Query performance analysis of document databases in e-commerce

Fernando Heredero López

Abstract

The following document databases have been evaluated to measure the performance of the queries: CouchDB, MongoDB, DynamoDB, CosmosDB, and Firestore.

E-commerce is growing on a large scale, and the purpose of this paper is to provide an analysis of query performance in this field. The paper is structured by the steps of state of art where each database is defined, and their features have been detailed. Then, the installation process of each database has been explained in detail, from a user's point of view. In the implementation section, it is presented the data used in the experiments. Also, the loading of the data in each database is included in this section. Then, the results of the simulation and experiments have been exposed. Finally, the benchmarking and the conclusion sections end the paper.

The objective of this paper is to show which database is faster in query execution. Also, this paper explains the difference in time execution of querying indexing and no indexing. As a result, queries with different complexities have been done.

Regarding the next steps and future directions. It is intended to assess these experiments and their results with other NoSQL databases. It would be very interesting to implement these experiments in databases such as ArangoDB, MarkLogic, eXist DB, and other document databases.

1. Introduction

In e-commerce, it is necessary that a database meets the following features. First, high performance is necessary to perform a good shopping experience for customers. For example, database queries should execute faster, and live customer interactions are requested. Second, high availability and scalability are necessary. A database should bear sudden traffic spikes and for that, it should be scalable when is necessary. As a result, this paper would be demonstrated why document databases are a better solution than relational databases.

Normally, relational databases are used in e-commerce services. These databases are built using the standard query language (SQL) and the data is stored in tables with columns and rows. Nevertheless, this paper focuses on the use of document databases in e-commerce. Document databases are non-relational databases. Are denominated as NoSQL and is a nontabular database with a flexible schema that works well with unstructured data.

There are three main advantages of using NoSQL rather than SQL. First, the flexible data models allow the inclusion of a variety of data, for example, a picture of the product. Second, the scaling in NoSQL is horizontal, for example, a company can add cheaper servers when they need them. Third, the queries are faster in NoSQL because complex joins are avoided.

Introduced the advantages of using the NoSQL database in e-commerce. The topic is to provide an assessment of which of the NoSQL databases is faster in query performance.

2. State of art

To start with, a closer definition of the main characteristics, advantages, and disadvantages of each document database will be presented.

CouchDB is an open-source document-oriented database that is designed for local replication and horizontal scaling. CouchDB manages a collection of JSON documents, these documents are schema-less. Regarding queries, they are done through the views, which are defined with aggregate functions and filters that are computed in parallel.

CouchDB provides ACID semantics (Atomicity, Consistency, Isolation, Durability). These properties are intended to guarantee data validity despite errors, power failures, and other mishaps. CouchDB treats all stored items as a resource, these items have a unique URI that gets exposed via HTTP. REST uses the HTTP methods POST, GET, PUT and DELETE for the four basic CRUD (Create, Read, Update, Delete) operations on all resources. HTTP is a widely understood, interoperable, scalable, and proven technology.

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling. The structure of the data stored in MongoDB is composed of field and value pairs. In MongoDB, exists embedded data models that reduce I/O activity on database systems. Also, indexes support faster queries and can include keys from embedded documents and arrays. Embedded documents are used to reduce the need to use complex joins that are expensive.

Important features to detail with respect to MongoDB. First, the high performance in data persistence, second, the rich query language supported. Third, high availability because exists a group of MongoDB servers that maintain the same dataset, providing redundancy and increasing data availability. Therefore, replication is possible. Fourth, MongoDB allows horizontal scalability. Besides, MongoDB has a deep query ability, enabling fast access to data.

MongoDB provides lower execution times than MySQL in the four basic operations, which are: Insert, Select, Update and Delete. So, this is essential when an application should provide support to thousands of users simultaneously, this way, it can be performed.

It is important to mention the problems with MongoDB.

First, there are problems with reliability because the updates are not confirmed. As a result, the updates make faster but less reliable. Second, initially, the schema-less design is a great advantage, because brings flexibility. Nevertheless, in MongoDB, the data is stored in nested JSON objects called documents. Then, the data forces a design of a schema in the app logic instead of the database. Thus, the rules and regulations of the data models are dictated by the app logic rather than the database itself.

DynamoDB is a NoSQL cloud database service that provides consistent performance at any scale. Hundreds of thousands of customers rely on DynamoDB for its fundamental properties: consistent performance, availability, durability, and a fully managed serverless experience. The goal of the design of DynamoDB is to complete all requests with low single-digit millisecond latencies.

