Comparing Databases for Inserting and Querying Jsons for Big Data

Panche Ribarski, Bojan Ilijoski, and Biljana Tojtovska

Faculty of Computer Sciences and Engineering,
Skopje, Macedonia
panche.ribarski@finki.ukim.mk,bojan.ilijoski@finki.ukim.mk,biljana.tojtovska@finki.ukim.mk

Abstract. This paper tackles the topic of performance in Big Data data ingestion, data querying and data analytics. We test the import of the Last.fm Million Song Dataset in Cassandra, Mongo, PostgreSQL, CockroachDB, Mariadb and Elasticsearch. We also test three types of queries over the JSON documents and present the test results for unique visibility in the direct comparison of the performance of the selected databases. We conclude that by using a combination of state of the art scalable database, for which we recommend Cassandra, and Elasticsearch for search and analytics, we can get a unique tool for efficiently storing and querying data which can schema easily along the way.

Keywords: Big Data · search · analytics · performance · Elasticsearch · Cassandra · MongoDB · PostgreSQL ·$MariaDB$· CockroachDM · JSON.

1 Introduction

The Big Data field and Big Data Processing have attracted a lot of attention in the past years. Many researchers are choosing this field to work in, and more importantly, the industry is grasping Big Data and implementing it in everyday work. There are many Big Data oriented databases, mainly under the NoSQL name[1]. In fact, all the classic databases are also going after the Big Data segment with additions in their products to match features from the newly created databases in the NoSQL field.

By using Big Data, we are now storing vast amount of data, which is getting harder and harder to understand and query. Having clusters of databases, changing schema and inserting lots of data makes it difficult to continuously get knowledge from our data and effectively query our databases.

In live systems, using Big Data means that we may have to (more frequently must) change our data schema along the way. Some systems allow dynamic schema in their collections, but some systems are strict and require data transformation with any schema change. From our experience, it is always better to allow dynamic changes of the schema, without the needed impact of the schema change on the old data in the collection.

[1] NoSQL - acronym for Not Only SQL

This paper tries to connect these three things - Big Data, effective querying, and dynamic schema. First, we test the load performance of JSON documents in the relational databases MariaDB, PostgreSQL and CockroachDB, using their native JSON support. Then, we test the load performance of the same set of JSON documents on MongoDB and Cassandra and additionally on the hybrid database/search tool - Elasticsearch. The second set of tests is for the query performance of the loaded JSON documents. We prepared three queries, involving counting and filtering on fields in embedded JSON documents. Finally, we assess the ability for schema change in the databases. In Big Data, sometimes the quick ingestion times are critical, and sometimes the faster queries are more important. An important feature in some scenarios is the possibility to change schema in the working data tables. This research tries to answer the question which database to use in which scenarios.

1.1 Related work

As Big Data becomes one of the top research fields, the number of papers related to the subject is growing very fast. Usually, the authors make a comparison between two data tables with similar purpose. In this section, we present a work related to our subject of research. The authors in [10] try to improve the Cassandra database performance by introducing the spatial data indexing and retrieval. They use latitude and longitude to create geohash attribute and develop new querying system for optimization of the number of queries. In [9] the authors discuss that Cassandra shows better performance compared to MongoDB. The comparison between these two NoSQL databases is made with several different types of CRUD workloads and it can be concluded that MongoDB shows better performance for small amount of data, but as the data size increases, the performance of MongoDB becomes very poor. Cassandra, however, gets faster.

In [13] SQL and NoSQL databases are compared, represented by PostgreSQL and MongoDB respectively. It is concluded that in general MongoDB shows better performance. However, select operations of PostgreSQL can be improved by indexing. There are also some articles that make comparison between Elasticsearch and other databases with similar characteristics. For example in the articles [11] and [12] the authors made a comparison between CouchDB and Elasticsearch, and they understood that CouchDB performance is better in CRUD, but Elasticsearch is far better in select and search. The article [14] shows the difference in performance between Elasticsearch and Neo4j applied on 100000 test records from Twitter database. Their experiment results show that Neo4j is efficient in both storage and data retrieval, but Elasticsearch is better with increasing workloads because of high scalability.

Our test will expand the number of databases in the performance analysis - we will use three relational databases, two NoSQL databases and one hybrid database/search and analytics tool.

