CS 585/ DS 503 Final Project: Project 5, Team 12

Jackson Powell, Caleb Ralphs

November 2020

1 Problem Definition

As applications seek to use larger amounts of data, those applications need to access that data efficiently, particularly in transactional systems that expect a response within a few seconds. However, to satisfy a query, the database system needs to find each record that matches the conditions to properly fulfill the request. Relational database systems do use some optimizations, such as making efficient query plans to deal with a subset of the data at each step of the query, however, a large component of the performance is still dependent the number of records that have to be checked [4]. One way of efficiently finding those records that match a condition is to create an index on the column that the query searches on [5]. Indexes generally work by creating a smaller data structure that organizes the column that is being indexed and has pointers to the corresponding records with that value [5]. By working with this smaller data structure, the pertinent records can be found on disk and read without needing to do a full table scan.

Efficiently querying data is particularly a problem for NoSQL databases, as they are working with data that may be even larger than the data relational databases must manage, requiring efficient responses to queries. Thus, this report will describe the indexing framework of one such NoSQL document-oriented database, MongoDB, by showing the storage overhead and time performance on a generated, sample dataset with a variety of index types.

2 Existing Solutions

To provide some context for the indexing framework of MongoDB, we will dive into various database and data structures, such as B-tree, Hash Indexes, SQL Database Indexes, and CouchDB Indexes.

2.1 B-tree

B-trees consist of a set of nodes in a tree format, where each node holds a set of n ordered keys and a set of $n+1$ pointers to the next level of the tree [2]. Each

node is at least half-full, except for the root node, which only needs to contain 2 pointers [2]. The leaf nodes contain the keys of the actual column values and pointers to the corresponding records, as well as a pointer to the next leaf node. To satisfy a query, the pointers are followed based on the value in the condition and the keys of the current node until the last level of the tree is reached. Then, nodes are scanned for those values that match the condition, and the respective pointers are followed to get the corresponding records from disk [3]. While the whole B-tree may not fit in memory, due to the restriction on the size of each node, at least one node can be loaded at a time, allowing traversal of the index and access to fewer disk blocks of the data, when compared to a table scan of the records [3].

B-trees can be used to efficiently satisfy range queries, such as *gt* and *lt* operators, and equality queries due to the limited scope of nodes over which they need to traverse and the links existing between leaf nodes [3]. However, for inequality queries, each data node would need to be accessed anyways, due to the likely sparseness of the particular condition value to exclude, leading to ineffective indexing.

To maintain a B-tree, there must be overhead for storage, inserting, updating, and deleting records to maintain the properties of the B-tree described above.

2.2 Hash Indexes

In order to support hashed sharding, a method of providing a more even, random distribution than traditional indexing, MongoDB leverages a hashed index type [5]. The hashed index type refers to the indexing of the hash of a given field value [5]. Upon computing the hashed index, the hash of the shard key is calculated, yielding that hash value instead of the original, actual value of field [6].

The purpose of the resulting, more evenly distributed, shards is to reduce the amount of unbalanced sharding. If the distribution of the shards is skewed, such that some shards have much more data associated with them, then the resulting data chunks can exceed the maximum chunk size [6]. On the other side of the spectrum, without hashed sharding, there may exist empty chunks which can hinder the performance of accessing the indexes. The hashed indexes of hashed sharding may also take up less memory and storage than traditional index keys [6].

Although the hashed sharding and index provides a nicer, more even distribution, the resulting hashed index only supports equality matches, not supporting range-based queries [5].

2.3 SQL Database Indexes

Relational databases, or SQL databases, allow for an index to be created on any column, predefining one on the primary key of a table. Typically these indexes are implemented using B-trees. However, they may also be implemented using static or extensible hashing. Regardless, the indexes require more space to

store the indexing themselves. Generally speaking, the index capabilities and performance implications of MongoDB indexing and indexing in RDBMS are similar. Both systems leverage query optimizers. In RDBMS, the query is optimized at execution time, analyzing possible ways to read the data, selecting the optimal method, unless an index is manually specified by the user, while MongoDB selects the optimal index for specific operations, again, unless the index is directly specified by the user [4].