DynamoDB achieves boundless scale for tables. Therefore, there aren't predefined boundaries for data that a table can store.

DynamoDB is ideal for applications with known access patterns. There are three core concepts, the tables are a collection of items, the items are collections of key/value pairs or attributes and exists a primary key that is a partition key and sort key.

In DynamoDB, there are use cases in retail and e-commerce. It can be appreciated in this table with the corresponding design pattern.

Use case	Design pattern
Shopping cart	1:1 modeling, 1:M modeling
Workflow engines	1:M modeling
Inventory tracking and fulfillment	1:M, N:M modeling
Customer profiles, accounts	1:1 modeling
Session state	1:1 modeling

Table 1: Use cases of DynamoDB in retail and e-commerce

In a NoSQL database, the data is shared in one table. So, it is avoided the multiple requests to many tables. Besides, a partition key is included which makes it even easier.

Azure CosmosDB is a fully managed database service with turnkey global distribution and transparent multi-master replication. You can run globally distributed, low latency operational and analytics workloads and Artificial Intelligence on transactional data within your database. In CosmosDB the cost of database operations is measured as Request Units (RUs).

The data is stored in documents that are in JSON format. The following key features are:

- Turnkey Global Distribution
- Regional presence
- Very high availability
- Elastic scale
- Low latency guarantee
- No schema

Besides, CosmosDB has multiple API, like MongoDB, Cassandra, SQL, Gremlin, and Azure Table Storage.

Azure Cosmos DB has three main functionalities. First, when a document is inserted or modified, this triggers a call to an API. This is the event-computing and notification that includes Azure Functions, Azure Notification Hubs, and Azure App Service. Second, there is a real-time processing (stream) of data, which includes Azure Stream Analytics, Apache Spark, and Apache Storm. Third, there are zero-downtime migrations, which include services such as Azure Data Lake and Azure Storage Table. The wide functionalities that offer Azure CosmosDB show that it is the most suitable in the IoT field.

Cloud Firestore is an integral part of the Google Firebase platform. It takes the form of a cloud-based NoSQL database server that is excellent for storing and syncing data. Besides, it is a high-performance database that supports automatic scaling. Also, it is very reliable and easy to use.

2. Installation

The installation of each type of database has been done. CouchDB and MongoDB are open-source document databases, and their installation is done locally. In the annex, are included some photos of the different interfaces.

Couch DB: Once CouchDB is installed, the direction to connect to *fauxton* interface is *http://localhost::5984/_utils*. The next step is login, and the user can see all the databases created and their size. Documents are created into a database in a JSON format. Besides, *cURL* is used to communicate with the CouchDB database. *cURL* provides easy access to the HTTP protocol directly from the command line and is therefore an ideal way of interacting with CouchDB over the HTTP REST API. Users can request information from a database, create a database, and import data to a database.

In the annex, photos of the interface of CouchDB are attached. In the image, one is attached a screenshot of the databases created in CouchDB.

MongoDB: The installation of MongoDB is easy and fast. The user can use MongoDB Compass which is the interface and the MongoDB shell (Mongosh). In Mongo Compass is very easy to create a database, create documents, and introduce data directly from JSON or CSV. Regarding importing data, MongoDB is much better than CouchDB.

DynamoDB: There is no installation involved in DynamoDB because is a cloud database service provided by Amazon Web Service. The user selects DynamoDB in the console management of AWS and can create tables.

CosmosDB: There is no installation involved in CosmosDB because is a multi-model database service provided by Microsoft. The process is to create an Azure Cosmos account and create a container to create a new database.

Firestore: There is no installation involved in Firestore because is a cloud NoSQL document database service provided by Google. The process starts with creating a Google Cloud account and choosing mode native. This mode includes a database of documents organized in collections and documents. Regarding the queries and transactions, the queries are solid and coherent in overall the database.

3. Implementation

Created the databases, the next step is the importation of data in each database. The data selected is an e-commerce dataset extracted from Kaggle. This dataset includes actual transactions from a United Kingdom retailer. [10]

The dataset is a CSV of size 45.58 MB. This file contains information about one hundred items sold. Not all the dataset has been imported, slices of the dataset has been imported in the different databases.

