ST207 - Databases: Assignment 2

NoSQL Database Programming Report

Candidate Number: 73702

1 Question 1: MongoDB - Northwind Database

1.1 1A: Database Design and Relationship Mapping

The Northwind database uses a hybrid approach combining embedded and referenced relationships. The choice between embedding and referencing depends on access patterns, data mutability, and cardinality relationships.

Embedded Relationships:

- **Categories in Products:** Category information is embedded directly into product documents. Product catalog queries are read-heavy, and displaying a product almost always requires its category name. There are only 8 categories that rarely change, and each *product* belongs to exactly one category (while each category can have many products). From the product's point of view this behaves like a one-to-one lookup, so embedding avoids joins and speeds up reads.
- **Order Details in Orders:** Order details (line items) are embedded within order documents. Line items only make sense within an order context and are always accessed together. This follows MongoDB's standard "order pattern" and keeps order creation atomic without multi-document transactions.

Referenced Relationships:

- **Suppliers in Products (ObjectId reference):** Supplier information is referenced using ObjectId. Suppliers have more fields, change over time (addresses, contacts), and are shared by many products. Embedding would duplicate data and create update problems—updating a supplier's phone would require touching every product document. Referencing keeps data normalized.
- **Customers and Employees in Orders:** These are referenced using ObjectIDs. Customers and employees exist independently of orders, and a single customer can place hundreds of orders. Embedding their details in every order would bloat the collection and make profile updates painful. References handle this one-to-many relationship efficiently.

1.2 1B: Products by Supplier Query

Retrieves all product names and unit prices for each supplier. The aggregation pipeline mimics a SQL JOIN but works with MongoDB's document structure:

1. `$match`: Filters out products without suppliers to reduce dataset size.
2. `$lookup`: Left outer join with the `suppliers` collection to bring in supplier details.
3. `$unwind`: Flattens the supplier array (since `$lookup` returns an array) to access supplier fields.

4. `$group`: Groups by supplier, pushing product details into a nested array.

The output is one document per supplier with their name and a consolidated product list. The aggregation framework produces this denormalized, hierarchical view efficiently, which works well for reporting.

1.3 1C: Customer Purchase History Query

Generates a purchase history for each customer. The pipeline transforms the data in several stages:

1. `$unwind`: Deconstructs the `orderDetails` array, creating a separate document for each line item for product-level aggregation.
2. `$lookup`: Joins with the `customers` collection to attach customer details.
3. **First** `$group`: Groups by customer and product ID to calculate per-product totals (quantity and amount spent), applying discounts.
4. **Second** `$group`: Aggregates product-level summaries to the customer level, building the final list of purchased products.

The result joins three collections (Orders, Customers, Products) in one query, producing a customer-centric view with personal info and purchase statistics.

1.4 1D: Customer Sales Summary Materialized View

A materialized view collection `customer_sales_summary` is created using the `$out` stage. The pipeline computes key metrics: total orders, quantity purchased, revenue, and unique product categories. It also tracks customer lifetime value and engagement period using min/max order dates.

Strategic Value: Persisting these aggregations speeds up dashboard queries. Pre-computing avoids reprocessing the entire orders dataset for every summary request, which is essential for read-heavy analytical workloads.

2 Question 2: Neo4j - Movie Recommendation System

2.1 2A.1: Property Graph Data Model

The property graph model supports movie recommendation algorithms with efficient traversal patterns.

Node Design: Four node types: (1) **User** with properties `userId`, `age`, `gender`, and `zipCode` for demographic analysis; (2) **Movie** with `movieId`, `title`, and `releaseYear` extracted from the title string; (3) **Genre** with `name` property, modeled as a separate node for multi-genre movies and efficient genre-based queries; (4) **Occupation** with `name` property, separated to reduce redundancy and support occupation-based user segmentation.

Relationship Design: Four relationship types: (1) **RATED** (User→Movie) with properties `rating` (1-5) and `timestamp`, forming the core of collaborative filtering; (2) **BELONGS_TO** (Movie→Genre) for multi-genre classification; (3) **HAS_OCCUPATION** (User→Occupation) linking users to occupations; (4) **SIMILAR_TO** (Movie→Movie) with `totalScore` property, computed in query 2D based on genre overlap.