Some of the differences are that RDBMS database tables create a clustered index on all primary keys in the table; a user must manually set the index if another indexing method is preferred [4]. Instead, by default, a unique clustered index is generated on all primary keys in the table [4]. MongoDB creates an index, by default, on the **_id** field for every collection. By default, the **_id** index can not be deleted and is always a unique index [4]. Additionally, in MongoDB clustered indexes are not featured because the regular indexing method provides all of the indexing needs [4]. Since the clustered indexing refers to the rule of ordering data records' physical location on disc, it is not required by MongoDB because MongoDB indexes uses mapped files to manage and interact with all the data in collections [4]. The last major difference between the two systems' indexing structure is the size limitations. RDBMS has a maximum allowable index key size of 900 bytes, while MongoDB has a maximum of 1024 bytes [4].

Although the systems have their differences, MongoDB, representing a NoSQL database, and RDBMS, representing a SQL database, both have powerful index implementations.

2.4 CouchDB Indexing

CouchDB is a NoSQL document-oriented database system, similar to MongoDB, that also uses indexes [1]. However, whereas MongoDB supports a fairly complex set of queries, CouchDB maintains a CRUD API with support for repeated queries on aggregated data using a MapReduce framework [1]. With this framework, CouchDB allows users to create a view, which is the result of applying a map function to a collection. Users may then apply a reduce function to get the result of an aggregation. CouchDB uses indexes only on the **_id** column of the documents in the database and of the views, whereas MongoDB allows the user to specify an index on any field, or the query optimizer selects the optimal index [1]. In the CouchDB database, the id consists of either a randomly generated unique Object ID, while the id in the views corresponds with the keys of the groups created by the map function [1]. These indexes also use a B-tree as the underlying data structure on the **_id** field. The aforementioned views in CouchDB are very similar to how indexes are used in SQL or RDBMS.

3 Simplifying Assumptions

To narrow the scope of the project, some aspects of the MongoDB indexing framework were neither tested in the experimental study nor discussed in de-

tail in the report. One aspect that we do not cover is the overhead of using MongoDB indexes beyond the associated storage cost, including the time cost of maintaining the indexes when documents are inserted, updated, or deleted. Another aspect of indexes that we did not cover in our experiments are some other index properties, such as unique, sparse, partial, case insensitive, or TTL indexes. We also did not cover all combinations of different types of fields. For example, instead of testing multikey indexes on arrays of documents, numbers, Strings, and GeoJSON, only two multikey indexes are tested and compared with the different indexes on those types.

Similarly, not every use case was tested on every index due to their applicability or requirements for an index. For example, queries over the 2dsphere index only used GeoJSON object query operators, such as $near and $geoWithin, because that is the purpose of such an index, and other more general use cases may not be as applicable for this type of data. However, both the $near operator and $text operator, for querying over the text index, require that an index of the respective type exists for the query to succeed.

The scope of this project was further limited with regards to the equality of conditions under which the different versions of the queries are run. This is particularly an issue with caching, as the first query set may need to load more data from disk, while successive queries will be able to get that data from the cache, thereby, creating a large difference in the execution time, even if the index was not primarily a factor in that difference. One way that this caching issue was mitigated was by running the query three times, each with and without the index hidden in different orders and then reporting the median of those times as an attempt to reduce outliers for queries not able to use the cache for the first run.

Due to limitations of available machines, the performance tests for this project were all run on a single local machine. With using a local machine, the conditions of available memory and processing power may have been somewhat variable during the experiment runs, causing some inconsistencies which may be partially mitigated by taking the median over the three test runs. Additionally, with only a single machine, this project did not cover performance over sharded clusters using hash indexes on the shard keys.