The attributes of the data are:

- **InvoceNo**: Invoice number. Nominal, a 6-digit integral number uniquely assigned to each transaction.
- **StockCode:** Product (item) code. Nominal, a 5-digit integral number uniquely assigned to each distinct product.
- **Description:** Product (item) name. Nominal.
- **Quantity:** The quantities of each product (item) per transaction. Numeric.
- **InvoiceDate**: Invoice Date and time. Numeric, the day and time when each transaction was generated.
- **UnitPrice:** Unit price. Numeric, Product price per unit in sterling.
- **CustomerID:** Customer number. Nominal, a 5-digit integral number uniquely assigned to each customer.
- **Country:** Country name. Nominal, the name of the country where each customer resides.

About the queries, the queries have been done with different complexities. Therefore, the aim is to measure the performance of the queries in each database.

CouchDB: The importation of the data in the database created is done in JSON format. A CSV of products has been imported into the database. Each row of the CSV which is a product is a document in the database with a unique identifier.

The queries in CouchDB are done with Mango Query in *Fauxton* interface. In Mango Query, there are two parts: index and selector. The index specifies which fields we want to be able to query on, and the selector includes the actual query parameters that define what we are looking for exactly.

The format of the queries is in Javascript format, for example:

```
{
  "selector": {
    "Quantity": {
      "$eq": 6
    }
  }
}
```

This query returns all the products that have been sold six times. Several queries have been done, evaluating their responses and their execution time.

In Mango Query, the execution statistics are provided in each query done. The information provided is the documents examined, the results returned, and the execution time.

The database "product" created contains different documents, each document is an item sold. There is an example attached in annex, image five. It was difficult to upload data into this database. One of these difficulties was that the mandatory format to upload the data is JSON format. Therefore, is a problem for enterprises in e-commerce which deals mainly with excel files. One solution is to convert the CSV file to a JSON format.

Nevertheless, this solution requires a strong converter able to handle large data files and able to transform them.

Uploading documents in CouchDB is a costly task because each document is assigned a unique id. This fact is useful in small datasets but is excessive for large datasets. For this reason, it was uploaded the information of forty items sold. The first forty rows of the CSV mentioned earlier.

Regarding the queries done in CouchDB, the results are quite good. The possibilities are quite like SQL queries. Mango Query offers queries with combination of operators, for example, equal, and, or. Although, the JavaScript format is somewhat uncomfortable for the user. Is necessary to be careful with parentheses, brackets, commas, etc.

MongoDB: The size of the dataset imported is 5 MB. The collection is composed of one hundred thousand documents, each document, a product sold. The database created "product" and the collection "products" which contains the data can be seen in the annex, image 6.

The uploading of the data in a MongoDB database is very easy for the user. The user creates a collection in the database, and the user can upload a csv or a JSON to the collection. In case of CSV, the data is stored file by file and the "products" in this case are being stored efficiently. In this matter, MongoDB is much better than CouchDB database. As a result, the dataset loaded in this collection is composed of 100 thousand products sold.

Regarding the queries, there is an option in the command, which provides the user execution statistics of the queries. The command is *explain.executionsStats*, which provides information as the estimated amount of time in milliseconds for query execution. Also, the execution stages, the number of docs examined, keys examined can be consulted. The information provided regarding executionststaistics is wide.

MongoDB uses indexes to support the efficient execution of queries. Without indexes, MongoDB must perform a collection scan. This means that the complexity of the queries would be O(n). Therefore, each document would be scanned in all queries. Is necessary to create an index before the query to get the query more efficiently. Therefore, it has been done queries without indexing and with indexing. In the annex, image seven, is attached a screenshot of an index created in the collection "products". This index is created on the attribute "Quantity".

DynamoDB:

There is no implementation in this database. The reason is the pricing of the database, DynamoDB charges for write capacity units and reads capacity units. For this reason, this database has been not considered. Also, there are no free trials for small data storages.

CosmosDB:

In the CosmosDB interface, a database named "e-commerce" was created, inside the database, user can create containers. In containers, the data can be uploaded in JSON format. The dataset uploaded in the container named "Container_1" is formed by one hundred products sold. It is the same dataset mentioned earlier, with the difference that in this case, the number of products is 100 items. The reason why the data must be short is because, a load of a higher dataset entails a big cost on the user billing statement. In the annex, image ten, it is attached a screenshot with the database and the container created.

