October 5th 2018

Assignment 1 Report

CMPE 432

Carson Cook (10186142) – 14cdwc
Primrose Chareka (10169343) – 14pc13

Table of Contents

1 – MongoDB

1.1 – Summary of Architecture

MongoDB is a document store architecture. Data insertion is conducted via what are essentially JSON objects. A key, or field, corresponds to a data point. Document storage also enables multiplicity, which means it is able to hold multiple sub-objects within each overall object. This allows for changing data, for example blog posts that have variable numbers of comments. It also ensures that all data for one record is held together, making MongoDB ideal for holding large quantities of unstructured data as no schema is required to create document collections.

Multiplicity means there are fewer individual documents, or Mongo's version of tables, and relevant data is stored together. This architecture means there are less queries performed to get a wholistic view of a record's data. The record's data can easily be filtered to specific fields and documents via proximity, range and key-value criteria. Indexes can also be used to ensure document uniqueness as well as improve query efficiency.

MongoDB provides easy, automatic data scaling. Sharding allows for physical partitions using hash, location or range-based algorithms. This scaling is built-in to MongoDB meaning the queries to retrieve or insert data do not change during scaling. The data movement is handled by the chosen storage engine for the MongoDB instances. WiredTiger is the default, providing effective currency control and data compression for performance and efficient storage. The Encrypted engine protects highly sensitive data while the In-Memory engine allows for real-time analytics and high-performance queries. Finally, the MMApv1 is available to interact with legacy MongoDB system.

1.2 – Database Creation and Insertion

As mentioned before, data objects are essentially the same as JSON objects. This makes database creation extremely easy, as no schema is needed, meaning no initial planning is needed and newly formed data can be inserted at will. This also makes insertion easy as the data only needs to be collected in an object. Insertion can be conducted in bulk, making insertion, especially initial database setup, more efficient via batching the insertion jobs. For this assignment, local instances were used.

1.3 – Querying

Querying is performed via JSON objects as well. The functions are the JSON keys, while the function arguments are provided via the JSON values. This is a very different system from SQL databases, which led to a large learning curve to query MongoDB. However, this allows for granular queries that have many operators and values to operate on.

Aggregate functions have a pipeline architecture; queries are performed sequentially, acting upon the result of the previous pipelined query. This structure allows for very granular, specific queries that act on data subsets. Being able to pair down the size of the dataset for future queries also improves efficiency. However, the pipeline structure can make complex queries much more difficult than other systems. Should the dataset be filtered, and then more data needed, it is very difficult to introduce new data while ensuring appropriately filtered results are returned.

Using JSON format for queries introduces complex syntax that is easy to make mistakes; the JSON format forces programmers to use many characters that other query languages do not require. Many aggregate functions also require an "_id" field, enforcing a naming convention that may not make sense for every aggregate query.

The arguments supplied in the aggregate pipeline to perform the three queries are shown in order in Appendix A.1. Except for the replica set and sharding queries, all queries were performed on a localhost instance for ease of access. These were submitted to the database instance via <collection>.aggregate([param1,param2]}.

1.4 – Appropriate Applications

MongoDB is perfect for data that grows and changes. There are no limits on types and numbers of data, so consistently changing data, such as location, should be stored in MongoDB. The dynamic schema also allows for quick development and release, which fits well with start-ups and agile teams. Also, Big Data applications work well with MongoDB for the easy database scaling.

1.5 – Replica Set & Sharding

Average time for one trial	Query 1 (sec)	Query 2 (sec)	Query 3 (sec)
Single Instance	49.79	53.98	58.94
Replica Set	50.57	58.76	63.01
Sharding	102.37	109.23	

Average time for 100 trials	Query 1 (min)	Query 2 (min)	Query 3 (min)
Single Instance	84.21	89.31	97.48
Replica Set	84.64	98.62	104.02
Sharding	169.96	191.39	

The queries took a significant amount of time to complete due to the poor quality of the Azure Virtual Machines. This trade off was made because poorer quality meant cheaper cost to deployment.

Sharding was clearly much slower than the other queries. This is likely because queries were performed in bulk batches, while sharding by nature splits data up. MongoDB enforces the same order of insertion as requested, so with sharding, the bulk batches had to wait to be split up and individually inserted into the appropriate shard, losing the bulk job efficiency of the other queries. Having a replica set only slightly affects performance; replica sets pick up new transactions via a log, so the only load increase is polling a transaction log.

Query 3 was not completed for the sharding configuration, as sharding does not allow for data to be joined.

1.5.1 – Configuration

For both sharding and replica set, virtual machines in Microsoft Azure were used. The same image, Basic A0 Windows 10 Pro, Version 1803, were used. For easy configuration, allow ports were allowed inbound and outbound, however port 27017 was primary used as the MongoDB default. All machines had 0.75 GB of RAM and a max IOPS of 1x300. All machines were in the East US region due to availability.

Once MongoDB was installed on each machine, the following commands were conducted to finish the setup.