4 Solution

4.1 Sample Collections

To run the experiments on time and storage performance of indexes in MongoDB, some sample datasets, or collections, were created. The difference between each of the collections that were created were the size of the collections, i.e., the number of documents contained in the collection. Each document in the collection has the same format, which is as follows:

```
{
    _id: [random ObjectId generated by MongoDB],
```

```
    name: [random string of length 5],
    home: [random GeoJSON point],
    salary: [random int 0-100000],
    birthday: {
        year: [random int 2000-2020]
        month: [random int 1-12]
        day: [random int 1-30]
    },
    friends: [random list of names from other documents],
    favoriteLocals: [list of random GeoJSON points, length 3]
}
```

These collections were generated through JavaScript functions and inserted into a list, referred to as entries. The entries were, then, inserted into the database using the following MongoDB function call:

```
db.people.insertMany(entries)
```

The sizes of each of the collections, the entries list, were determined by the number of documents in the list. The numbers of documents in each list of entries were 1 million, 10 million, and 250 thousand, inserted into collections with names people1, people10, and people250k, respectively. Since the insertMany statement has a maximum insert size of 100000 documents, the entries were inserted into the respective collection in batch sizes of 99999.

4.2 Indexes

The format of the documents for the dataset were chosen to provide a range of fields to test the different kinds of indexes that MongoDB provides. These index types include [5]:

- Simple: An index on any field with a single value. This value can either be singular, a subdocument, or a field in a subdocument.
- Compound: An index on multiple fields in a particular order with a sorting order for each field.
- Multikey: An index on a field with an array value, where each value in the array is added to the index.
- Geospatial: A category of two indexes (2d and 2dsphere) that allow for efficient queries over geometries by hashing coordinate pairs into regions of the applicable area.
- Text: An index on one or multiple string fields that supports string search over those fields using language-specific knowledge for stemming and stop words.
- Hashed: An index on the hash-value of a field, such that only equality searches are supported over the index.

Each of these index types were tested individually by creating a simple index on salary, a compound index on salary and birthday, multikey indexes on favoriteLocals and friends, a geospatial 2dsphere index on home, a text index on all string fields (specified by the wildcard $**), and a hash index on name. The command for creating these indexes is generically "db.collection.createIndex(field: index type or sort order)". Collection is replaced by the collection that is having an index created on it (e.g. people1, people10, or people250k). Field is the field that the index is being added to. The value for that field is the sort order, 1 or -1, for generic indexes, such as simple, compound, and multikey, or the type of the index for other index types which are "2dsphere" for geospatial 2dsphere indexes, "text" for text indexes, and "hashed" for hash indexes. While the sort order is straightforward for simple data types, for whole documents it appears that the sort is based on successively sorting by the fields in the order that they are specified. In this manner, the documents are sorted by the name of the first field, the first field's value, and then successively the next field name and values as long as more tiebreakers are needed. MongoDB also creates an index on the _id field for every collection. All of these indexes were created for each of the three collections that were used for testing.

4.3 Queries

Each of the indexes were tested on each of the collections by creating a set of queries where each query in a set tests a particular index for some use case. A particular index is tested for each query by using the hideIndex and unhideIndex commands to isolate that index for testing. These commands are "db.collection.hideIndex(indexName)" and "db.collection.unhideIndex(indexName)" respectively. Collection is replaced by the name of the collection that the index is being hidden or unhidden on. indexName is the String name of the index which is created by joining the value(s) and field(s) specified in the createIndex command with underscores. Alternatively, indexName may be replaced by the same document that was used in the createIndex command. When an index is hidden, it is not used by MongoDB, even if it could make a query more efficient. However, the _id index cannot be hidden and some queries do not work unless there is an appropriate index unhidden and available.

The specific queries that were tested for each index will be covered in Section 5 to provide context closer to the description of the results for those queries. Generally, the use cases that were covered by these queries included equality search, range search, inequality search, search over multiple fields, aggregation queries, and recursive queries. However, not all of these use cases were tested for every index as described in Section 3.