Regarding queries, the format of the queries is SQL. When a query is done by the user, the results and the execution statistics are provided in the interface of CosmosDB. The following metrics are provided:

- Request charge (RUs)
- Showing results
- Retrieve document count
- Retrieve document size (Bytes)
- Output document count
- Output document size (Bytes)
- Index hit document count
- Index lookup time (ms)
- Document load time (ms)
- Query execution time (ms)
- System function execution time (ms)
- User defined function executed (ms)
- Document write time (ms)

These metrics can be downloaded by the user in CSV format. This is an advantage compared with other databases. This is the unique database that provides this possibility.

Firestore:

There is no implementation in this database. The reason is the pricing of the database, DynamoDB charges for write capacity units and reads capacity units. For this reason, this database has been not considered. Also, there are no free trials for small data storages.

4. Experiment

In this section, an outline of the experiment has been illustrated with a schema to each database.

The data is loaded in the databases created and the queries are done providing the execution statistics.

The architecture of Couch DB is shown in the next image.

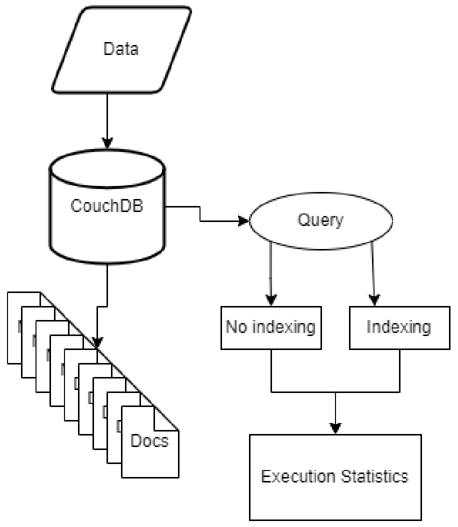


Image 1: CouchDB architecture

The architecture of Couch DB is shown in the next image.

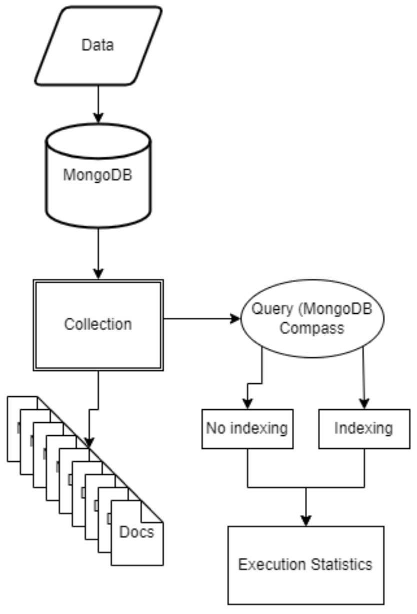


Image 2: MongoDB architecture

The architecture of CosmosDB is shown in the next image.

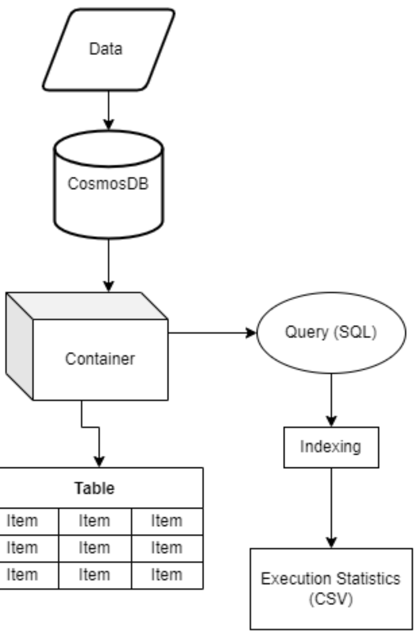


Image 3: CosmosDB architecture

This information about the architecture is relevant to follow adequately the information of the paper. Each database has his own manner to store the data in the databases. Therefore, in the next section, the results of the experiment and the process of the experiment is explained in more detail.

5. Results

CouchDB: The size of the database created was 1.6 MB, and the number of documents was forty. The reason why these values were short is because, as has been said before, each document needs to be uploaded one by one. Therefore, it was not worth it to upload a big number of documents to the database.

Six kinds of queries have been done in this database. The results provided are execution time, documents examined, and results returned.

