No-SQL term paper - MongoDB*

Anonymous

ABSTRACT

The fast and unprecedented growth rate of technology and cheap cost of various tech devices has created a lot of opportunities for data collection and storage. But now not all the data could be easily stored in the relational database in a strict structured format, there emerged a need for new kinds of the database which started the era of NoSQL databases. There are many new NoSQL database introduced which are significantly different than traditional RDBMS and provides varied functionality. MongoDB is one of the NoSQL database. In MongoDB, the data records are stored in document format and each document structure can be different in the same collection thus providing flexibility. It also provides a wide range of other functionalities. We will be discussing in-depth about MongoDB, its structure, features and where it is beneficial to prefer MongoDB over tradition RDBMS.

ACM Reference format:
Anonymous. 2018. No-SQL term paper - MongoDB. In *Proceedings of NA, NA, 2018,* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 OVERVIEW

In this paper, we will be talking about MongoDB database which is a document based database. This means that each record stored in MongoDB is a document. How do we visualize it? It's similar to a dictionary structure like field and value pairs. Each document is similar to a JSON object [2]. It also allows storing complex structure where you can store the value as an array of elements, other embedded documents and an array of other documents too [2].
Consider the following example:

```
{
   "_id": "abc123",
   "first_name": "John",
   "last_name": "Wick",
   "address": {
      "street_name": "222B Baker Street",
      "zip_code": "98143",
      "country": "United Kingdom"
   }
   "email": "john.wick@gmail.com"
   "phone_number": "598765432"
}
```

Here the *"_id"* is the field which if not provided by the user is autogenerated. *_id* is the default primary key of the collection. MongoDB provides the flexibility for users to provide the id too if they intend to use it for other purposes too apart from indexing. *first_name* indicates the name of the person and *John* is the actual name of the Person. So the left attributes "first_name", "last_name", "address", "email", and "phone_number" are the keys of JSON documents and next to semicolon are all the values for these attributes. It stores documents in BSON format, a binary representation of JSON format. The value of each field is a BSON data type. It supports a wide variety of data types. It also allows for storing embedded documents. All these records are stored in document form in a collection. A database can contain multiple collections. If we want to understand using RDBMS terminology a collection is similar to a table and a document record is similar to a tuple entry in the table in the relational database.

The key features of MongoDB databases are:

(1) High Performance where it's support for embedded documents and data models and indexing support makes queries faster and reduce I/O activity.
(2) Horizontal scalability where they use sharding distribution technique to distribute the data into multiple clusters across different machines.
(3) Rich query language where it supports a huge range of queries which allows performing complex data aggregation using aggregation pipelines, text search by allowing regular expressions, and geospatial queries too which allows us to query data that contains geospatial shapes and points.

Apart from this, the other features include high availability and support for multiple storage engines. [2]

1.0.1 Comparing with SQL. : In general, when we start with any RDBMS application, we first need to design a typical schema which consists of different tables and shows the relation between these tables. This also forces the need to normalize the table to store data in a more structured format which increases the number of joins needed whenever we need something from more than one table, which is the most common case. Following the rules and norms of building a relational schema, we end up building such a schema where we cannot perform basic real-life queries without using at least one join. Even with the presence of indexing strategies, performing joins between two tables containing millions of rows is definitely going to consume some time. MongoDB is designed to be schema-less, means no strict structure necessary, each document can be different from other, it can contain a different number of fields, can be of different size and can also contain different content [2]. It becomes very flexible to store data. MongoDB supports document-based query language which allows querying the database easily without performing complex joins. Also, it is very easy to scale due to flexible format. Apart from that it also provides flexibility of putting an index on any attribute which also

*Produces the permission block, and copyright information

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
NA, NA
© 2018 Copyright held by the owner/author(s). 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

helps with fast in-place updates [2]. There are many more benefits which we will discuss in the respective sections later.

We will be using IMDB movie dataset to illustrate the data management support for MongoDB. This dataset contains information about all the movies like the title of the movie, year of release, list of genres, list of directors who directed it, and lastly the list of actors who acted in this movie. The structure of each document can be something like shown below:

```
{
   "_id" : unique id for movie,
   "title" : name of movie,
   "year" : release year,
   "genres" : [ list of genres],
   "directors" : [ {Document containing
       director information},{},...],
   "actors" : [ {Document containing actor
       information},{},...]
}
```

Our application is built on top of this dataset. It is a simple Movie application where you can search for different movies and details related to those movies. A user can query and search for information about different movies and also add the new movies which are released. Using this dataset a lot of analysis can be performed as in how many movies did two particular actors acted together. What is the maximum number of movies a director and an actor did together? Or can also be used as just a simple movie database. As mentioned above will be a rough structure of a record in movie collection.