4.4 Code Structure

There are two main sections of the JavaScript code used to create this experiment over MongoDB indexes.

The first section generates and inserts the dataset, creates the indexes, prints the stats of the collections with the created indexes, and then hides all of the indexes for the experiment to start off with. This section only needs to be run once to populate the collections with data and indexes and can be run in the Mongo shell using the command "load('Generator.js')". The Generator script then loads "GenerateDataset.js" to create and insert the entries into the collections, "GenerateIndexes.js" to create the indexes and print stats on the collection, and "HideAllIndexes.js" to hide all of the indexes.

The second section runs the experiment over the dataset generated by the first section. The experiment is run using the command "load('Experiment.js')" which iterates through the collections to test and runs all of the queries on each collection, printing out when one collection starts testing. For each collection, "Queries.js" is loaded to run the function runAllQueries which loads separate files that isolate the queries run for a particular index and contain their own version of runAll¡field¿Queries for running all tests for a field. Each query file loads "QueryTemplate.js" to make use of a queryRunner function that prints given information about the query being run, statistics for the performance of running without using an index, and statistics for running with an index by unhiding a given indexName. By using this template function, queries can be easily added by specifying Strings to describe and identify the query within the overall experiment, the name of the collection to test the query on, the index that should be unhidden to test running the query with, and the function for running the query that statistics should be calculated on.

4.5 Metrics

To measure the performance of the indexes for MongoDB, two sets of metrics were used, one for either of the two code sections.

The first metric is used after generating the collections and indexes by using the command "db.collection.stats()" where collection is the name of the collection to get the statistics of. From the resultant object, the storage overhead of the indexes can be found in the indexSizes field and the total size of the records in the collection can be seen in the size field. All of these sizes are in bytes, unless a scale factor is specified in the call to stats.

The second set of metrics is used for each time a query is run in "QueryTemplate.js". Within this set, there are two versions of the metrics depending on the type of query being run.

For simple queries using one call to find, then statistics can be extracted about the execution by using the resultant cursor from the query to call cursor.explain("executionStats"). The statistics that are printed from the resultant object include the execution time, whether the query succeeded, which index was used if any, how many documents the query looked at, how many keys the query looked at, and the number of documents returned in the result. The main performance metric that will be reported from this is the time taken for the query, however, the additional information clarifies whether the index was used, shows that results with and without the index are consistent, and how an

index can be more efficient in terms of the number of documents accessed even if that is not reflected by the execution time.

For more complex aggregation and recursive queries, using the explain method would not cover all of the steps within the query. Instead, only the performance time as measured by the JavaScript method Date.now() is reported as well as the results from the query for checking consistency with and without the index.

5 Experimental Study

The following results were collected by following the procedure laid out in Section 4. These results are organized by, first, reporting the storage overhead of each of the indexes that were used for each collection, and then, reporting the performance of each query for each index, with some commentary describing the possible reasons for those results. The units for the storage overhead are in megabytes and the units for the performance time of the queries are in milliseconds.

The performance tests used hiding and unhiding indexes, instead of the hint() method to compare a query with or without the index. Hint was not used in order to show when MongoDB would use an index if it was available. In some cases, an index was not used even if it was available, likely due to the MongoDB query optimizer. If this is the case for a query version with an index, then instead of just listing the time, the table entry will be in the format of "NA (time)", to indicate that the index was available but not reportedly used with the given execution time. However, this can only be determined for those queries which used the explain() method to get the statistics, so there may be cases of aggregation or recursive queries that do not use the index, but do not report that the index was not used.

5.1 Storage overhead

After creating the indexes for each collection that would be tested, the storage overhead was found using the db.collection.stats() command. This command returns a document containing an "indexSizes" field that lists the storage required for each index on the collection in bytes. These sizes were then divided by 2^{20} to get the storage overhead in megabytes which is reported in Table 1.

