Phase 2: Advanced Databases/Databases Technologies 2024/2025 Project

Group 11

Bruna Santos 56328
Department of Informatics, Faculdade de Ciências da Universidade de Lisboa, fc56328@alunos.ciencias.ulisboa.pt
Contribution to the project: Minor parts of the report documentation. Optimized NoSQL schema and indexed MongoDB. Re-inserted data into MongoDB and tested query times before and after indexing. Code implementation (MongoDB related tasks).
Percentage of contribution: 25%

Sofia Lopes 58175
Department of Informatics, Faculdade de Ciências da Universidade de Lisboa, fc58175@alunos.ciencias.ulisboa.pt
Contribution to the project: Majority of the report documentation. Tested performance improvements for both databases. Helped indexing MongoDB and testing query times after indexing. Code implementation (collaborative tasks for MongoDB).
Percentage of contribution: 25%

António Estêvão 58203
Department of Informatics, Faculdade de Ciências da Universidade de Lisboa, fc58203@alunos.ciencias.ulisboa.pt
Contribution to the project: Minor parts of the report documentation. Helped test query times before indexing for MySQL. Helped Indexing MySQL and tested performance improvements. Code implementation (collaborative tasks for MySQL).
Percentage of contribution: 25%

Diogo Venes 58216
Department of Informatics, Faculdade de Ciências da Universidade de Lisboa, fc58216@alunos.ciencias.ulisboa.pt
Contribution to the project: Minor parts of the report documentation. Optimized relational schema and re-processed data. Re-inserted data into MySQL and MongoDB, reset data for testing. Tested query times before and after indexing for both databases. Code implementation (MySQL related tasks).
Percentage of contribution: 25%

1. PHASE 1 EVOLUTIONS

Considering the development done in the Phase 1 of the project with both relational and NoSQL databases [1], and according to the project statement guidelines, we decided to make some modifications to their structures to improve their performance.

Relational Schema Improvements:

- Firstly, we didn't track query execution times in the Phase 1, so we ran query performance tests to establish a baseline for comparison with post-optimisation results.
- Secondly, we established the database connection using SQLAlchemy's *create_engine* function.
- Thirdly, we dropped the MySQL database *'BDAProject'*, effectively removing all previously created tables.
- Fourthly, we recreated the tables with the following modifications:
    - Redefined and created the *'listings'* table referencing the *'hosts'* table.
    - Split *'calendar'* into *'calendar_2024'* and *'calendar_2025'*, hoping to improve the performance of the queries, by reducing the dataset size used in them. We initially thought about splitting this table into *'calendar_availability'* and *'calendar_price'*, but found it wasn't too effective as the problem isn't the number of columns but rather the number of rows.
    - Created a new *'hosts'* table to comply with the Third Normal Form (3NF), with the attributes *'host_id'*, *'host_name'*, *'host_since'* and *'host_listings_count'*, reducing the duplication of data and redundancy.
    - The *'reviews'* table remained the same.

- Finally, we re-processed the data and reinserted it into MySQL using the *to_sql* method. This restructuring aimed to optimize the database schema by normalizing the data and separating the large calendar table, which should improve query performance in subsequent operations.

NoSQL Data Model Enhancements:

- Firstly, as with the relational schema, we didn't track query execution times in the Phase 1, so we ran query performance tests to establish a baseline for comparison with post-optimisation results.
- Secondly, we dropped the MongoDB database *'BDAProject'*, effectively removing all previously created collections.
- Thirdly, we thought about reducing the number of collections, so one of the options we came up with was to embed the *'calendar'* and *'reviews'* data into *'listings'*, hopefully reducing the number of join-like *'$lookup'* operations in the second complex query, allowing us to retrieve the data faster.
- Finally, we re-inserted the data into MongoDB.

We looked at the implementation of the simple and complex queries we defined in Phase 1 and decided to make some changes to the complex operations for MySQL:

- In the first complex query (Insert a new review to a listing whose host has been active since 2015):
  - The subquery now includes a *JOIN* with the *'hosts'* table.
  - The condition *YEAR(host_since) = 2015* now references the *'hosts'* table.
  - The *id* value for the new review is now generated randomly.
- In the second query (Delete the listings of the first 5 hosts with the highest number of 'Entire rental unit' (property_type) available on 2024-12-25):
  - The *'calendar'* table reference has been changed from *'calendar'* to *'calendar_2024'*.