The table below show the execution results of the queries

Query	Execution time (ms)	Documents examined	Results returned
1	12	40	38
2	6	40	2
3	5	40	1
4	16	40	39
5	176	40	39
6	29	40	5
7	18	40	11

Table 2: Execution statistics CouchDB without indexing

The queries are:

Q1: {"selector": { "Quantity": {"$eq": 6}}}
Q2: {"selector": { "Quantity": {"$eq": 2}}}
Q3: {"selector": { "Quantity": {"$gt": 35}}}
Q4: {"selector": { "Quantity": {"$lt": 35}}}
Q5: {"selector": { "Quantity": {"$lt": 35}},"fields":"Invoce_No","Country"]}
Q6: {"selector": { "Quantity": {"$gt": 10}},"fields":"Invoce_No","Country"],"limit":5}
Q7: {"selector": { "Quantity": {"$gt": 10}},"fields":"Invoce_No","Country"],"limit":30, "skip":4}

The execution time is higher when the query returns a higher value of results. For example, queries two and three return two and one result, and the execution time is very low, 6 miliseconds. Nevertheless, queries that return higher results, for example, one or four, show a higher value of time execution.

Besides, queries six and seven show an increase in time execution, being the same query. But, "limit" parameter was added, and that is the reason why the execution time increased. Looking that the execution time depends mainly on the results returned. It has been collected these values and the following plot shows the results.

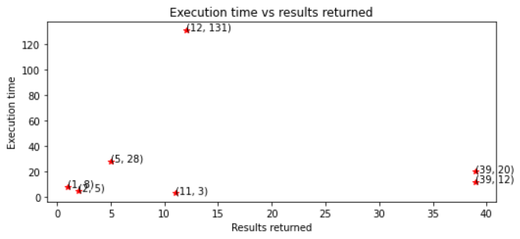
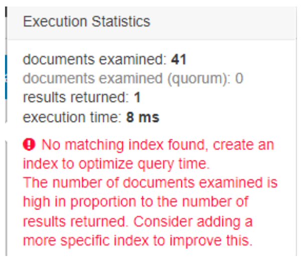


Image 4: CouchDB execution time vs results returned

In the plot, y-axis corresponds to execution time and x-axis corresponds to number of results returned. The results presented correspond to queries with complexity O(n). It can be seen in the table on the column "Documents examined". In all queries, all the documents are being visited.

When a query is done in CouchDB, a warning alert appears advising to use indexing. The following alert message appears.



Execution Statistics

documents examined: **41**
documents examined (quorum): 0
results returned: **1**
execution time: **8 ms**

❗ No matching index found, create an index to optimize query time.
The number of documents examined is high in proportion to the number of results returned. Consider adding a more specific index to improve this.

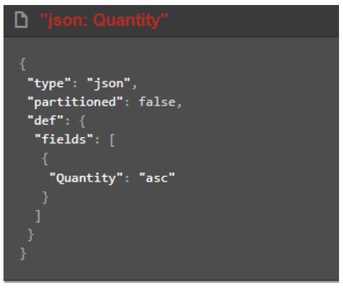
Image 5: Warning message

This alert message appears when the results are much lower than the documents examined.

In this case, it is very useful to create an index in this attribute. So, the query complexity would be O(1) instead of O(n).

Then, in CouchDB the user can create an index, the format is the same as the queries, JavaScript. So, an index on "Quantity" feature was created and the same queries have been done.

The index created is:



```
"json: Quantity"

{
 "type": "json",
 "partitioned": false,
 "def": {
  "fields": [
   {
    "Quantity": "asc"
   }
  ]
 }
}
```

Image 6: Index creation

In this table is presented the results collected:

Query	Execution time (ms)	Documents examined	Results returned
1	6	12	12
2	5	2	2
3	1	1	1
4	12	39	39
5	12	39	39
6	4	5	5
7	3	15	11

Table 3: Execution statistics CouchDB with indexing

In this case, the results returned are the same as the document examined. This is because the complexity of the queries is O(1). Viewing the results, the execution time is much lower than doing the queries without indexing. The reason is that queries don't need to go through all the documents. This effort is avoided by the query system, so, the execution time decreases.

This graph shows the difference to use indexing and not using it. Red points are the execution statistics of queries done without indexing and the blue points are the execution statistics of queries done with indexing.