The rest of the paper contains the following sections. The section 2 contains database storage and indexing related information of MongoDB like the structure of data stored, different indexing benefits etc. Section 3 cover query processing and optimization of MongoDB and how it is different than RDBMS. Likewise, the section 4 covers transaction management related information of how it also allows updates to be performed in the transaction and provides security features for data. In the section 5, we describe the application we worked on to explore the features of MongoDB. And finally, the section 6 concludes the paper.

2 DATA STORAGE AND INDEXING

2.1 Data Storage

MongoDB supports multiple storage engines which allow storing the data in memory as well as on disks. Each engine provided different types of support and performance depending on the data stored. MongoDB provides flexibility to choose amongst different option based on the data we want to use and performance we want to achieve. The two options which it provides are:

(1) WiredTiger Storage Engine which is default option and works well with all kind of workloads. It provides document-level concurrency, compression, and also checkpointing [2].
(2) In-Memory Storage Engine which retains the data in memory rather than on disks, to avoid data latencies, this feature is only available in the enterprise edition[2].

Apart from that it also maintains a log file named journal which helps with database recovery, providing performance and reliability of database. There is one more feature GridFS which MongoDB uses to store document of large size which exceeds 16MB [2]. The use of it is not straightforward, it has to be used after performing a proper analysis and only if suitable for the application. In general term, MongoDB creates a data directory where it stores all the data in a file with extension ".wt" which is default storage engine option.

MongoDB also provides sharding functionality where data may be distributed among several nodes based on distribution key. This helps in scaling the application without any downtime as nodes can be configured and added even at runtime. Each shard may contain the different number of data records depending on the shards key. There is one primary node assigned that routes the query to appropriate by using the shared key of all the slave nodes. Furthermore, updates performed on a node goes to the primary node and then update is distributed among all nodes in the network.

2.1.1 Comparing with SQL. : The scaling of SQL databases is little troublesome since we would require a separate server to host the database which would serve to be an expensive task. On the other hand MongoDB is very easy scale even by using commodity server and with the aid of *Sharding* scalability is very easy. Furthermore, SQL was designed for structured data which had specific format and defined relationships. MongoDB can store structured as well as unstructured data. In way schema-less MongoDB can encapsulate the structure of SQL.

2.1.2 Comparing with Other NoSQL. : Unlike MongoDB which has *Single Master* mode where updates are written only on the primary node, even when we have multiple shards available, Cassandra, has *Multiple Master* mode where the updates can be distributed on any server [1].

2.2 Indexing

MongoDB also supports indexing of the fields in the documents, which helps to query the database and to retrieve the documents faster. Without indexes, MongoDB has no other option but to scan through all the documents in a collection to find the results.MongoDB stores indexes data in s special data structure that contains a small portion of the dataset. It becomes easy to traverse as they are stored in ordered form and supports equality and range search both efficiently. It also allows users to create an index on all levels. MongoDB also has default index _id which cannot be dropped and is also used by the sharding technique to distribute the data. Apart from that MongoDB allows the creation of user-defined indexes namely single field indexes, compound indexes, multikey indexes, geospatial indexes, text indexes and hashed indexes. And all these indexes also have different properties like unique indexes, partial indexes, sparse indexes and TTL indexes. MongoDB also supports secondary index when the data is nested and one can use this secondary indexes to query the data efficiently.

Suppose we want to perform a search on our movie database based on actors and directors names since these are stored in embedded document form searching would mean to scan through all the records which is not efficient. Here we can create an index on the names using the query shown below, this way when we

perform any search operation the query would be much faster and would not require whole file scan.

```
db.Movies.createIndex( {
    "actors.first": "text",
    "actors.last": "text",
    "directors.first": "text",
    "directors.last": "text"
} )
```

MongoDB also supports a multikey index which for the contents that are stored in array formats. When an index is created on the array field it creates a separate index entry for all the elements in the array. MongoDB automatically determines whether a multikey index creation is needed or not, no need to explicitly mention it. For example, if we create want to create an index on genre field which is stored in an array format, we just specify the index creation using the query shown, MongoDB will automatically create a multikey index for genres.

```
db.Movies.createIndex( {
    "genres": "text"
} )
```

2.2.1 Comparing with SQL. : Unlikes MongoDB, not all SQL databases provide support to text indexing other than that indexed supports is pretty much strong in both MongoDB and SQL databases.