1.2 Paper organization

The first section is the introduction to the topic of the paper and the related work. The second section presents the databases that we will test and gives background on the test JSON documents in the dataset we will use across the databases. The third section

sets the test scenarios and the metrics used for the tests. The results of the executed tests are presented in the forth section, and the final section concludes the topic of the paper and introduces an idea for combined database and query technologies for best performance.

2 Big Data database choices

Big Data poses big challenges to the industry. To implement Big Data means to have the capability of processing, storing and querying huge quantities of data. The classic relational databases have the capability to store such quantities of data, but their setup and maintenance can become tedious at times. The newer NoSQL databases are designed from the ground up for Big Data. They are easier to setup, and scaling up and down instances of the database is almost automatic.

2.1 JSON support in databases

JSON[2] became defacto standard for storing unstructured data in a plain text format. This unstructured data can be used to represent data more complex than the standard one dimensional row in traditional relational databases. For that reason, JSON is the most used format in Big Data databases. Also, for the same reason, the traditional databases are also implementing a way of storing JSON objects, trying to be on par with the new NoSQL technologies. In the following paragraphs we will explain the JSON capabilities of PostgreSQL, CockroachDB, MariaDB, Mongo and Cassandra.

PostgreSQL supports native JSON format as a field type since version 9.2. The definition of this format is as simple as defining the field type as json, for example the `CREATE` statement for a table orders with ID as primary key and info as json string is `"CREATE TABLE orders (ID serial NOT NULL PRIMARY KEY, info json NOT NULL) ;"`. Inserting JSONs is also simple, you just insert the json string as you would insert any other field type in an `INSERT` statement, for example `"INSERT INTO orders (info) VALUES (' "customer": "John Doe", "items": "product": "Beer","qty": 6');"`. The querying of this JSON field can be done in several ways. The regular `"SELECT info FROM orders;"` will return the full JSON strings in the table. By using the `"->"` keyword, for example `"SELECT info -> 'customer' AS customer FROM orders;"` we get all the customers from the JSONs in the table in a JSON format. With the keyword `"->>"`, for example `"SELECT info ->> 'customer' AS customer FROM orders;"` we get all the customers as text. In PostgreSQl, we can use `WHERE` clause on internal fields from the JSON string, for example the query `"SELECT info ->> 'customer' AS customer FROM orders WHERE info -> 'items' ->>'product' = 'Diaper';"` will answer who bought "Diaper". Aggregations are also supported, as well as some more advanced JSON operators [8].

CockroachDB implements the PostgreSQL interface completely. This means that we can use the same drivers and the same queries on CockroachDB as we do with PostgreSQL.

[2] JSON - acronym for JavaScript Object Notation

MariaDB also supports JSON internally, starting from version 10.2.7. Internally, MariaDB uses `LONGTEXT` as data type, but they use the alias JSON for compatibility reasons with MySQL. MariaDB claims that the performance of their `LONGTEXT` data type for supporting JSON is at least as good as the native JSON of MySQL. MariaDB also provides native JSON functions that help with inserting and querying data [3].

MongoDB is a database designed for storing documents from the ground up. For this, it natively supports JSON as a primary data format in their database engine. Here, the notion of table as in traditional databases is called collection. MongoDB collection is the first level of logical organization for JSON strings, which in MongoDB is called documents. Also, by default, MongoDB supports flexible schema - documents in one collection do not necessarily need to conform to the same specification. MongoDB supports relations between documents in the same or different collections. This removes the need for duplication of referenced JSON data, which will directly lead to smaller databases for the same datasets. MongoDB provides drivers for all popular programming languages [7]. Collections can be created in two ways, implicitly by directly adding a document in it with `"db.orders.insert("customer": "John Doe", "items": "product": "Beer", "qty": 6)"`, or explicitly by calling the `createCollection` function `"db.createCollection("orders")"`. The explicit way supports adding additional advanced options to the collection being created [5]. Inserts are done by calling the insertOne or insertMany functions, and queries are done by calling the find function which supports filtering and embedded/nested search [6]. MongoDB supports aggregations on the data natively, providing real-time computed results [4].