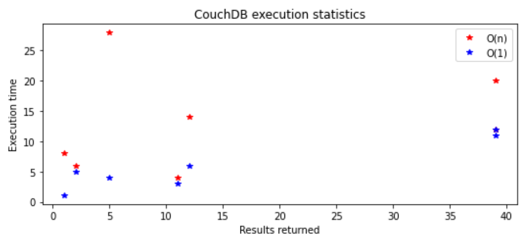


Image 5: CouchDB indexing and without indexing results

It can be appreciated that being the same queries and returning the same results, the indexing performs the queries faster.

MongoDB: The size of the collection is 5 MB, and the number of documents is formed by 100 thousand items sold. The queries done in this collection are the same as those used in CouchDB. Nevertheless, the results collected seek to explain the fact of advantages using indexing in MongoDB. The queries done were shown earlier, and are labeled as Q1, Q2, Q3, and Q4.

These queries have been done in two different collections. The data of the collections are the same. Nevertheless, one has an index assigned to the attribute "Quantity" and the second doesn't have indexing. Therefore, the queries filter values depending on the "Quantity" attribute.

In this table is presented the results collected.

Query	Execution time (ms)	Documents examined	Keys examined	Results returned
1 (no index)	70	100000	0	7407
1 (indexing)	39	7407	7407	7407
2 (no index)	94	100000	0	15295
2 (indexing)	25	15295	15295	15295
3 (no index)	76	100000	0	22475
3 (indexing)	44	22475	22475	22475
4 (no index)	72	100000	0	75971
4 (indexing)	137	75971	75971	75971

Table 4: Execution statistics MongoDB

It can be appreciated that the execution time is lower in the case of the queries in the collection with indexing. The reason is that, with indexing, the query complexity is O(1) instead of O(n).

Therefore, the time execution is lower because the number of documents examined in the query is the same as the number of results returned. Then, the query doesn't need to go through all the documents. In the table, can be seen that the queries in collection without indexing need to go through 100 thousand documents.

Nevertheless, in some cases, creating an index in attributes is not an improvement in execution time. For example, Q4 returns all the products which are sold less than 34 times. The query execution time is less in case of collection without indexing, 72 milliseconds (No indexing) and 137 milliseconds (Indexing). This happens because the number of results returned is high with respect to the total collection. In this case, the number of results returned is 75971 from a collection of 100000. Then, it is not worth it to create an index. In conclusion, indexing is a good practice when the results are much less than the overall collection.

This plot shows the execution time of the queries with indexing (blue points) and without indexing (red points)

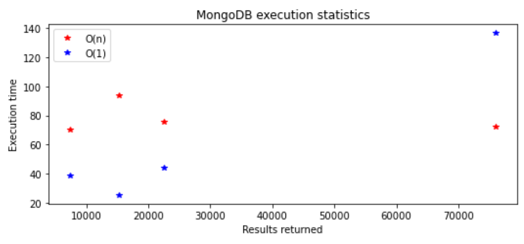


Image 6: MongoDB indexing and without indexing results

It can be appreciated, how to decrease the execution time in case of queries with indexing. Nevertheless, the point on the right side of the plot shows a higher value of execution time without indexing. This case is explained earlier, in some cases, it is not worth it to create an index because the number of returned documents is closer to the total documents in the database.

<u>CosmosDB:</u>

In CosmosDB, the user creates a database in CosmosDB. Into it, several containers can be created, these containers store data. The data loaded in the container is one hundred of products sold. Several queries have been done and the execution statistics have been collected. In order to maintain the same queries to all databases. The queries Q1, Q2, Q3, Q4 and Q5 have been done. Also, another query was done which sorted the values in ascendent order with respect to the unit price of the product sold. This query is:

Q6:
```
SELECT * FROM Container_1 c
 WHERE c.Quantity > 10
 ORDER BY c.UnitPrice ASC
```

Regarding to indexing, in CosmosDB the queries are done in SQL. Then, the index is created when a condition is defined. The queries, which include conditions are done with indexing.

The following table collects the execution statistics results obtained from the queries. The results shown are "document write time" which is the time spent to write query result set to response buffer. Second, the "query engine execution time", which is the time spent by the query engine to execute the query expression. Third, document load time and the showing results. The following table shows these parameters, being the first row the Q1, the second row the Q2,...etc.

