# Schema Design Strategies and Scaling Solutions for E-Commerce in MongoDB

Dumitru Andreea-Alexandra, 344
Nedelcu Ionuț-Daniel, 342

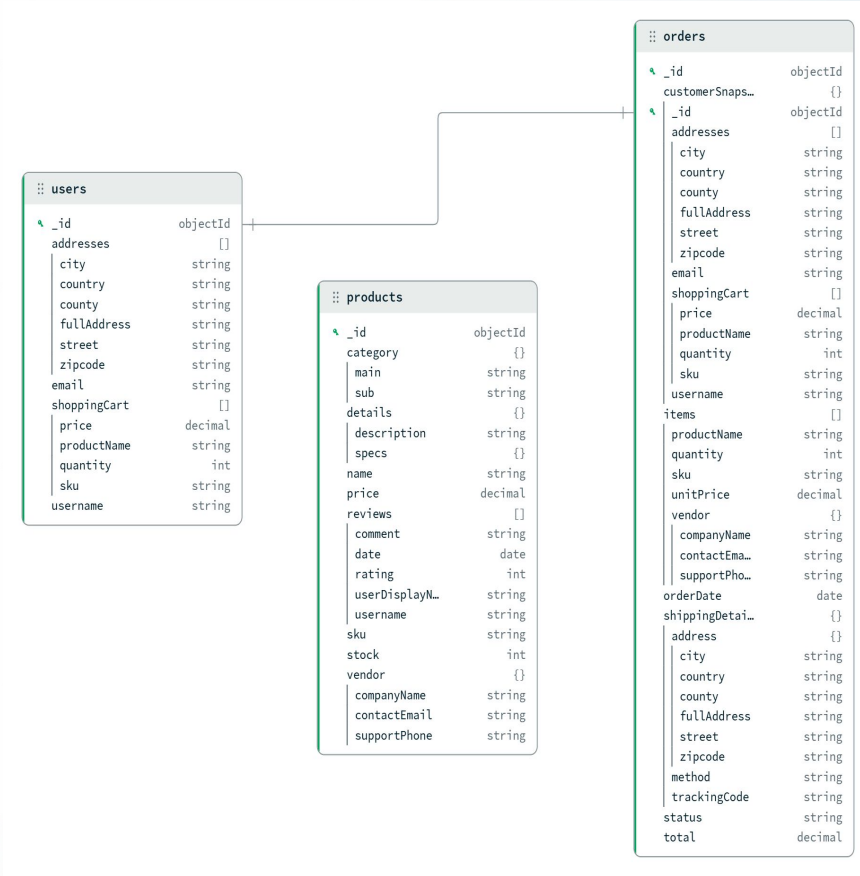
# Embedding Strategy

Choosing to embed everything is designed for extreme read performance and data isolation.

Denormalizing helps retrieve the entire entity someone wants to query in a single disk seek.

Advantage: the data in the database looks like the data the Frontend of an app would need

Trade-off: even if it offers high speed read, relying on embedding data makes the database write actions heavy (due to data redundancy).