Cassandra supports JSON natively since version 2.2 [1]. But, this native support of JSON just extends the table schema, allowing us to insert a JSON conforming to specific table schema. The only advantage of using JSON with Cassandra is that there is no need to parse a JSON in the client in order to insert in Cassandra - the database will do it by matching the keys in a JSON with the table structure. The rest of the process of working, the JSON data is the same as the classic table operations in Cassandra.

2.2 Target scenario

Our target scenario is a very specific use case where we want to store vast amounts of data - hence the Big Data tag, and we want to index and effectively query that data. We will select a dataset represented with JSON documents and we will test the dataset on PostgreSQL, CockroachDB, MariadDB, Mongo and Cassandra. The test will include inserts and queries of the data. Additionally, for the query part, we will index everything in Elasticsearch and measure the queries against the queries done in the databases mentioned above. The results of the tests will give us a deeper comparative knowledge about what database and query engine to choose when using Big Data.

2.3 Test dataset

The dataset [2] we are using for this scenario is last.fm dataset. The data set contains 839,122 json files and has 943,347 matched tracks between MSD and Last.fm [3],

[3] This is the test dataset from last.fm

```
{
  "artist": "Casual",
  "timestamp": "2011-08-02 20:13:25.674526",
  "similars": [
      {"key": "TRABACN128F425B784", "value": 0.871737},
      {"key": "TRIAINV12903CB4943", "value": 0.751301},
    ],
  "tags": [
      {"key": "Bay Area", "value": "100"},
      {"key": "hieroglyiphics", "value": "100"}
     ],
  "track_id": "TRAAAAW128F429D538",
  "title": "I Didn't Mean To"
}
```


Listing 1: JSON file example

505,216 tracks with at least one tag, 584,897 tracks with at least one similar track, 522,366, unique tags, 8,598,630 track - tag pairs, 56,506,688 track - similar track pairs. An example of one file is showed below. There are six different fields in the JSON files. The first one is artist, which represents the artist that performs the song. Then, timestamp which presents the date and time when the file was created (added to database), track_id, the unique identifier of the song in the last.fm database, title is the title of the song, similars contains the list of track_id, value pairs which represent the precomputed similarity measure between this song and other songs, and the last fields is tags, which is also a list of key, value pairs of song - level tags.

3 Test scenarios and metrics

Our test scenario involves the test dataset from last.fm which contains 839122 JSON documents, all together 5 gigabytes of text data. The first step is to import the JSON documents in the databases, which is repeated 5 times to verify consistency. For easy maintenance of the test environment, we use official Docker images for the databases. For each database, we create a fresh Docker container, and import the JSON dataset into the container of the database. It is worth noting that we use default configuration parameters for each database. Before each import, we drop the tables and create them again. After each import, we run the three test queries over the imported data, also 5 times each query for consistency verification. The test was executed on Intel Core i7-8550U with four cores, 32GB RAM and M.2 2280 NVMe storage. The operating system running Docker was Arch Linux with 4.19.36 kernel version. The Docker version was 18.09.5-ce, and the docker-compose version was 1.23.2. The Python scripts were executed with Python 3.7.3 .

4 Results

In this section we will present the results from the testing of the databases. The average load time for all databases is presented at the figure 1. We insert the whole dataset into the databases several times and compute the average load time in seconds. As we can see there are three different clusters if grouped by load time. In the first one, there are Mongo and Cassandra databases that have significantly better time compared to the others. CocroachDB and Elasticsearch are in the second one and they need very long time to load the whole dataset. MariaDB and Postres are in the third one and their time is somewhere in the middle.

