️ Database Security and User Management

Setting Up Authentication:

python

```python
def setup_database_security():
    """Configure MongoDB security for production"""

    # Connect as admin
    admin_client = pymongo.MongoClient("mongodb://admin:password@localhost:27017/")
    admin_db = admin_client["admin"]

    # Create application-specific users
    users_config = [
        {
            "user": "data_engineer",
            "pwd": "secure_password_123",
            "roles": [
                {"role": "readWrite", "db": "ecommerce"},
                {"role": "read", "db": "analytics"}
            ]
        },
        {
            "user": "analyst",
            "pwd": "analyst_password_456",
            "roles": [
                {"role": "read", "db": "ecommerce"},
                {"role": "read", "db": "analytics"}
            ]
        },
        {
            "user": "etl_service",
            "pwd": "etl_service_789",
            "roles": [
                {"role": "readWrite", "db": "ecommerce"},
                {"role": "readWrite", "db": "staging"}
            ]
        }
    ]

    ecommerce_db = admin_client["ecommerce"]

    for user_config in users_config:
        try:
            ecommerce_db.command("createUser", **user_config)
            print(f"Created user: {user_config['user']}")
        except Exception as e:
            print(f"Error creating user {user_config['user']}: {e}")
```

```python
# setup_database_security()
```

📊 Monitoring and Performance Tuning

Database Performance Monitoring:

python

```python
def monitor_database_performance():
    """Monitor MongoDB performance metrics"""

    client = pymongo.MongoClient("mongodb://admin:password@localhost:27017/")
    db = client["ecommerce"]

    # Get database statistics
    db_stats = db.command("dbStats")
    print("=== Database Statistics ===")
    print(f"Collections: {db_stats['collections']}")
    print(f"Data Size: {db_stats['dataSize'] / 1024 / 1024:.2f} MB")
    print(f"Index Size: {db_stats['indexSize'] / 1024 / 1024:.2f} MB")
    print(f"Storage Size: {db_stats['storageSize'] / 1024 / 1024:.2f} MB")

    # Collection-level statistics
    for collection_name in db.list_collection_names():
        collection = db[collection_name]
        stats = db.command("collStats", collection_name)

        print(f"\n=== {collection_name} Collection ===")
        print(f"Documents: {stats['count']}")
        print(f"Avg Document Size: {stats.get('avgObjSize', 0)} bytes")
        print(f"Total Size: {stats['size'] / 1024:.2f} KB")

        # Index usage
        index_stats = collection.aggregate([
            {"$indexStats": {}}
        ])

        print("Index Usage:")
        for index_stat in index_stats:
            print(f"  {index_stat['name']}: {index_stat['accesses']['ops']} operations'

def analyze_slow_queries():
    """Analyze slow queries using profiler"""

    client = pymongo.MongoClient("mongodb://admin:password@localhost:27017/")
    db = client["ecommerce"]

    # Enable profiling for slow operations (>100ms)
    db.set_profiling_level(1, slow_ms=100)

    # Run some test queries
```

```
    db.products.find({"category": "Electronics"}).sort("rating", -1).limit(10)
    db.products.find({"actual_price": {"$gte": 100}}).count()

    # Analyze profiler output
    profiler_data = db["system.profile"].find().sort("ts", -1).limit(5)

    print("=== Recent Slow Queries ===")
    for operation in profiler_data:
        print(f"Operation: {operation.get('command', 'N/A')}")
        print(f"Duration: {operation['millis']}ms")
        print(f"Documents Examined: {operation.get('docsExamined', 'N/A')}")
        print(f"Documents Returned: {operation.get('docsReturned', 'N/A')}")
        print("---")

monitor_database_performance()
# analyze_slow_queries()
```

🌐 MongoDB in Cloud Environments

☁️ MongoDB Atlas Integration

Connecting to MongoDB Atlas:

python

```python
import os
from urllib.parse import quote_plus

def setup_atlas_connection():
    """Setup connection to MongoDB Atlas"""

    # Atlas connection string (use environment variables in production)
    username = quote_plus("your_username")
    password = quote_plus("your_password")
    cluster_url = "your-cluster.mongodb.net"

    connection_string = f"mongodb+srv://{username}:{password}@{cluster_url}/ecommerce?

    try:
        client = pymongo.MongoClient(connection_string)

        # Test connection
        client.admin.command('ping')
        print("Successfully connected to MongoDB Atlas!")

        return client

    except Exception as e:
        print(f"Error connecting to Atlas: {e}")
        return None

# atlas_client = setup_atlas_connection()
```

📊 Data Lake Integration Pattern

MongoDB as Operational Database + S3 as Data Lake:

python

```python
import boto3
import json
from bson import json_util

def export_to_data_lake():
    """Export MongoDB data to S3 data lake"""

    # AWS S3 client
    s3_client = boto3.client('s3')
    bucket_name = 'your-data-lake-bucket'

    client = pymongo.MongoClient("mongodb://admin:password@localhost:27017/")
    db = client["ecommerce"]

    # Export products by category
    categories = db.products.distinct("category")

    for category in categories:
        print(f"Exporting {category} products...")

        # Get products for this category
        products = list(db.products.find({"category": category}))

        # Convert to JSON
        json_data = json_util.dumps(products, indent=2)

        # Create S3 key with partitioning
        s3_key = f"products/category={category.replace(' ', '_')}/data.json"

        # Upload to S3
        try:
            s3_client.put_object(
                Bucket=bucket_name,
                Key=s3_key,
                Body=json_data,
                ContentType='application/json'
            )
            print(f"Exported {len(products)} {category} products to S3")

        except Exception as e:
            print(f"Error uploading {category} to S3: {e}")
```

```python
    # export_to_data_lake()
```