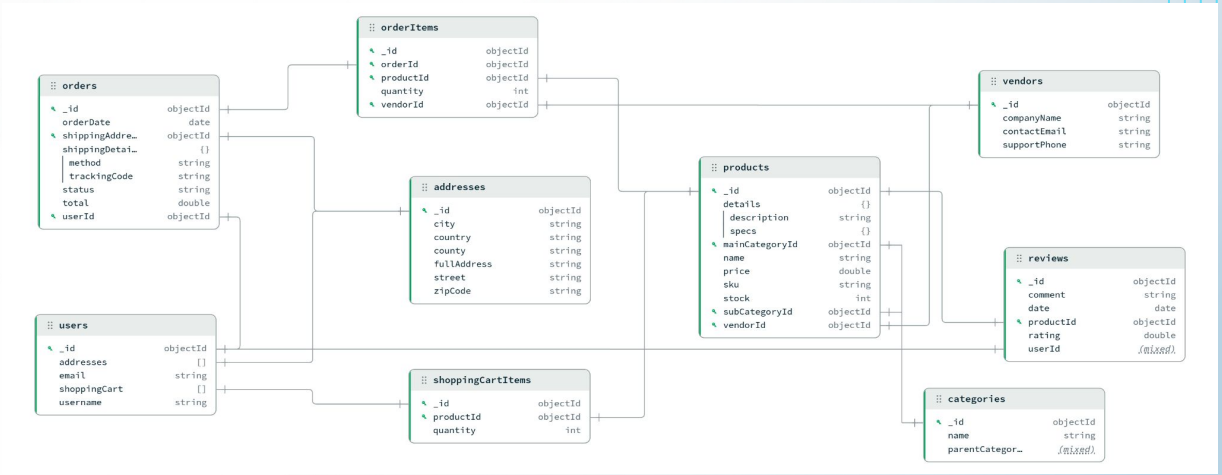


# Referencing Strategy

The Referencing Strategy implements a Normalized Data Model, effectively replicating a relational database structure (SQL-like) within MongoDB. As illustrated in the diagram, data is distributed across highly granular collections to ensure strict separation of concerns.

Advantage: it provides maximum Data Consistency. Because the data is normalized, an update to a master record is performed on a single document in the vendors collection and is instantly reflected logically across all products that reference it.

Trade-off: retrieving complex hierarchical data requires resource-intensive $lookup (JOIN) operations across multiple collections which significantly increases the execution time of aggregation pipelines => low read performance.

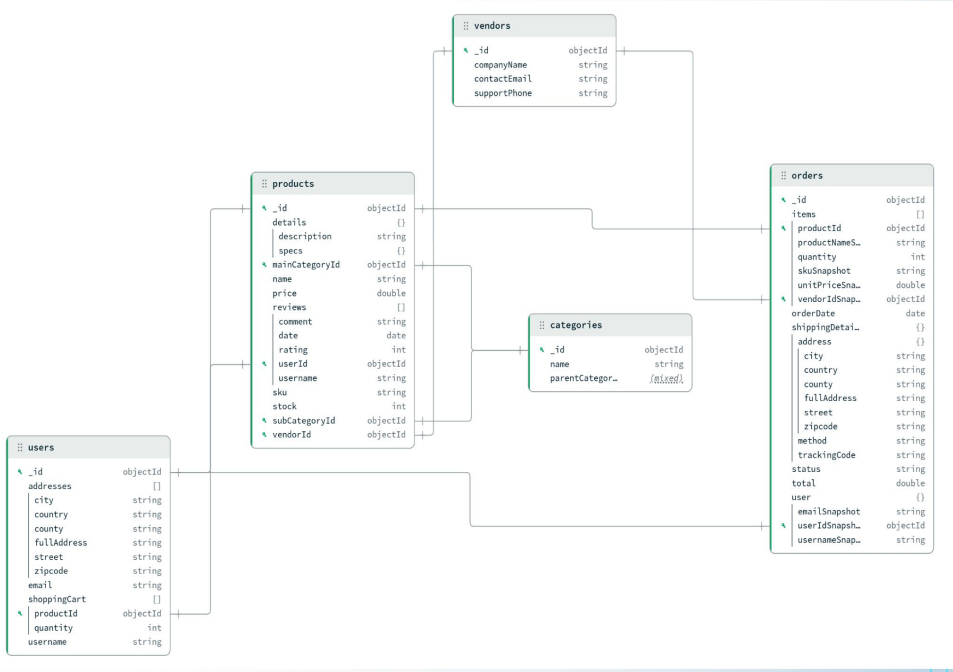


# Hybrid Strategy

A Hybrid Schema balances the speed of embedding and the scalability of referencing. It stores a limited amount of frequently accessed data OR high-volatility data, but also keeps the full detailed data in a separate collection, accessed through referencing

Most of the time pure embedding fails (documents grow endlessly), and so does pure referencing (queries and pipelines are too slow).

Unlike the "pure" approaches, a hybrid approach offers reduced IOPS (Input/Output Operations): the most used snippets of other objects/documents are probably already stored in the main document we need, but if we need all data (or a big chunk of it) we can look it up through referencing.