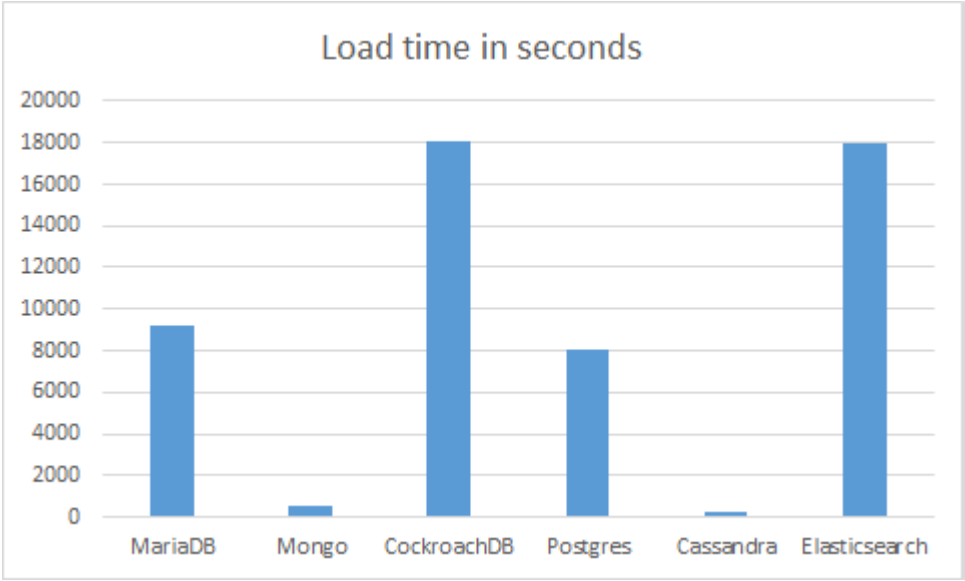

Fig. 1. Load time in seconds

At the Figure 2 the average time of the execution of each query is shown 2. The MariaDB is certainly slowest in performing all of the queries. Postgres keeps an average between 15 and 25 seconds for all of the queries with the best result in the first one and the worst in the second query. Mongo shows pretty good results in the second and especially in the third query, but in the first one is worst then CockroachDB and Postgres. CockroachDB on the other hand, shows good results in the first two queries and very bad result in the last one. It is interesting that Mongo has the worst performance in the first query, Postgres in the second and CockroachDB in the last one. The absolute winner in this measurement is Elasticsearch. It has by far better times in all three queries then the others. If we count an average for executions of the all three queries, then the average time for MariaDB is 121s, Mongo 12s, CockroachDB 27s, Postres 20s and Elascitsearch just 0.35s. As you can see we didn't measure times

on Cassandra database because it is not possible to perform this kind of queries on that database.

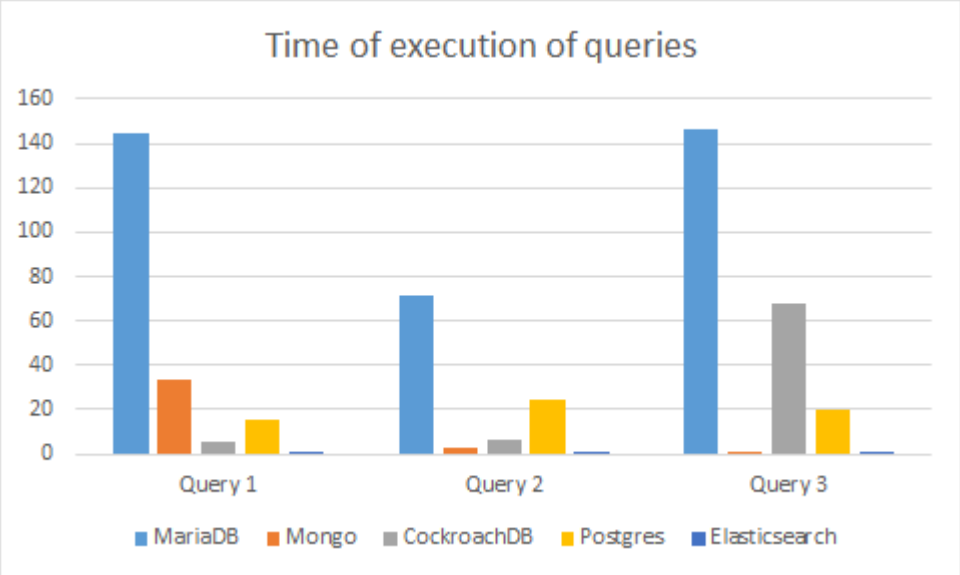

Fig. 2. ime of execution of queries

5 Conclusion

The results show clearly that the NoSQL databases are built for faster manipulation with unstructured data, such as the JSON documents in our tests. The import time of the test dataset for Mongo is 521 seconds and Cassandra is 261 seconds. The import time for Postgres is 8016 seconds, MariaDB is 8441 seconds and CockroachDB is 18023 seconds - these times are 30-70 times slower than the Cassandra import and 15-35 times slower than the Mongo import. In the test with the three queries, Mongo leads with the fastest timings, except for the first query which should probably be optimized in a more Mongo native manner. It is worth noting that Cassandra couldn't execute the queries because it doesn't support native JSON functionality.

