Big Data Lab 2

23D020: Big Data Management for Data Science

Lab 2: Document Stores

Group: L2-T13
Marvin Ernst
Melisa Vadenja

Tuesday 27th May, 2025

C. Results and Discussion

Once the pipelines were executed with 50,000 total documents, the following execution times (in seconds) were measured for each query across the three data models. Note, that the exact times always varied a bit for each execution of the code, however, the order from best to worst was not affected for any of the queries.

Table 1: Query Execution Times per Model

Model	**Q1**	**Q2**	**Q3**	**Q4**
Model **1** (Normalized)	0.0096	1.6673	0.2778	0.0030
Model **2** (Person embeds Company)	0.0001	0.0418	0.3637	0.7121
Model **3** (Company embeds Persons)	0.0052	0.0092	0.0426	0.0870

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

Q1 (Read: Person + Company name) performs best in Model 2 (0.0001s), then Model 3 (0.0052s), and finally Model 1 (0.0096s). Model 2 is fastest because the data is already embedded in a single document, you can just access directly each persons name and their company without joins. Model 1 is slowest because it requires a `$lookup` to join the persons and companies collections, followed by `$unwind` to flatten the resulting array. Model 3 is also slower than Model 2 because while company and employee data are embedded together, you need to `$unwind` the persons array in each company document to access individual names.

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

Q2 (Count employees per company) is fastest in Model 3 (0.0092s), followed by Model 2 (0.0418s), and slowest in Model 1 (1.6673s). Model 3 outperforms others because employee lists are embedded in the company document, allowing a simple `$size` projection, in other words, each company is a main document and has their employees directly stored inside, i.e., persons are embedded in companies.

Model 1 is slowest due to costly `$lookup` and grouping steps across two collections. (Note, that we also show a second solution for this query, that does not account for if there is companies with zero employees, which is faster, however, Model 1 is still the slowest. Also, note, that this improvement we did is not really necessary since we created the companies, such that every company has about the same number of employees. For more details, see `model1.py`.) Model 2 is slower than Model 3 because the company data is embedded within each person document, making it necessary to scan all persons and group by the embedded company name to count employees.

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

Q3 (Update age for people born before 1988) performs worst in Model 2 (0.3637s), followed by Model 1 (0.2778s), and best in Model 3 (0.0426s). Model 2 is the slowest because it requires updating many individual person documents that each contain embedded company subdocuments (even though those subdocuments are not modified). Model 3 performs best by using efficient array filters to directly update only the relevant embedded person entries within company documents. Notably, when observing multiple runs, the performance of Model 1 and Model 3 tends to be very similar. This is expected, as the update queries for these two models are structurally quite similar.

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

Q4 (Append "Company" to company name) performs worst in Model 2 (0.7121s), followed by Model 3 (0.0870s), and is fastest in Model 1 (0.0030s). Model 2 is slowest because each person document embeds the company information, so the update must be applied repeatedly across many documents containing the same company. Model 1 is fastest because each company is stored once as a separate document, allowing the update to be applied only once per company without affecting any subdocuments. Interestingly, Model 1 performs significantly better than Model 3, even though both store companies as top-level documents and use the same update query structure. The likely explanation is that Model 3 also embeds a potentially large array of persons within each company, which may increase document size and update cost, even if those embedded arrays are not directly modified.

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

Denormalization, as implemented in Models 2 and 3, significantly improves query performance for read operations (Q1 and Q2), particularly when related data is frequently accessed together. However, this advantage comes at the cost of data duplication, which negatively affects update performance (Q3 and Q4). For example, the same company information is redundantly stored in every person document in Model 2, leading to costly repeated updates. In contrast, Model 1, which follows a normalized design, performs worse on read queries due to the need for joins, but enables more efficient and centralized updates to shared entities such as company names. Overall, for applications that are read-intensive and involve mostly static data, denormalized models (M2 or M3) offer better performance. Conversely, for workloads where updates are frequent or data consistency is critical, normalized models like Model 1 are more suitable.