2.2.2 Comparing with Other NoSQL. : Other NoSQL database like Cassandra, are limited when it comes to secondary indexes and only allows querying for equality search and that too on a single column. If we don't have any nested structure than querying with primary indices works best in both NoSQL databases.

3 QUERY PROCESSING AND OPTIMIZATION

3.1 Query Processing

MongoDB allows the creation of databases and collections plus inserting, updating and deleting documents from the collections by providing the query support using document bases query language. It supports CRUD (Create, Read, Update and Delete) operations. The example of finding a document in the database is as shown below:

```
db.Movies.find({
   "title":/^Jurassic/i ,
   "directors.first":/^steven/i,
   "directors.last":/^spielberg/i
}).pretty()
```

The MongoDB provides rich ability to analyze data in place by making use of Aggregation Framework. This framework allows developer to create complex query that goes through the query processing pipelines to perform data analytics and certain transformations [3]. The MongoDB is rich in providing visualizations for the data present in it's collections. For instance, MongoCharts and plug-ins to connect to other business intelligence tools. Since, MongoDB uses javascript to query the data, developers can write expressive update queries that helps in processing complex updates for matching elements including those embedded in nested structures. All this is possible in a single transactional update operation [3].

Consider an example of where we want to find all the actors who worked in a comedy film with director Woody Allen. For this search, the MongoDB query using aggregation pipeline would be as simple as given below. Each stage searches for the particular result set and passes to the next stage in the pipeline. There is no need to store any of the intermediate results thus making the process simple and efficient.

```
db.Movies.aggregate([
    {$match:{"directors.first":/^woody/i,
          "directors.last":/^allen/i,
          "genres":{$regex:/^comedy/i}}},
    {$unwind:"$actors"},
    {$group:{_id:"$actors.id",WAActorMovies:
        {$push:"$id"}}},
    {$project:{_id:1,WAActorMoviesCount:{
        $size:"$WAActorMovies"}}},
    {$match:{WAActorMoviesCount:{ $gt: 3}}},
]).pretty();
```

3.1.1 Comparing with SQL. : As we saw above, MongoDB provides rich aggregation framework, on the other hand, SQL has very restricted aggregation framework and the size of complex queries tends to grow very fast due to the need of storing the data in normalized form. Furthermore, use SQL requires some kind of ORM (Object Relational Mapping) layer which translates objects in codes to the relational tables, MongoDB doesn't require this because of it's flexible model. Apart from that if we want to consider querying the SQL database for same query of finding the actors associated with woody allen, the query would become so complex performing at least 4 joins at different levels.

3.1.2 Comparing with Other NoSQL. : Since MongoDB is schema-less, it supports dynamic querying unlike CouchDB, which support regular queries even though CouchDB is also schema-less [4]. Taking an example of another NoSQL databases like Neo4j doesn't even have aggregation framework.

3.2 Query Optimization

The normal search will scan through the documents in the collection which might be slow. To speed up the process and to find the results faster MongoDB also provides indexing flexibility which was discussed in section 2. The set of indexing options includes text querying and query containing unique indexes and in some cases query on geospatial data. It also provides the feature to analyze the query performance using the functionality of *explain* which breaks down the query structure and provides the statistical information about the performance of the query [2]. This data can be used to analyze and optimize the query further.