Index name or collection property	people1	people10	people250k
Number of documents	1 million	10 million	250 thousand
Size of records in collection [Mb]	417	4168	104
_id_	9	96	2
salary_1	8	75	2
salary_1_birthday_-1	28	200	8
home_2dsphere	18	173	4
name_hashed	18	176	4
$**_text	41	406	10
friends_-1	9	86	2
favoriteLocals_1	82	810	20

Table 1: Storage overhead in megabytes for creating indexes on each collection

5.2 Simple index: Salary queries

Six queries were tested over the simple index on the numeric salary field. All of these queries used the index when it was available, even if doing so resulted in a slower execution time. The queries were:

- S1: A range query over approximately half the collection that does not sort the results.
- S2: A range query over approximately half the collection that sorts the results in ascending order.
- S3: A range query over approximately half the collection that sorts the results in descending order.
- S4: A range query over approximately half the collection that does not sort the results and only projects the salary field to make this a covered query.
- S5: An equality search on the salary of the first document retrieved from the collection using find()
- S6: An inequality search on the salary of the first document retrieved from the collection using find()

QueryId	Index	people1	people10	people250k
S1	salary_1	1076	13468	207
	none	389	4163	100
S2	salary_1	1116	13464	205
	none	1930	22674	252
S3	salary_1	1103	13838	212
	none	1836	19010	249
S4	salary_1	353	3467	80
	none	574	5335	127
S5	salary_1	0	0	0
	none	401	4087	100
S6	salary_1	2282	27774	474
	none	404	4000	98

Table 2: Execution time in milliseconds for queries over the salary_1 index

Based on the performance of each of the queries shown in Table 2, with and without the salary index being available, some conclusions can be made about simple indexes. One such conclusion is that covered queries, where the only involved fields for selection and projection are indexed fields, are very fast, having negligible execution time when an index is present, when compared to up to 4 seconds for the largest collection without the index. Simple indexes can improve the performance of sorting on the indexed field by approximately the same amount for either direction of the sort. Inequality searches take much longer when using an index due to the sparseness of records not to include, and surprisingly, MongoDB did use the index for the inequality query even without the hint command being used to force its use. Similarly, the range query without sorting took longer with an index, and an index was used without the use of the hint command. This slow performance by the index may have been due to the range being so large that a large portion of the data gets read and the fewer random reads becomes more costly than more sequential reads when executing without the index. Overall, the queries over the salary index show generally expected results for the performance of B-trees on generic use cases, with the main unexpected results being that the index was still used to satisfy the query even when it was slower than not using it.

5.3 Compound index: Salary and birthday queries

Five queries were tested over the compound index on the salary and birthday fields. These queries focused on the properties of compound index prefixes and the conditions under which the index could be used. The queries were:

- SB1: An equality search on the salary of the first document retrieved from the collection using find(). This checks the performance of an equality selection on the prefix of the compound index.
- SB2: An equality search on the year of the birthday of the first document

retrieved from the collection using find(). This checks the performance of an equality selection that does not use the prefix of the compound index.

- SB3: An equality search on the salary and birthday year of the first document retrieved from the collection using find(). This checks the performance of an equality selection on all of the fields of the compound index.
- SB4: A range search on the salary field over approximately a tenth of the collection. This checks that both range queries on the prefix of the compound index and where there are fewer results than half the collection.
- SB5: The same range query as SB4, but sorted by the salary field. This checks using the prefix for both a range query and sorting.