# Comparative Analysis

| | Embedding | Referencing | Hybrid |
|---|---|---|---|
| **Data Consistency** | Strong Consistency (Single Document) Weak Consistency (Duplicated Data) | Very Strong Consistency (Single Source of Truth) | Strong Consistency (Referenced Data) Weak Consistency (Embedded Data) |
| **Data Redundancy** | High & Uncontrolled | Very Low (Zero Duplication) | Partial & Strategic |
| **Performance** | Read: Very High Write: Low-Moderate | Read: Low Write: Very High | Read: High Write: Moderate |
| **Flexibility** | Low: data is "trapped" inside the parent | Very High: independent and granular data | Moderate: usually optimized for a specific structure |

# Indexes

Our indexing strategy was planned carefully: we created a core set of indexes based on how an application would use the data (like specific search queries), rather than just looking at the database structure.

We designed these indexes first for the Embedding model (which is the best fit for NoSQL) and then "translated" the same logic to the Referencing and Hybrid models to see if they worked there too.

This experiment taught us a key lesson: the same index works differently depending on how the data is organized.

While indexes made the Embedding model much faster, they barely helped the Referencing model. This proves that indexes are great at finding data quickly, but they cannot fix the slowness caused by joining many collections together (using $lookup).

# Queries

To analyze performance, we defined a set of 10 standard queries (Q1-Q10) representing common e-commerce actions, such as finding a product by its SKU, searching for a user by email, or filtering orders by status.

Just like with our indexing strategy, we defined these requirements based on the business logic of the Embedding model and then adapted the code to fit the Referencing and Hybrid schemas.

The results showed that for simple read operations, the database structure matters less than it does for complex reports. Across all three strategies, the execution times were remarkably similar and efficient.

This proves that if an application primarily needs to look up individual records without connecting deep webs of data, a Normalized (Referencing) architecture performs just as well as a Denormalized (Embedding) one.

# Aggregation Pipelines

To test the computational power of our database, we selected one complex aggregation pipeline for each of the three main collections defined in our initial Embedding strategy: **Products**, **Users**, and **Orders**.

## P1: Products

This pipeline filters for products with an average rating above 4 stars to test heavy data aggregation

## P2: Users

This targets users with a shopping cart value exceeding 100 to test real-time access (while an index drastically fixed the Embedding model, it failed to speed up the complex Referencing structure)

## P3: Orders

This analyzes vendors who generated more than 1000 in revenue, representing a standard business report. Even here, the Referencing model was significantly slower because it had to traverse connections between more collections

We implemented these pipelines first on the Embedding model, where data is kept together, and then adapted the logic for the Referencing and Hybrid models. This step revealed the true cost of "joins" in a non-relational database.

The difference was most visible in P1: while the Embedding model calculated ratings in milliseconds (because reviews are stored inside the product), the Referencing model took over 70 seconds to perform the exact same task, as it had to pull data from thousands of separate review documents.

# Sharding

To implement horizontal scaling, we configured a cluster where each database version (Embedding, Referencing, and Hybrid) is distributed across 3 separate shards.

These shards are managed by a single mongos router, which acts as the orchestrator, automatically directing read and write operations to the specific server holding the relevant data chunk.

To ensure a strictly valid comparison, we populated these sharded clusters with the exact same dataset used in our single-server MongoDB Atlas tests.

We then executed the identical suite of Queries (Q1-Q10) and Pipelines (P1-P3) to analyze specifically how physically splitting the data impacts execution time compared to the non-sharded environment.

# Concluding Facts

**140k+**

**Database Entries**
In total in the 3 different database approaches

**52X**

**Speed Boost**
P2 - Embedding
(from 17.3s to 0.3s)

**1700x**

**Faster**
P1 - Hybrid compared to Referencing

**0 Latency**

**60%**
of the test queries dropped to negligible (~0ms) execution time after indexing

**70 vs 0.7**

**Sharding**
the Referencing Database reduced the P1 time/node to 0.7s compared to the 71s seen in the non-sharded referencing environment.