Apart from that the MongoDB query optimizer also uses query plans to optimize the query performance [2]. It creates most efficient query plan possible for a query based on all the indexes available and caches this plan. Later it uses this plan each time the query is run to make the query faster. Again explain function can be

used to study this query plans and statistics about them. When we perform a search on using directors and actors name after creating the index shown in section the query performance increases a lot. Doing a search on the data without indexes takes approximately 1700 milliseconds whereas creating index reduces it drastically to 2-3 milliseconds. Apart from that when we perform the complex Woody Allen query which was explained above without indexes that too consumes 5000 milliseconds but with indexes, it reduces to 6-7 milliseconds. Now when we compare this with performance on SQL database with indexes after joins it overall consumes 2.5 seconds which is very slow compared to MongoDB. All this query performance test was performed on a database containing millions of records.

```
db.Movies.find({
  "actors.first":/^tom/i,
  "actors.last":/^hank/i
}).explain("executionStats");
```

3.2.1 Comparing with SQL. : A relational database has no such supports for query plan where it caches the plan and reuses it to perform better next time. It supports indexing and optimizing the queries using different types of joins, but despite that support sometimes when the database design is very complex to perform analysis you need to run the queries for days and wait for the analysis to complete. If a user doesn't want to wait then he needs to pay hefty prices for the fast services to gain servers with high CPU capacities and capabilities. MongoDB or any other open source NoSQL databases thus beats SQL because they provide different optimizations and can easily be achieved using commodity servers.

4 TRANSACTION MANAGEMENT AND SECURITY SUPPORT

4.1 Transaction Management

MongoDB supports all of the existing CRUD (Create, Read, Update, Delete) operations on the collections within the same database or across different databases. MongoDB enables the use of multi-document transactions, wherein the atomicity of each document is maintained at all times. When a transaction executes all of the operations in the transaction are saved when the transaction is committed. And the transaction is aborted and all of its changes are reverted or rolled back all if any of the operation transaction fails. None of the operations within the transaction are visible outside of the transaction until the transaction has been committed. MongoDB has a special feature where the commit operation can be retryable, which means that the database will retry the operation once if the commit on the transaction aborts. When a single write operation modifies multiple documents, the modification of each document is atomic, but the operation as a whole is not atomic [2]. The data inconsistency problem across the multiple documents is handled concurrency control mechanism.

4.1.1 Syntax. : Syntax of transaction looks similar to that of transaction syntax used in relational databases. Example

```
s.start_transaction()
//operations
s.commit_trasanaction() or s.
    abort_transaction()
```

4.1.2 Write Conflicts. : Transaction tries to lock the document for the making changes (for read operation there is no need to lock the document). if a transaction fails to acquire the lock on the document, it will fail with write conflict and transaction is rolled back completely.

If a non-transactional write operation tries to modify a document that is currently being held by a multi-document transaction, then that write will block behind the transaction completing. This non-transactional write will be infinitely retried with backoff logic until maxTimeMS is reached.

If a transaction locks a document and has made uncommitted changes to the document, other transactional or non-transactional database operation will be able to read committed values (previous values) of the document and not the uncommitted state document.

4.1.3 Comparing with SQL. : Even though MongoDB is schema-less the transactions are eventually consistent depending upon which shard the updates are being performed. ACID properties supported by most SQL databases is also present in mongoDB and additionally it also provides support for multi-document transactions.

4.1.4 Comparing with Other NoSQL. : MongoDB is the only open source NoSQL distributed database system which provides ACID properties to the transactions.

4.2 Security Support

MongoDB has various encryption, access, and authentication mechanisms to provide data confidentiality and security. MongoDB has a security checklist which is given as follows [2]:

- Enable Access Control and Enforce Authentication
- Configure Role-Based Access Control
- Encrypt Communication
- Encrypt and Protect Data
- Limit Network Exposure
- Audit System Activity
- Run MongoDB with a Dedicated User
- Run MongoDB with Secure Configuration Options
- Request a Security Technical Implementation Guide
- Consider Security Standards Compliance

4.2.1 Comparison with other database systems. Other relational as well as NoSQL databases provides similar security options.

5 NO-SQL DATABASE APPLICATION

To create a movie database in MongoDB, we installed MongoDB and created a database using the following command:

```
use imdb
```

We then created the collection Movies in which we will store all the movie documents using the following command:

```
db.createCollection("Movies")
```

We also added some validations which are submitted in the implementation files. We then loaded the collection with the data using JSON file using the following command:

```
mongoimport --db imdb --collection Movies --
    file imdbdenorm.json
```