Doc write time (ms)	Query exe time (ms)	Doc load time (ms)	Showing results
0.01	0.04	0.14	43
0	0.02	0.04	6
0	0.05	0.04	6
0.03	0.06	0.27	92
0.02	0.09	0.28	92
0.05	0.12	0.13	28

Table 5: Execution statistics CosmosDB

Regarding query execution time, results are quite similar, because the effort is alike. Nevertheless, the last query is the query that sorts the results with respect to a condition. Then, an increase in the query execution time has been in this case. This is because the effort of the query system is higher.

It has been to try to connect to the database in CosmosDB through SQL API client with the python library. Nevertheless, it is necessary to connect to a server and this involves a cost in the billing statement.

6. Simulation

To show a benchmarking on the document databases. In order to make a comparative analysis between the databases. The same queries have done on the different document databases (CouchDB, MongoDB, and CosmosDB) with the same size of data loaded. And the execution statistics have been collected.

The same dataset of products has been done, seeing the difficulty of uploading documents in the database CouchDB. The data imported are forty products sold of the same dataset from a UK retailer.

The queries done in CouchDB are done with indexing in "Quantity" ordering by "ASC". The same index is created in MongoDB. Then, these results correspond to the same queries in the same collection.

Making this experiment it has been realized that the results from MongoDB and CosmosDB are much better compared with CouchDB in terms of time execution.

This graph shows the execution time by CouchDB (red points) and MongoDB (blue points). The execution times showed in the interface in MongoDB were 0 ms. This means that the execution time in MongoDB was very close to zero, so much, so that this value is approximately zero.

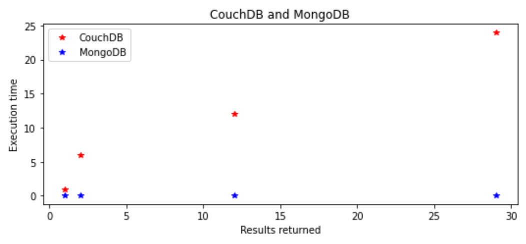


Image 7: CouchDB and MongoDB execution statistics

In CosmosDB is very similar, because the values collected are close to zero. Nevertheless, in CosmosDB the values collected are split in Index lookup time, document load time, query engine execution time.

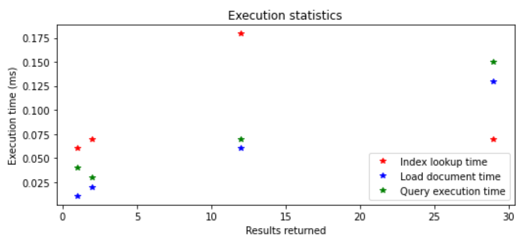


Image 8: Execution statistics

In this plot, red points are index lookup time, blue points are load document time and green points corresponds to query execution time. These values are very low, it can be seen in the y-axis scale that is measured in ms.

Viewing that MongoDB and CosmosDB have similar values of execution statistics on queries done in a small collection. Another experiment has been done with a greater amount of data stored in a collection. The data loaded is formed by 50000 products sold extracted from the dataset of UK retailer.

The queries executed are the same referenced earlier as Q1, Q2, Q3, Q4 and Q5. The queries done are with indexing, then, the complexity of the queries is O(1). This table shows the results collected.

Query	Execution time (MongoDB)	Execution time (CosmosDB)
1	89	476
2	37	358
3	36	321
4	62	620

Table 6: Execution statistics MongoDB and CosmosDB

Other queries with different operators (OR, AND, between) have been done, and the fact is MongoDB executes the results faster than CosmosDB. Besides, a problem that has been encountered in CosmosDB is the following.

When a query is done, the user can see the first one hundred results returned. For example, the first query return 7749 values that comply with the condition. So, in CosmosDB, the user sees the first one hundred values returned and their corresponding execution statistics. The user can load the following one hundred values and so on. But the problem is that the following execution statistics are from generating the following one hundred values. Then, in MongoDB, the execution statistics of all return values are provided. Images are attached in the annex showing this fact in CosmosDB.

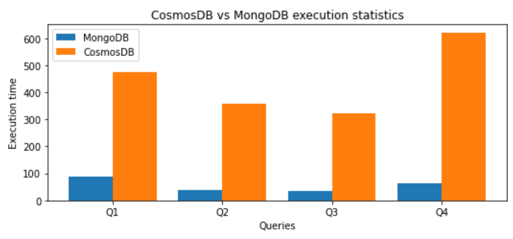


Image 9: CosmosDB and MongoDB execution statistics

We see that the execution time required in CosmosDB are higher than MongoDB.