Design Rationale and Key Decisions:

- **Nodes vs. Properties: Genre** and **Occupation** are separate nodes, not properties. This turns shared values into structural connections. Queries like "Find all Engineers" or "Find all Action movies" use pointer traversal instead of index lookups, which is faster. Multi-valued attributes (e.g., a movie with 3 genres) also work more naturally this way.
- **Rating as a Relationship:** Ratings are properties on the `RATED` relationship. A rating isn't an entity—it's a qualified interaction between a User and a Movie. The relationship captures interaction metadata (score, timestamp).
- **Constraints:** Unique constraints on `userId`, `movieId`, `Genre.name`, and `Occupation.name` prevent duplicate nodes and create indexes that speed up `MATCH` clauses.

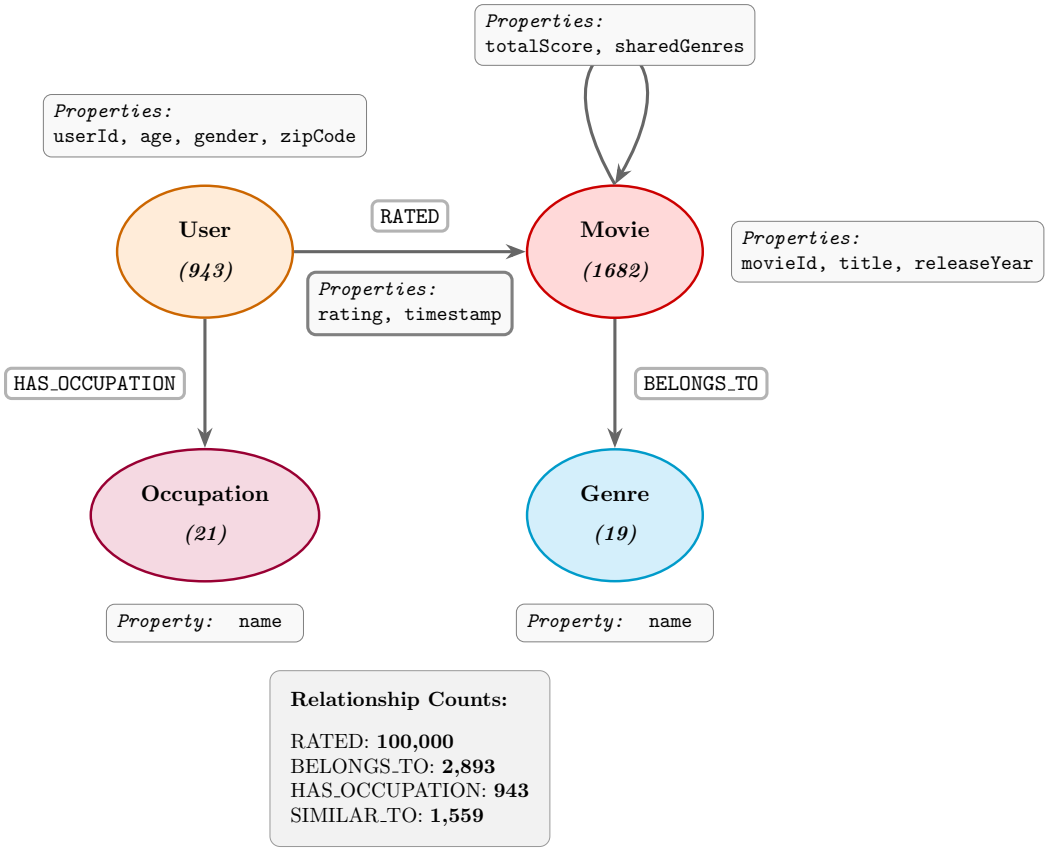

Figure 1: Property Graph Data Model for MovieLens 100K Movie Recommendation System

2.2 2B: Movie Recommendations Query

Implements collaborative filtering for movie recommendations.

Threshold Decision: "Highly rated" means rating ≥ 4 stars. This captures movies users actually enjoyed, filters out mediocre ratings (1-3 stars), and keeps enough data density for useful recommendations.

Algorithm: (1) Find movies the target user rated ≥ 4 stars; (2) Find other users who also rated those same movies ≥ 4 stars (similar users); (3) From similar users, find movies they rated ≥ 4 stars that the target user hasn't watched; (4) Rank by average rating and number of recommending users; (5) Return top 10 recommendations.

2.3 2C: Movie Triangles Query

Finds triples of movies sharing at least one genre, connected through users who rated all three highly.

Threshold Decision: "Highly rated" means rating > 4 stars (i.e., 5-star ratings only), following the assignment suggestion. This strict threshold captures strong user preference signals.

Query Strategy: Neo4j AuraDB free tier has a 250MB memory limit, so a staged approach is used: (1) identify the 15 most popular 5-star movies (those with ≥ 15 five-star ratings); (2) find all triangles (movie triples sharing a genre) among these; (3) verify each triangle has at least one user who rated all three movies > 4 stars.