1.5.2 – Replica set

mongod –replSet “replicaset” --bind_ip 40.114.79.23, 40.114.70.147

Within mongo: mongo rs.initiate(_id: “replicaset”, members: [{ _id: 0, host: “40.114.79.23:27017”}, { _id: 1, host: “40.114.70.147”}]})

1.5.3 – Sharding

mongod --shardsvr --replSet “shardreplica” --bind_ip 40.114.79.23, 40.114.70.147

Initiate the replica set in the same way as above.

```
mongos --configdb shardreplica/40.114.79.23:27017,40.114.70.147:27017 --bind_ip 40.114.79.23:27017,40.114.70.147:27017
```

Shards were added and then the database enabled to shard with the following, in a similar format for each shard:

```
sh.addShard( "<shradreplica/40.114.79.23:27017")

sh.enableSharding(“dataset2shard”)
```

Note that these virtual machines have since been destroyed in Azure, to avoid cost.

2 – Cassandra

2.1 – Summary of Architecture

Cassandra is a peer-to-peer distributed database designed to handle large amounts of data by distributing workloads over homogenous nodes. With regards to the CAP theorem, Cassandra focuses on availability and partition tolerance by sacrificing consistency. This occurs because all writes are automatically partitioned and replicated through the cluster. The number of replicas which are made is specified by the user when the configure the replication strategy. Users can thus connect to any node in any data centre and submit a read and write request and should be able to receive a response. If a node is to fail, the request can simply be sent to another node in the cluster. However, because Cassandra uses the gossip protocol (in which nodes periodically exchange information about their state with other nodes they are aware of in the cluster, usually no more than 3) it can take sometimes before all copies of the data are updated and hence it is plausible for a user to read old data.

Cassandra uses a compound primary key which consists of the partition key and one or more additional columns. The Partition Key is fed into a practitioner which is a hash function that determines which node data will be written to. That node then replicates the data to as many other nodes as specified by the replication strategy. The additional column, known as the clustering key, then specifies the order in which rows are stored within a node.

2.2 – Database Creation and Insertion

Cassandra was installed on a local machine by downloading it from the apache website and unzipping the installation folder. This was fairly straight forward. Cassandra has a query language known as CQL. CQL was accessed by writing cqlsh in a terminal window while was Cassandra launched. With the proper drivers installed it was also possible to write python to execute a batch of CQL statements. Combined this is how the database was created. The following are the data object which were made to satisfy the required queries.

KEYSPACE – movRating: High level structure which contained the tables for the data and specified the replication strategy as class = simpleStrategy, replication factor = 3

TABLE – ratingData(uid int, iid int, rating int, tstamp int) – Kept all of the rating data

TABLE – uidratedMovies(uid int Primary Key, ratedMovies List<int>) – Used the user id as the primary key and kept the movies rated by a user in a list allowing for the table to be more general

TABLE - avgRate(iid int Primary Key, numRatings counter, totRating counter) – Used the item ID as the primary key and for each entry in with the corresponding iid, incremented the number of ratings the movie had received and the total sum of the ratings

FUNCTION avgRating (num_rating counter, tot_rating counter) – Took the sum of the ratings and item had and divided it by the number of ratings it had received to calculate the average

FUNCTION numItems(my_list List<int>) – Took a list of integers and returned the number of items in the list

APPENDIX A.2 contains the CQL statements used to insert data into the columns

2.3 – Querying

Query three required what traditionally would have been a join operation in a relational database, however, this is not supported in Cassandra. The traditional method of denormalizing the data also doesn't work because of the variable values we are considering. Cassandra is designed for looking up items based on a static key and not a variable column and hence this query is not feasible for the database

2.4 – Appropriate Applications

Cassandra is suited best for application in which writes exceed reads by a large margin. In particular it means Cassandra is good for transaction logging systems such as tracking purchases, or the number of hits received by a web page. Applications which read using a known primary keep and no not need aggregates are also a good use for Cassandra with an example being keeping track of messages sent by a user.

3 – Neo4j

3.1 – Summary of Architecture

Neo4j is a graph database, meaning nodes hold information and are connected to other nodes via relationships. Each relationship is given a type to allow for differentiation and provides linear performance when querying for nodes.

Neo4j is a cluster architecture, so it provides data replication. Write operations are performed on core machines, which then update the replica machines with the raft protocol. Raft ensure a majority of machines in a cluster accept the write operation, which ensures data durability, but comes at the cost of write latency. The replica machines asynchronously poll a transaction log to determine when an update operation is needed. The reading work from queries is spread out amongst the replica nodes.

3.2 – Database Creation and Insertion

When creating a Neo4j database and making the initial data insertion, the structure needs to be planned effectively. Choosing what data is collected into one node, and what relationships connect which nodes is crucial for effective querying.

For this assignment, each user (user ID), rating (value and timestamp), genre (type) and movie (title, url, id, etc.) were nodes, with relationships between each user and the ratings they supplied, each rating and movie it rates, and each movie and each genre it holds.