QueryId	Index	people1	people10	people250k
SB1	salary_1_birthday_-1	0	0	0
	none	385	4052	94
SB2	salary_1_birthday_-1	NA (407)	NA (4113)	NA (100)
	none	411	4174	99
SB3	salary_1_birthday_-1	0	0	0
	none	426	4245	106
SB4	salary_1_birthday_-1	245	3001	68
	none	382	3883	95
SB5	salary_1_birthday_-1	239	2943	68
	none	526	6542	127

Table 3: Execution time in milliseconds for queries over the salary_1_birthday_-1 index

The conclusions that can be drawn from the results shown in Table 3 include that queries on indexed fields execute very quickly as long as the query is on the prefix of an index, i.e., SB1 and SB3 querying over salary and both salary and birthday respectively. If the query is not on the prefix, then the index cannot be used efficiently, as seen by SB2 not using the compound index when it was not hidden for querying over the birthday field. It can also be seen that use of the prefix of the compound index can speed up both range queries and sorting on that field. The result for the range query indicates that the slowness of S1 with the index was likely due to the large number of documents in the result of the query because the index improved performance when the range was smaller.

5.4 Multikey indexes: Friends and locals queries

Five queries were tested over the friends multikey index and three queries were tested over the favoriteLocals multikey index. These queries were generally used for testing the efficiency of the indexes for element matching over the arrays, membership queries, array equality, and array attributes. The queries were:

- F1: A query finding people who have the name of the first document in the collection as a friend. Checks efficiency of determining membership in a list.
- F2: Queries people that have zero friends. This checks the efficiency of determining attributes of the whole array in the edge case of an empty array.
- F3: Queries people that have four friends, which is the maximum number of friends that someone can have in the generated collections. This checks the efficiency of determining array attributes in more general cases.
- F4: Queries for people with the exact same array as the first document in the collection. This checks the efficiency of full array matching.
- F5: Queries people that have friends of the first document in the collection, but are not friends themselves of that first document. This checks more complex membership queries.
- L1: Queries for people that have at least one favoriteLocal that is within 1 latitude and longitude of WPI. This checks element matching using a manual version of the geospatial operator.
- L2: Queries for people that have at least one favoriteLocal that is within 1 latitude and longitude of the first document's home. This also checks element matching using a manual version of the geospatial operator.
- L3: Queries for people that have a favoriteLocal that is in the northern hemisphere or in the western hemisphere. This checks queries over the array not using elemMatch.

QueryId	Index	people1	people10	people250k
F1	friends_-1	0	0	0
	none	554	5759	137
F2	friends_-1	NA (417)	NA (4225)	NA (102)
	none	416	4150	101
F3	friends_-1	NA (449)	NA (4246)	NA (102)
	none	418	4394	105
F4	friends_-1	53	4035	14
	none	627	5629	142
F5	friends_-1	171	0	22
	none	628	5491	151
L1	favoriteLocals_1	NA (1585)	NA (16256)	NA (466)
	none	1897	21504	417
L2	favoriteLocals_1	NA (1385)	NA (13209)	NA (348)
	none	1457	13689	351
L3	favoriteLocals_1	NA (3568)	NA (31621)	NA (1106)
	none	2556	29712	1100

Table 4: Execution time in milliseconds for queries over the friends_-1 and favoriteLocals_1 indexes

Based on the results from queries over the friends index as shown in Table 4, it can be seen that both membership and exact array matching can make use of the multikey index to significantly improve performance. However, queries over attributes of the array, such as the size, can not take advantage of the index.

Based on the results from queries over the favoriteLocals index which is also shown in Table 4, it can be seen that none of the queries were able to use the index to improve efficiency of the query. Each time the index was available it was ignored, and if the hint method was used, performance of the query using the index got several times worse than without the index. In the future, element matching should be tested on arrays with elements of simpler data types, rather than GeoJSON objects, which should make use of geospatial indexes for queries, in general, as shown by the home queries.

5.5 Geospatial 2dsphere index: Home queries

Three queries were tested on the geospatial 2dsphere index on the home field, one for each applicable GeoJSON operator. The queries are as follows:

- H1: Tests the $geoWithin operator for querying people with point values in the home field that are within a defined polygon over a larger area that includes WPI.

- H2: Tests the $near operator for querying point values in the home field that are within a certain distance of a point near East Hall.
- H3: Tests the $nearSphere operator for querying point values in the home field that are close to East Hall, without needing to specify limits for how close the point needs to be.