5. Benchmark

The comparative analysis focuses on the query performance, looking at the execution time and other parameters.

CouchDB is the document database that makes the queries slower compared to MongoDB and CosmosDB. It can be seen on the results exposed in image 6. In this matter, the performance of the queries in CouchDB is far from MongoDB and CosmosDB. This conclusion has been taken viewing the results of the first simulation.

In the second simulation, the results show that MongoDB offers the best query performance in terms of time execution. This fact can be seen in the table 6 and in the image 8.

7. Conclusion

Regarding to success results, it was possible to show the execution differences between queries with complexity O(n) and O(1). Also, it was cleared in which cases use indexing is suitable or not. Besides, the comparative between MongoDB, CouchDB and CosmosDB has been evaluated in terms of execution statistics. The objective to find the best databases in query performance between the databases evaluated has been achieved.

Regarding to failure results, the problem encountered in CosmosDB about how the execution statistics are calculated was unexpected failure. This fact provoked that the results of the table 6 in case of CosmosDB are approximations. Besides, Firestore and DynamoDB are not used in the experiment and simulation because the reasons exposed. Nevertheless, the comparative analysis would be better including these databases.

8. Reference

[1] Pandey, M. (2022) 8 most popular NoSQL databases, Analytics India Magazine. Available at: https://analyticsindiamag.com/8-most-popular-nosql-databases/ (Accessed: October 18, 2022).

[2] General Elibrary Search (no date) instructional media + magic. Available at: https://www2.immagic.com/?page_id=40 (Accessed: October 16, 2022).

[3] Yih, J.S.Y. (2021) Query in apache couchdb: Mango Query, DEV Community 👩‍💻👨‍💻. DEV Community 👩‍💻👨‍💻. Available at: https://dev.to/yenyih/query-in-apache-couchdb-mango-query-lfd (Accessed: October 20, 2022).

[4] jlb333333jlb33333336122 silver badges1212 bronze badges (1964) CouchDB Mango Queries (couchdb 2.0.1), Stack Overflow. Available at: https://stackoverflow.com/questions/45976416/couchdb-mango-queries-couchdb-2-0-1 (Accessed: November 7, 2022).

[5] Amazon dynamodb: A scalable, predictably performant, and fully ... - usenix (no date). Available at: https://www.usenix.org/system/files/atc22-elhemali.pdf (Accessed: October 28, 2022).

[6] Microsoft Azure Cosmos DB revealed (no date) Google Books. Google. Available at: https://books.google.com/books?hl=es&amp;lr=&amp;id=DYtHDwAAQBAJ&amp;oi=fnd&amp;pg=PR5&amp;dq=CosmosDB&amp;ots=JRvUote8i4&amp;sig=wNmL2_lY4zTUCyb20r7YpXu-Xf4#v=onepage&amp;q=CosmosDB&amp;f=false (Accessed: November 3, 2022).

[7] CouchDB - installation (no date) Tutorials Point. Available at: https://www.tutorialspoint.com/couchdb/couchdb_installation.htm (Accessed: November 12, 2022).

[8] 1.4. curl: Your command line friend¶ (no date) 1.4. cURL: Your Command Line Friend - Apache CouchDB® 3.2 Documentation. Available at: https://docs.couchdb.org/en/3.2.2-docs/intro/curl.html (Accessed: November 4, 2022).

[9] Azure (no date) Azure/azure-cosmos-python: 🚨🚨🚨this SDK is now maintained at https://github.com/azure/azure-sdk-for-python 🚨🚨🚨, GitHub. Available at: https://github.com/Azure/azure-cosmos-python#create-a-container (Accessed: November 16, 2022).

[10] Amykagg (2019) Starter: E-commerce data 1dc04828-4, Kaggle. Kaggle. Available at: https://www.kaggle.com/code/amykagg/starter-e-commerce-data-1dc04828-4/data (Accessed: October 20, 2022).

[11] Convert CSV to JSON (no date) CSV To JSON Converter. Available at: https://www.convertcsv.com/csv-to-json.htm (Accessed: November 3, 2022).

[12] Lahtela, M. and Kaplan, P.(P. (1966) ES, Amazon. Oberbaumpresse. Available at: https://aws.amazon.com/es/dynamodb/ (Accessed: October 10, 2022).

[13] Firebase pricing (no date) Google. Google. Available at: https://firebase.google.com/pricing/ (Accessed: October 21, 2022).