3.3 – Querying

Querying with the graph database architecture allows for efficient, specific queries. Specifying relationships and nodes on either end of the relationship allows for simplified queries. Retrieving a subset of nodes and relationships is very easy and making operations on those subsets is simple, while joining in new data is still easy. The code for the Cypher queries is located in APPENDIX A.3.

3.4 – Appropriate Applications

As a graph database, Neo4j is appropriate for applications that are visualized as a graph, such as Facebook friends. The graph architecture also allows for data status to be easily retrievable. Status such as where the nodes, or data, is stored is simple to find and easy to keep modular using relationships. Data compliance is a significant problem in the modern climate, which makes graph databases a quality choice for international, or European, data storage applications. This concept is highly applicable to any scenarios where data status and query location statistics are pertinent.

Overall, the relationship structure makes graph databases effective for data that needs to be modular, but still maintain a strong structure.

4 – NoSQL vs SQL Databases

The most obvious difference between NoSQL and SQL databases is the difference in the database design approach. SQL databases are designed to resemble the entities which they are represent. This makes their design much more straight-forward. NoSQL databases however relish in virtually limitless options for structuring data. The lack of structure imposed on NoSQL databases thus forces programmers to worry less about what it is they are representing and more about how they want to represent the data. Efficiency is the main priority of NoSQL databases and so programmers must take a query focused design approach as opposed to an entity driven design model in order to maximize the efficiency of read and write the data

The most obvious difficulty of working with NoSQL databases as opposed to SQL databases is the lack of documentation and support. SQL databases have been prevalent for much longer and have thus gained considerably more documentation, community support and have established best practices. SQL has more official documents online with more accurate information, as well as more community questions and answers that have more detail. However, this means NoSQL systems are more equipped to handle newer problems such as scaling.

SQL also ensures there is more thought to the data structure than NoSQL systems. This makes it harder to change the data structure but means there is more pre-thought as to the data being stored and the application's use in general.

Data scaling is easier in NoSQL than in SQL systems, as previously mentioned. Due to a loser structure, spreading the data across multiple machines is easier, and have automatic handlers to encapsulate the scaling technologies. Scaling is also achieved via bulk insertions and the data storage techniques. Relevant data is kept close together in NoSQL systems, whereas in SQL data is spread across multiple tables. Keeping relevant data together limits the queries needed to retrieve from each table and allows for range-based query efficiency improvements. However, for some NoSQL systems, querying across multiple "tables" is more difficult than in SQL systems, due to the lack of structure.

APPENDIX 1: Database Query Code

A.1 - MongoDB

A.1.1 – Query 1

```
countParams ={"$group": {"_id": "$userId", "count":{"$sum":1}}}
```

A.1.2 – Query 2

```
avgParams={"$group": {"_id": "$itemId", "avgRating":{"$avg": "$rating"}}}

sortParams={"$sort": {"avgRating": -1}}

limitParams={"$limit": 10}
```

A.1.3 – Query 3

```
pushParams={"$group":{"_id":"$itemId", "ratings":{"$push":"$rating"}}}

inParams={"$project":{"itemId":"$itemId","ratings":"$ratings","hasLess3":{"$or":[{"$in":[1,"
$ratings"]},{"$in":[2,"$ratings"]}]}}}

less3Params = {"$match":{"hasLess3":True}}

unwindParams = {"$unwind":"$ratings"}

avgParams={"$group": {"_id": "$_id", "avgRating":{"$avg": "$ratings"}}}

sortParams={"$sort": {"avgRating": -1}}

limitParams={"$limit": topNumber}

joinParams={"$lookup":{"from":
"movies","foreignField":"movieId","localField":"_id","as":"movieInfo"}}

selectParams={"$project":{"movieTitle":"$movieInfo.movieTitle","avgRating":"$avgRatin
g"}}
```

pushParams is used to collect all ratings into a list.

inParams is used to find ratings that are less than 3

unwindParams is used to take ratings out of list form so they can be averaged

In addition to the pipeline argument in the aggregate query, the allowDiskUse option had to be passed as True. This was because the data exceeded the maximum memory limit, so MongoDB needed to write temporary files.

A.2 – Cassandra

The CQL queries are shown below:

A.2.1 – Query 1

SELECT uid, numItems(ratedMovies) from uidratedmovies;

A.2.2 – Query 2

SELECT iid, avgRating(numRating, totRating)) from avgRating;

A.3 – Neo4j

The Cypher queries are shown below:

A.3.1 – Query 1

MATCH (u:user)-[:GAVE]->() RETURN count(u) as ratingsCount, u.id as userId

A.3.2 – Query 2

MATCH (r:rating)-[:RATES]->(m:movie) RETURN m.id as movieId, avg(r.value) as avgRating ORDER BY avgRating DESC LIMIT 10

A.3.3 – Query 3

MATCH (lowr:rating)-[:RATES]->() WHERE lowr.value < 3 MATCH (lowr)-[:RATES]->(m:movie) MATCH (r:rating)-[:RATES]->(m) RETURN m.id as movieId, avg(r.value) as avgRating ORDER BY avgRating DESC LIMIT 10