And also some changes to the complex operations for MongoDB:

- In the first complex query (Insert a new review to a listing whose host has been active since 2015):
  - The collection reference has changed from *'collection_reviews'* to *'collection'*, as we now only have one collection.
  - Embedded reviews eliminate the need for a separate insert operation.
- In the second query (Delete the listings of the first 5 hosts with the highest number of 'Entire rental unit' (property_type) available on 2024-12-25):
  - The query now includes an availability check for 2024-12-25 using the *$elemMatch* operator.
  - The aggregation pipeline is now more concise, combining the matching and grouping stages.
  - The availability check is now part of the initial *$match* stage, filtering for "Entire rental unit" property type available on Christmas Day.
  - Eliminated the need for *$lookup* and *$unwind* operations.

For both MySQL and MongoDB the simple queries remained the same.

To further optimize our complex operations from Phase 1 and after we made changes to the database structures, we applied indexes to both databases:

MySQL Indexes:

- For the first complex query:
  - **hosts (host_since, host_id):** Compound index with the name *'index_host_since_id'* to optimize the first complex query involving the filtering of the host registration date `'WHERE YEAR(h.host_since) = 2015'` and ID `'JOIN hosts h ON l.host_id = h.host_id'`.
- For the second complex query:
  - **listings(host_id):** Index with the name *'index_host_id'* to improve the performance of the second complex query joining on host_id `'JOIN listings l ON c.listing_id = l.id'`.
  - **listings(host_id, property_type):** Compound index with the name *'index_host_property_type_num_listings'* to enhance the second complex query filtering on both host_id `'GROUP BY host_id'` and property_type `'WHERE property_type = 'Entire rental unit'`.

- **calendar_2024(date, available):** Compound index with the name *'index_date_availability'* to accelerate the filtering of the date-based availability of the property `'WHERE c.date = '2024-12-25' AND c.available = TRUE'` on the second complex query.

MongoDB Indexes:

- For the first complex query:
  - **host_since:** Unique index with the name *'index_host_since'* to improve the first complex query's performance since it allows faster retrieval of listings based on the host's start date.
- For the second complex query:
  - **property_type, calendar.date, host_id**: Compound index with the name *'index_host_property_availability'* to optimize the second query's filtering by `property_type,` to check availability based on `calendar.date`, and assisting in grouping by `host_id`.

These indexes were strategically chosen based on the fields accessed during the execution of our complex queries.

2. COMPARATIVE ANALYSIS

Now we can test the performance of our complex operations on both databases with pre-optimisation (Phase 1 of the project), after optimisation (after the database changes and before indexing) and after indexing (the database changes were already applied). Before comparing the performance of the different optimisations, it's important to note that the two examples of reported execution times were measured on different computers and that these times seemed to vary somewhat across consecutive runs on the same computer.

The following aspects have been highlighted for MySQL:

- For the first complex query:
  - For the two examples, we observed an increase in execution time after optimisation and a decrease in execution time after indexing.
  - In the first example (0.0073 → 0.0118), the query showed a slight increase in execution time, so optimization efforts did not consistently improve performance; this could be due to the addition of a *JOIN* operation between the *'listings'* and *'hosts'* tables in the subquery.
  - Indexing further slightly decreased the execution time for the second example (0.0124 → 0.0110).
- For the second complex query:
  - For both examples, we observed a decrease in execution time after optimisation and an increase in execution time after indexing.
  - In the first example (0.2195 → 0.1993), the query showed a slight decrease in execution time after optimization, this could be attributed to the reduced data volume in the split calendar tables, however, after indexing (0.1993 → 0.4669) a two-fold increase was detected, which we found very peculiar, given the fact that we associate indexing with reduced query times. This could be explained by index overhead. Deleting records requires updating multiple indexes, which can be time-consuming for complex queries involving joins and subqueries.
- We observed inconsistent performance execution times after indexing, with some queries showing increased execution times and others showing decreased execution times. This counterintuitive result can be attributed to the phenomenon of index overhead on the complex insertion and deleting operation [2]. While indexes generally improve query performance, excessive indexing can negatively impact database efficiency. It is important to note that, to confirm this theory, we tested a modified version of the second complex query that simply retrieved the listings instead of deleting them. Here we found a more expected progression of times, going from 0.0030 prior to optimizations, 0.0019 after optimizations and 0.0010 after indexing.