🎯 Production Deployment Considerations

🏗️ MongoDB Replica Set Configuration

Production Setup with Docker Compose:

yaml

```yaml
version: '3.8'

services:
  mongo-primary:
    image: mongo:7.0
    container_name: mongo-primary
    command: mongod --replSet rs0 --bind_ip_all
    ports:
      - "27017:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: password
    volumes:
      - mongo-primary-data:/data/db
    networks:
      - mongo-cluster

  mongo-secondary:
    image: mongo:7.0
    container_name: mongo-secondary
    command: mongod --replSet rs0 --bind_ip_all
    ports:
      - "27018:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: password
    volumes:
      - mongo-secondary-data:/data/db
    networks:
      - mongo-cluster

  mongo-arbiter:
    image: mongo:7.0
    container_name: mongo-arbiter
    command: mongod --replSet rs0 --bind_ip_all
    ports:
      - "27019:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: password
    networks:
      - mongo-cluster
```

```yaml
volumes:
  mongo-primary-data:
  mongo-secondary-data:

networks:
  mongo-cluster:
    driver: bridge
```

🔄 Backup and Disaster Recovery

Automated Backup Strategy:

python

```python
import subprocess
from datetime import datetime
import os

def create_mongodb_backup():
    """Create MongoDB backup using mongodump"""

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    backup_dir = f"/backups/mongodb_{timestamp}"

    # Create backup directory
    os.makedirs(backup_dir, exist_ok=True)

    # MongoDB connection parameters
    host = "localhost"
    port = "27017"
    username = "admin"
    password = "password"
    database = "ecommerce"

    # Run mongodump
    dump_command = [
        "mongodump",
        "--host", f"{host}:{port}",
        "--username", username,
        "--password", password,
        "--db", database,
        "--out", backup_dir
    ]

    try:
        result = subprocess.run(dump_command, capture_output=True, text=True)

        if result.returncode == 0:
            print(f"Backup created successfully: {backup_dir}")

            # Compress backup
            tar_command = f"tar -czf {backup_dir}.tar.gz -C {backup_dir} ."
            subprocess.run(tar_command, shell=True)

            # Upload to S3 (optional)
            upload_backup_to_s3(f"{backup_dir}.tar.gz")
```

```python
        else:
            print(f"Backup failed: {result.stderr}")

    except Exception as e:
        print(f"Error creating backup: {e}")

def upload_backup_to_s3(backup_file):
    """Upload backup to S3"""
    import boto3

    s3_client = boto3.client('s3')
    bucket_name = 'mongodb-backups'

    try:
        s3_client.upload_file(backup_file, bucket_name, os.path.basename(backup_file))
        print(f"Backup uploaded to S3: {backup_file}")
    except Exception as e:
        print(f"Error uploading to S3: {e}")

# create_mongodb_backup()
```

📚 Essential Resources for Day 12

📖 Official Documentation

- **MongoDB Documentation:** https://docs.mongodb.com/
- **MongoDB University:** https://university.mongodb.com/
- **PyMongo Documentation:** https://pymongo.readthedocs.io/
- **MongoDB Best Practices:** https://docs.mongodb.com/manual/administration/production-notes/

🛠️ Tools and Libraries

- **MongoDB Compass:** GUI for MongoDB
- **Robo 3T:** Lightweight MongoDB GUI
- **PyMongo:** Python MongoDB driver
- **Motor:** Async Python MongoDB driver
- **MongoDB Connector for Spark:** Big data integration

📊 Sample Datasets for Practice

- **Amazon Products:** https://www.kaggle.com/datasets/jithinanievarghese/amazon-product-dataset

- **MongoDB Sample Datasets:** https://docs.atlas.mongodb.com/sample-data/
- **JSON Generator:** https://www.json-generator.com/ (for creating test data)

🎯 Key Takeaways for Data Engineers

Document Database Strengths: ✅ **Flexible Schema:** Adapt to changing requirements quickly ✅ **Natural Data Structure:** JSON documents match application objects ✅ **Horizontal Scaling:** Built-in sharding for large datasets ✅ **Rich Query Language:** Powerful aggregation framework ✅ **Developer Productivity:** Faster development cycles

When to Use MongoDB:

- Product catalogs with varying attributes
- Content management systems
- Real-time analytics applications
- IoT data collection
- Social media and user-generated content

When to Prefer SQL:

- Financial transactions requiring ACID compliance
- Complex relational data with many joins
- Reporting systems with fixed schemas
- Legacy system integration requirements

🚀 Tomorrow's Preview: Data Warehousing Concepts

Day 13 Focus: ETL vs ELT methodologies, data warehouse architecture patterns, and modern data stack design principles.

Preparation: Review today's MongoDB concepts and think about how document databases fit into larger data architecture patterns.