The sample format and structure of such a movie document is as shown below:

```
{
   "_id":2548875,
   "title":"Titanic",
   "year":2012,
   "genres":[
      "Drama",
      "Romance"
   ],
   "directors":[
      {
         "_id":1537002,
         "first":"James (I)",
         "last":"Cameron"
      }
   ],
   "actors":[
      {
         "_id":5552225,
         "first":"Leonardo",
         "last":"DiCaprio",
         "gender":"M",
         "role":"#1 Man"
      },
      {
         "_id":26792614,
         "first":"Kate",
         "last":"Winslet",
         "gender":"W",
         "role":"#1 Woman"
      }
   ]
}
```

It provides flexibility is storing movies data. Each movie can be part of more than one genre which we can represent using the list of genres. Each movie can also be directed by more than one director, which again we can represent using the list of directors, but here each directors information is stored as a document. So the value of directors is the list of embedded documents where each document contains the first and last name of director plus unique id. Similarly, we store the list of actor documents which actors information. Each movie can belong to any genre category and it can be more than one, can have more than one director and also more than one actors. MongoDB helps to represent this data easily without the complex relationship between tables and foreign key references headache. Plus allows querying data much easier and faster. We have built an application on top of this dataset which allows inserting new movies into the database and also the features of querying the database using different search criteria. Using the application we can also update the entries, add actors and delete the movie entries too.

5.0.1 Comparing with SQL. : The same data set if we wanted to represent in RDBMS, firstly if we just wanted it to store in denormalized form like we stored in MongoDB it wouldn't be possible so easily. The question would arise how to keep dynamic columns for genres directors and actors as the count is not fixed before hands when we are building a schema, very bad option would be to assign 100 columns and then add null for rest of the empty columns for each movie. So for this dataset, we have no other option but to normalize the schema. Now while doing that too we need to analyze that a particular director can be associated with more than one movie, same as with actors. To represent this accurately we end up creating 7 tables. Tables Movie, Actor, Director, and Genre contains the specific information and then directedby tables show the relationship between director and movie with which director which movie. Similarly with roles and moviegenres. Now when you want to query the database to find details about movie titanic like we did in section 3 the same query would look like shown below in SQL:

```
SELECT *
FROM   movies
       JOIN directedby
         ON id = movieid
WHERE  directorid = (SELECT id
                     FROM   directors
                     WHERE  first LIKE "Steven%"
                            AND last = "Spielberg")
       AND title LIKE "jurassic%";
```

This would just return the movie details and director id. If we want all the information for this movie like it's genre, all the actors who acted in this movie plus other directors details who directed this movie with James Cameron, we will need to do other whole lots of join whereas in MongoDB just one simple query gives us all this information in one go.

6 FINAL REMARKS

From what we learned and understood about MongoDB, it is a great document based database with very high-end supports and features. It is also widely used by many organizations. But we need to be very careful while selecting any database over any other. It needs to analyze based on the requirements of any application. When you know that the schema of the database of your application is not fixed and might vary based on requirements in future, when we know that the writes are going to be very high on databases, when complex query support might be needed on huge dataset with scaling option, MongoDB might be a very good option.

But when we know that data is very sensitive and high transactional atomicity is needed with complex transaction then definitely opting for a traditional relational database will be a good option. As MongoDB do not strongly comply with ACID properties [3].

REFERENCES

[1] Scale Grid. 2016. Cassandra vs. MongoDB. (2016). https://scalegrid.io/blog/cassandra-vs-mongodb/.

[2] Inc MongoDB. 2008. MongoDB Manual. (2008). https://docs.mongodb.com/manual, Accessed July 06, 2018.
[3] Inc MongoDB. 2018. TOP 5 Considerations when evaluation NOSQL databases. (2018). https://webassets.mongodb.com/_com_assets/collateral/10gen_Top_5_NoSQL_Considerations.pdf.
[4] Dr. GP Pulipaka. 2016. The differences between MongoDB and Neo4J. (2016). https://medium.com/@gp_pulipaka/the-differences-between-mongodb-and-neo4j-babd234ac1f7.

A WORKLOAD DISTRIBUTION

A.1 Team member 1

Contributed to the implementation of the MongoDB application, wrote about data storage and indexing section. Also contributed to Overview and No-SQL database application sections.

A.2 Team member 2

Contributed to the implementation of the MongoDB application, wrote about query processing and optimization section. Also contributed to Overview and No-SQL database application sections.

A.3 Team member 3

Contributed to the implementation of the MongoDB application, wrote about transaction management and indexing section. Also contributed to Overview and No-SQL database application sections.