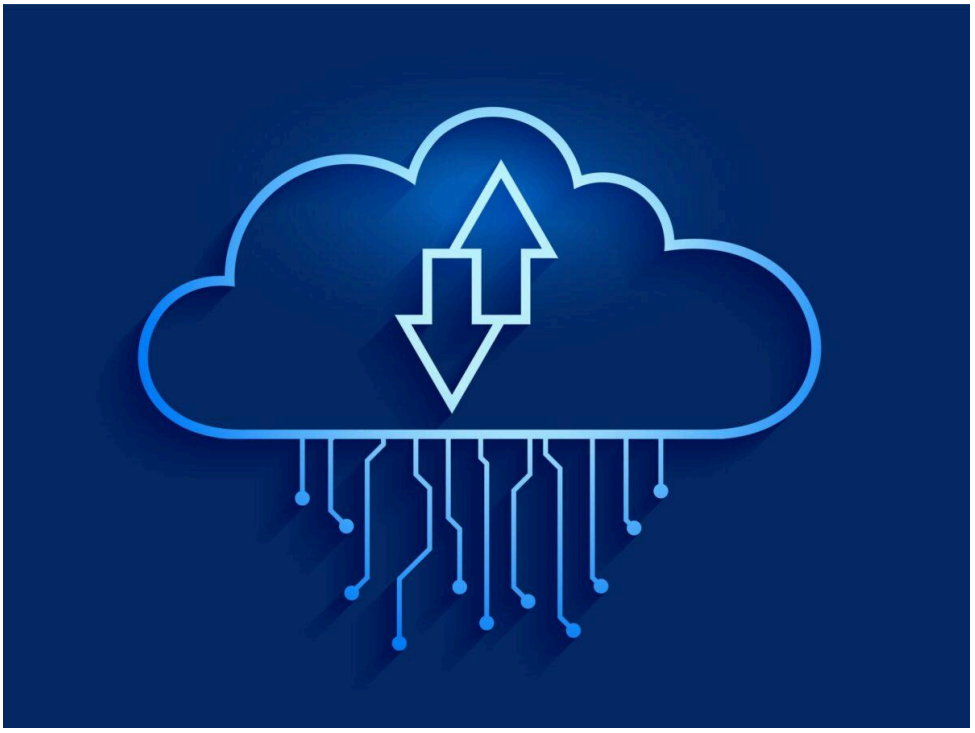
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Relatório P1 SCC

2024/2025



Group members:

Guilherme Antunes nº 70231
Guilherme Carvalhão nº 70735

Implementation Choices:

Application Specifications:

In our application we have three variables in the TukanoRestServer class, one is the appName, as the name suggests this variable is for the base url for the application. We also have two boolean variables called cacheOn and sqlOn, these variables are used to define whether or not the application should use redis cache and sql database.

Likes and Followers:

For the likes and followers we created new containers in our CosmosDB because it was on one hand simpler, and on the other more efficient. For example, if we wanted to add a follower to a user, we would, additionally, need to update the receiving user, which implies reading the user either from the cache or the database. This is undesirable because if a user has one million followers, then we need to read from the cache or from the database a user containing a list with one million usernames, and then update the cache and the database.. Instead it's better to verify if a much simpler follow object exists and possibly add it.

The same thing applies to likes and shorts.

For PostgreSQL we have separate tables for likes and follows.

Blobs:

For the creation of the blobs we had to hardcode the server URI so that the blobUrl generated while creating a short would work.
We also changed thecopyWithLikes_And_Token method in the Short class to get the token with the shortId instead of the blobUrl, we made this change because we were already creating tokens using the shortId in the AzureFileSystemStorage class.

Cosmos DB:

For our CosmosDB implementation we created a single class and we define the container name with an argument when we create the necessary CosmosDB objects, one for each container.

Application Tests:

We started by testing our application with postman, to verify if all the operations worked properly, and also because it was easier to debug one request at a time. After we verified that everything was working as intended, we proceeded to test our application with artillery.

For the artillery tests we used the ones provided by professor Kevin.
First we tested our application with different combination of CosmosDB no SQL, PostreSQL, and Redis Cache, the latency results we got were as follows: (note that the values are in milliseconds)

Min_time/ms	GetFeed	GetShorts	Follow	Like
No SQL/Redis	143	152	139	265
No SQL/No Redis	119	114	193	193
SQL/Redis	68	75	69	65
SQL/No Redis	73	84	69	67

Max_time/ms	GetFeed	GetShorts	Follow	Like
No SQL/Redis	615	542	303	265
No SQL/No Redis	793	159	213	193
SQL/Redis	110	266	106	77
SQL/No Redis	109	98	69	104

Mean_time/ms	GetFeed	GetShorts	Follow	Like
No SQL/Redis	286.5	282.8	222.7	265
No SQL/No Redis	237.5	134.2	203	193
SQL/Redis	87.9	170.5	83.5	71.5
SQL/No Redis	89	91	69	82.8

The GetShorts operations involve queries, which is why it performed worse while using the cache. This is expected because queries are usually different, meaning that it is difficult for them to be present in the cache, and this implies a request to the database. The scenarios with no cache were faster because the application went straight for the database instead of verifying the cache.
The mean response time for the GetFeed operations also shows a similar pattern, which is also to be expected because it incorporates the GetShorts operation.

For the Like operation we can also see that for CosmosDB with cache the operation takes longer, this is because when the like is created it's added to the cache and the database, while with no cache it's only added to the database, meaning we cut a part of the operation.

We can also observe that the response time for PostgreSQL is around three times less for GetFeed, close to half for GetShorts, and again around three times less for likes and follows. This degree of consistency throughout our tests clearly shows that PostgreSQL is much faster than CosmosDB with No SQL.
CosmosDB NoSQL focuses on flexibility, which makes it more complex and less optimized when compared to PostgreSQL, which uses a relational model with structured schemas, making it easier to optimize.

Conclusion

After doing this project we reached the conclusion that in most cases the best solution out of the four we developed is SQL without cache for operations that involve queries, and with cache for every other operation that can be repeated multiple times. This is due to the fact that queries are frequent in these types of applications and, as we explained before, cache doesn't perform well when there are a lot of queries involved.