Limitation: This may miss triangles with less popular movies. Focusing on highly-rated popular movies yields more meaningful recommendations. Results show 10 valid triangles, mostly in Drama, including *Braveheart*, *Pulp Fiction*, and *Shawshank Redemption*.

2.4 2D: Movie Similarity Computation

This query computes and stores SIMILAR_TO relationships using genre overlap.

Exploratory Analysis: Genre overlap distribution is analyzed to inform the threshold:

Shared Genres	Movie Pairs	Percentage
1	458,002	93.5%
2	30,230	6.2%
3	1,512	0.3%
4+	47	0.01%

Threshold Sensitivity Analysis: The threshold (≥ 3 shared genres) balances graph density and relevance:

- **Lower Threshold ($\geq$ 1 or 2):** A threshold of 1 creates edges between 93.5% of movie pairs (approx. 458,000 edges). The graph becomes too dense, causing "super-node" problems that make traversal expensive and recommendations less specific.
- **Higher Threshold (≥ 4):** Only 47 relationships result, which is too sparse for useful recommendation components.
- **Optimal Threshold (≥ 3):** Generates 1,559 edges (0.3% of pairs). These represent strong content similarity (e.g., Action-Adventure-SciFi movies) while keeping the graph sparse enough for fast traversal.

Computational Complexity: Naive pairwise comparison is $O(N^2)$, which is too expensive for large datasets. The Cypher query optimizes by matching on shared `Genre` nodes (`m1-[]->g<-[]-m2`), reducing the search space compared to a Cartesian product.

Similarity Score Calculation:

- `genreScore` = number of shared genres between two movies
- `actorScore` = **Not implemented** — the MovieLens 100K dataset does not include actor/cast information (only user ratings and movie genres are provided).

- `totalScore` = `genreScore` (since actorScore is unavailable).

Limitations and Future Work: The model only uses genre overlap, which is coarse. Two "Comedy-Drama" movies can have very different tones. With cast/director data, a weighted formula (e.g., $W_1 \cdot \text{genre} + W_2 \cdot \text{cast} + W_3 \cdot \text{director}$) could give finer-grained similarity. The Jaccard Index could also normalize scores against each movie's total genre count.

Relationships are stored with `totalScore` and `sharedGenres` properties, allowing $O(1)$ similarity lookups for recommendation queries.

3 Statement on Use of Generative AI Tools

I used Cursor to generate MongoDB aggregation pipelines and Neo4j Cypher queries.

- **Typing my own prompts based on the assignment requirements.** All prompts were formulated based on the specific questions (1B, 1C, 1D, 2B, 2C, 2D) and their requirements from the assignment brief.
- **Reviewing and modifying the generated code manually.** For MongoDB aggregation pipelines (e.g., `$lookup`, `$group`, `$out` stages), I reviewed the structure and adjusted the logic to match the Northwind database schema. For Neo4j Cypher queries, I modified pattern matching, traversal logic, and optimization strategies to fit the MovieLens dataset.
- **Executing and testing all queries in MongoDB Atlas and Neo4j AuraDB before including them in my notebooks.** All aggregation pipelines were tested in MongoDB Atlas, and all Cypher queries were executed in Neo4j AuraDB. The notebook outputs show the actual execution results, proving I tested everything myself.
- **AI functioned as a syntax assistant and code structure helper. All design decisions, validation, and understanding are my own.** The AI helped with MongoDB aggregation pipeline syntax and Neo4j Cypher query structure, but all database design decisions (embedding vs. referencing in MongoDB, node/relationship design in Neo4j), threshold choices (rating thresholds, similarity thresholds), and query logic are my own.
- **For MongoDB development (Question 1):** AI helped structure complex aggregation pipelines (e.g., multi-stage `$lookup` and `$group` operations, `$out` for materialized views), but I modified the pipelines to correctly handle the Northwind schema, verified all foreign key relationships, and tested each query to ensure correct output format matching the assignment requirements.
- **For Neo4j development (Question 2):** AI assisted with Cypher query syntax and graph traversal patterns (e.g., collaborative filtering queries, pattern matching for movie triangles, similarity computation algorithms), but I adapted the queries to fit the MovieLens dataset, optimized them for Neo4j AuraDB free tier limitations (e.g., staged approach for memory-intensive queries in 2C), and validated all results against the assignment requirements.
- **For data modeling decisions:** All decisions regarding MongoDB document structure (embedded vs. referenced relationships) and Neo4j property graph model (node types, relationship types, properties) were made by me, with AI only providing syntax suggestions.