QueryId	Index	people1	people10	people250k
H1	home_2dsphere none	0 1376	1 14039	0 324
H2	home_2dsphere none	1 Error	8 Error	0 Error
H3	home_2dsphere none	10006 Error	355385 Error	2172 Error

Table 5: Execution time in milliseconds for queries over the home_2dsphere index

The results from these queries shown in Table 5 show that two of these operators $near and $nearSphere require a geospatial index to exist on the queried field. The results also show that for the one operator that can be run without an index, $geoWithin, the index makes the query take negligible time to be much faster than the, up to 14 seconds, that the query takes on the largest tested collection.

5.6 Text index: Text wildcard queries

Three queries were tested over the wildcard index on text fields to try the different use cases of the $text operator. The queries are as follows:

- T1: Searches all text fields of each document for the name of the first document in the collection.
- T2: Searches all text fields of each document for the name of the first document in the collection, sorting results by the textScore for how close they are to the search term.
- T3: Uses an aggregation pipeline that counts the number of search results for the name of the first document in the collection.

QueryId	Index	people1	people10	people250k
T1	$**_text none	0 Error	0 Error	0 Error
T2	$**_text none	0 Error	1 Error	0 Error
T3	$**_text none	247 Error	0.5 Error	27 Error

Table 6: Execution time in milliseconds for queries over the $**_text index

The results shown in Table 6 show that the $text operator must be used with a text index and appears to perform quickly when using the index based on the negligible times for all of the queries that used the index.

5.7 Hashed index: Name queries

Five queries were tested for the hashed index on the name field, checking performance over equality, range, aggregation, and recursive queries. The queries are as follows:

- N1: Equality search on the name of the first document.
- N2: Range query on name's greater than the name of the first document.
- N3: Inequality search on the name of the first document.
- N4: Aggregation on the count of documents, min salary, and max salary, grouping by name.
- N5: Recursive query that finds the documents with a given name and finds the documents of their friends, processing each in the same way. Starts with the name of the first document.

QueryId	Index	people1	people10	people250k
N1	name_hashed	0	0	0
	none	386	4064	99
N2	name_hashed	NA (388)	NA (4265)	NA (96)
	none	392	4058	94
N3	name_hashed	NA (400)	NA (4826)	NA (97)
	none	400	4402	98
N4	name_hashed	6255	72221	679
	none	5590	74078	743
N5	name_hashed	2135	4172	289
	none	1972	4162	284

Table 7: Execution time in milliseconds for queries over the name_hashed index

The results shown in Table 7 show that hash indexes are not applied to range or inequality queries while performing much faster for equality queries. However, this performance improvement for equality search does not seem to translate to aggregation or recursive queries on the hashed field.

References

[1] Apache. *CouchDB*. URL: https://docs.couchdb.org/en/stable/intro/overview.html. (accessed: 12.08.2020).

[2] Cornell. *B-Trees*. URL: https://www.cs.cornell.edu/courses/cs3110/2012sp/recitations/rec25-B-trees/rec25.html. (accessed: 12.08.2020).

[3] Guy Harrison. *Effective MongoDB Indexing (Part 1)*. URL: https://dzone.com/articles/effective-mongodb-indexing-part-1#:~:text=The%20B%2Dtree%20(%E2%80%9Cbalanced,tree%20is%20the%20header%20block.. (accessed: 12.08.2020).

[4] Victoria Malaya. *SQL vs. NoSQL*. URL: http://sql-vs-nosql.blogspot.com/2013/11/indexes-comparison-mongodb-vs-mssqlserver.html. (accessed: 12.08.2020).

[5] MongoDB. *Indexes*. URL: https://docs.mongodb.com/manual/indexes/. (accessed: 12.08.2020).

[6] Dharshan Rangegowda. *The Case for MongoDB Hashed Indexes*. URL: https://scalegrid.io/blog/the-case-for-mongodb-hashed-indexes/. (accessed: 12.08.2020).