Viktoriia Yuzkiv, Angelo di Gianvito; 29.04.2024

Setup for all models: n_people=50000, n_companies=1000

	Q1	Q2	Q3	Q4
M1	0.0576 seconds	9.8691 seconds	0.1989 seconds	0.0069 seconds
M2	0.000015 seconds	0.0339 seconds	0.1845 seconds	0.3329 seconds
M3	0.0053 seconds	0.000018 seconds	0.0226 seconds	0.0607 seconds

Assumptions made:

- Model 2: Since many people can work in one company, we created an array of companies first and then randomly assigned companies to each person.
- Model 3: Since we create the company document with the person data embedded, we need to know the number of people to assign. Therefore, we decided to take an average number of people per company (total number of employees/total number of companies, 50 in our case) for all companies.

1. Order queries from best to worst for Q1. Which model performs best? Why?

The order for this query of the best performing model: Model 2, Model 3, Model 1.

It makes sense that Model 2 is significantly faster for this query because it has the companies directly embedded for each person. This streamlined structure makes the query execution much faster compared to Model 1, where every person would need to reference external company documents, resulting in additional lookups and queries.

2. Order queries from best to worst for Q2. Which model performs best? Why?

The order for this query of the best performing model: Model 3, Model 2, Model 1.

In Model 3, where each company document contains an embedded array of employees, accessing the number of employees is efficient because it's directly embedded within the company document. This structure allows for faster query execution compared to Model 1, where checking the employees for each company would involve more computational operations and potentially slower performance due to external references. Therefore, Model 3 is more optimized for this query scenario.

3. Order queries from worst to best for Q3. Which model performs worst? Why?

The order for this query of the best performing model: Model 3, Model 2, Model 1.

Models 1 and 2 have very similar performances as the query is making the exact same operations in the two models. Surprisingly, Model 3 performs better than the other two models.

This means that MongoDB performs faster when it is operating on an array that is embedded into documents compared to analyzing them one by one like it does in Model 1 and 2. Following some research there are possible reasons for this. Model 3, where each company document embeds person documents, allows for efficient “bulk updates" by updating multiple persons (employees) within a single company document in one operation. MongoDB's handling of arrays optimizes this process, as it aggregates the updates within the document and leveraging data locality to reduce disk reads/writes and network operations compared to Models 1 and 2, which involve more complex querying or individual document updates across multiple collections or separate documents. The multiple reading and writing documents in the memory require more time compared to doing it a lower amount of times due to multiple documents stored in the same memory location.
This efficiency with bulk updates and data locality likely is the reason for the better performance of Model 3.

4. Order queries from worst to best for Q4. Which model performs worst? Why?

The order for this query of the best performing model: Model 1, Model 3, Model 2.

For this query it makes sense that model 2 is performing worse than the other models as it will have several duplicates to update as each person has the corresponding company embedded. This will cause a much higher number of operations compared to the case in which the company names are updated just once like in Models 3 and 1. Model 1 is performing better as normalization is better performing for these write-heavy operations as it is avoiding data redundancy and it ensures smaller document size compared to Model 3.

5. What are your conclusions about denormalization or normalization of data in MongoDB? In the case of updates, which offers better performance?

Conclusions about denormalization versus normalization in MongoDB depend on specific use cases and performance needs. Denormalization, combining related data into a single document or collection, can boost performance for read-heavy operations by reducing the need for joins or lookups, which is beneficial for efficient retrieval without redundancy or storage issues. Conversely, normalization, which separates data into multiple collections and uses IDs for relationships, is useful for write-heavy scenarios or when maintaining data integrity and flexibility is crucial.
Regarding updates, denormalization typically offers better performance by localizing updates within a single document, minimizing complex queries or references to related documents. This sometimes leads to faster updates compared to normalization, where updates may involve modifying multiple documents across different collections, but slower when it creates larger document sizes. However, the choice between denormalization and normalization should consider trade-offs related to data consistency, query complexity, and scalability based on the database's purpose. Each approach has its strengths and considerations that should be carefully evaluated based on the specific MongoDB use case in the long run.