Elasticsearch, the hybrid database in which search and analytics are first class citizens, is a special case which should be discussed separately. There are many discussions whether Elasticsearch can be used as a database, be it SQL or NoSQL, (see https://www.elastic.co/blog/found-elasticsearch-as-nosql). The joint opinion between the experts is that Elasticsearch should not be used as a primary database, mostly because of the lack of elegant handling of OutOfMemory exceptions. There is also the lack of transactions and the lack of any authentication and authorization mechanisms,

at least in the open source version of Elasticsearch. However, if we pair this excellent search engine and indexer with another excellent database which supports easy clustering for storing huge data, we would have a Big Data dream team. Such a technology would be Cassandra - a Big Data database store which can manage massive amounts of data, supports fault tolerance by replicating data and is scalable in the number thousands of nodes in one cluster. Apache Cassandra is a proven technology in the industry, used by Apple, Netflix, CERN, Reddit, Instagram and many more big and small companies which leverage Big Data. The best field in which Apache Cassandra shines is easy storage and usage of petabytes of data across a cluster of nodes installed on commodity hardware. Our idea for pairing Cassandra and Elasticsearch means that we would use Cassandra for storing JSON documents in a purely string manner, and at the same time, index the JSON document in Elasticsearch for search and analytics. The string representation of the JSON documents in Cassandra lead to flexible schema - Cssandra would not care if we changed the JSON documents. However, since Elasticsearch needs schema, we would create a new index for each change of the JSON schema, and if all documents are needed in the same index, then reindex the old schema documents with a possible transformation in the pipeline.

References

1. Cassandra json support. https://www.datastax.com/dev/blog/whats-new-in-cassandra-2-2-json-support, accessed: 2019-07-22
2. The last.fm dataset. http://millionsongdataset.com/lastfm/, accessed: 2019-07-22
3. Mariadb json function. https://mariadb.com/kb/en/library/json-functions/, accessed: 2019-07-22
4. Mongo - aggregation. https://docs.mongodb.com/manual/aggregation/, accessed: 2019-07-22
5. Mongo - create database collection. https://docs.mongodb.com/manual/reference/method/db.createCollection/, accessed: 2019-07-22
6. Mongo - crud query documents. https://docs.mongodb.com/manual/tutorial/query-documents/, accessed: 2019-07-22
7. Mongodb drivers and odm. https://docs.mongodb.com/ecosystem/drivers/, accessed: 2019-07-22
8. Postgresql json functions and operators. https://www.postgresql.org/docs/current/functions-json.html, accessed: 2019-07-22
9. Abramova, V., Bernardino, J.: Nosql databases: Mongodb vs cassandra. In: Proceedings of the International C* Conference on Computer Science and Software Engineering. pp. 14–22. C3S2E '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2494444.2494447
10. Ben Brahim, M., Drira, W., Filali, F., Hamdi, N.: Spatial data extension for cassandra nosql database. Journal of Big Data 3(1), 11 (Jun 2016), https://doi.org/10.1186/s40537-016-0045-4
11. Gupta, S., Rani, R.: A comparative study of elasticsearch and couchdb document oriented databases. In: 2016 International Conference on Inventive Computation Technologies (ICICT). vol. 1, pp. 1–4 (Aug 2016)
12. Gupta Sheffi, R.R.: Data transformation and query analysis of elasticsearch and couchdb document oriented databases http://tudr.thapar.edu:8080/jspui/handle/10266/4215

13. Jung, M., Youn, S., Bae, J., Choi, Y.: A study on data input and output performance comparison of mongodb and postgresql in the big data environment. In: 2015 8th International Conference on Database Theory and Application (DTA). pp. 14–17 (Nov 2015)
14. Zhu, J., Tirumala, S.S., Anjan Babu, G.: A technical evaluation of neo4j and elasticsearch for mining twitter data. In: Singh, M., Gupta, P.K., Tyagi, V., Flusser, J., Ören, T. (eds.) Advances in Computing and Data Sciences. pp. 359–369. Springer Singapore, Singapore (2018)