Query	DataBase	Prior Optimization (s)	After Optimization (s)	After Indexing (s)
Insert a new review to a listing whose host has been active since 2015	MySQL	0.0073 0.0109	0.0118 0.0124	0.0115 0.0110
Delete the listings of the first 5 hosts with the highest number of 'Entire rental unit' (property_type) available on 2024-12-25	MySQL	0.2195 0.2808	0.1993 0.1441	0.4669 0.2252

Table 1. Comparative query execution time table for MySQL

The following aspects have been highlighted for MongoDB:

- For the first complex query:
  - There was no observable improvement despite changing the database structure and indexing a relevant field.
  - The structure change is irrelevant since listings are found by the `host_since` field, which is already part of the listings.
  - The index on `host_since` was expected to improve query performance, however, it can be observed that when the number of qualifying listings is high, the index doesn't have much value.
- For the second complex query:
  - We observed a significant decrease in execution time after optimization. Having embedded `calendar` data eliminated the need to perform a join between collections, removing the expensive *`$lookup`* and *`$unwind`* operations.
  - On the other hand, the indexing on *`host_id`*, *`property_type`*, and *`calendar.date`* did not affect performance. In this case, rather than relying on indexing, embedding data was the main factor in performance improvement. This was because the previous bottleneck (separate `calendar` collection) was removed, overshadowing the impact of indexing.

Query	DataBase	Prior optimization (s)	After Optimization (s)	After Indexing (s)
Insert a new review to a listing whose host has been active since 2015	MongoDB	0.0015 0.0030	0.0021 0.0236	0.096 0.0084
Delete the listings of the first 5 hosts with the highest number of 'Entire rental unit' (property_type) available on 2024-12-25	MongoDB	4.2533 8.7449	0.0204 0.0586	0.0345 0.1090

Table 2. Comparative query execution time table for MongoDB

3. DISCUSSION

a. OPTIMIZATION AND INDEXING

It's worth emphasising that before we ran our indexing performance tests, we implemented a crucial dataset reset mechanism. This was particularly necessary due to the destructive nature of our second complex query, which deletes listings from both databases.

The advantages and disadvantages of making changes to our databases and indexing for each complex query were discussed in the previous section.

b. ADDITIONAL FEATURES

We did not implement any additional features beyond what was required by the project specification and what we learnt from the practical class guides.

c. UNIMPLEMENTED ASPECTS

As for the unimplemented aspects, we found nothing to change in order to optimize the simple queries for both databases developed in Phase 1 of the project, however, we did manage to successfully implement everything defined in the requirements of the Phase 2.

d. KNOWN ERRORS

Initially we found some challenges in understanding what kind of changes we could make to the structure of both databases to optimize the performance of the complex operations defined in Phase 1 of the project.

Other challenges included understanding whether the indexes we created would genuinely enhance data retrieval speeds, or if over-indexing would lead to increased storage overhead. This uncertainty required us to conduct further testing and analysis to evaluate the trade-offs associated with our indexing strategies.

4. CONCLUSION

In conclusion, the Phase 2 our project offered a comprehensive understanding of the performance of different queries before and after optimisation, the use of indexing in databases to observe the efficiency while searching, the use of normalisation to organise data in a database, and changes made in the goals of the Phase 1, such as changes introduced to the relational schema to improve performance and alterations to the data model in NoSQL to improve performance.

Furthermore, additional features, unimplemented aspects and known errors identified have also been shared.

REFERENCES

1. Cortez, R. R. (n.d.). New York Airbnb Open Data [Dataset]. Kaggle. Retrieved November 26, 2024, from https://www.kaggle.com/datasets/rhonarosecortez/new-york-airbnb-open-data/data
2. Stack Overflow. (2018, March 22). *How indexes are an overhead on insert, update, delete of a table?* Retrieved from https://stackoverflow.com/questions/52094534/how-indexes-are-an-overhead-on-insert-update-delete-of-a-table