COMP5338: Assignment 1

Jesse Serina Narvasa | jnar3156

Introduction

This report aims to provide findings on the optimal implementation of each MongoDB query workload. There are six query workloads for implementation, with the dataset being focused on the two JSON files of Tweets and Users, as extracted from the Twitter API. The performance of the six query workloads will be analysed through the use of execution statistics, with the implementation being adjusted accordingly. Moreover, two of the query workloads will also be provided with an alternative implementation, with the intention of comparing the original and alternative for difference in its efficiency.

For the remainder of this report, the dataset used within the tweets and users collections are related to **Yosemite**. As such, specific references to values from the query's execution statistics will be based on the Yosemite-related Twitter data. Overall, there are 10,000 documents within the tweets collection and 7,385 documents within users collection.

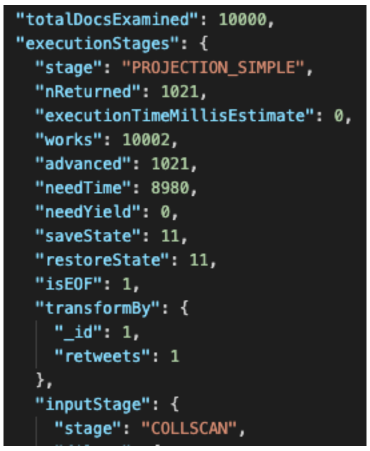
Performance Analysis of Query Implementations

Query 1

The objective of this query is to find the number of general tweets with at least one reply and one retweet within the dataset. Since this query will be requiring to ensure that the tweet is a general tweet, and hence replyto_id and retweet_id fields will have to checked, an index will have to be considered for addition in order to ensure our query is well optimised.

In this instance, because we are looking for tweets where retweet_id and replyto_id do not exist, an implementation of a partial index may not help. While it is possible to create a partial index, with a filter expression checking for each field being false, this index will have very limited usage just for this particular query. Other queries which will be interested in connecting the replyto_id or retweet_id to the parent tweet will not make use of a partial index which is limited to just the documents whom does not have this field. On the contrary, setting a partial index to "exists: true" for both fields will not have these documents indexed, and therefore result in a collection scan. Hence, the approach to be taken is to create a separate general index for retweet_id and replyto_id. This non-sparse index can therefore be used by multiple queries, particularly on instances where we look-up on each field separately, and hence why we don't make a compound index.

Without creating any index, the match stage of this query will result in a full collection scan on the 10000 documents. This further cascades onto the next lookup stage, where without the retweet_id field, the 1021 general tweets returned from the previous match stage will then be looked up if it's been retweeted within the dataset. This results in a total of 1021 * 10000 documents being examined. This also highlights the importance of having an index on the foreignField of a lookup stage. The project and match stages then come after, with the documents being filtered to just those with at least a retweet, hence reducing it to 232 documents. However, without an index on the replyto_id field, the next lookup stage would result in 232 * 10000 documents being examined, again highlighting why we should also create an index for replyto_id field. The remaining project and match stage then filters the documents returned to those with at least one retweet and also one reply tweet. This reduces it to just 23 documents or tweets.



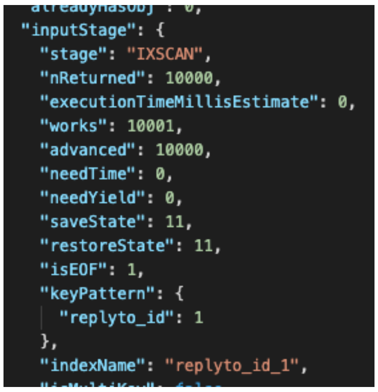
```
"totalDocsExamined": 10000,
"executionStages": {
  "stage": "PROJECTION_SIMPLE",
  "nReturned": 1021,
  "executionTimeMillisEstimate": 0,
  "works": 10002,
  "advanced": 1021,
  "needTime": 8980,
  "needYield": 0,
  "saveState": 11,
  "restoreState": 11,
  "isEOF": 1,
  "transformBy": {
    "_id": 1,
    "retweets": 1
  },
  "inputStage": {
    "stage": "COLLSCAN",
```

Figure 1 Lookup stage performing a collection scan when no index is found

On the contrary, with the two indexes, one for replyto_id and retweet_id being created, we are able to proceed with the query workload much more efficiently. On the first stage, the winning query plan is with index on retweet_id. This is used instead of the other two rejected plans of replyto_id index and also the use of an index intersection between retweet_id and replyto_id, as denoted by "AND_SORTED". As a result, the total documents examined is 2119 instead of the full collection scan. The next stage which is the lookup, utilised the retweet_id index, and hence did not have to examine the 1021 documents returned from the previous stage against the whole collection. Through the use of the index, it only examined 7042 documents. After the project and match stages, the number of documents or tweets were reduced to 232 as before. The next lookup with replyto_id as the foreign field, also only resulted in 46 documents being examined, making use of the other replyto_id_1 index. The project and match stages then reduces it down to 23 documents as before, resulting in the same output.

Query 2

The second query workload for implementation, is one that'll allow us to find the reply tweet that has the most retweets in the dataset. The approach for this query, is therefore to first filter though the tweets, only keeping the tweets which have the replyto_id field, and hence indicating that it is a reply tweet. A lookup stage is then used to find the tweets which are a retweet of that reply tweet, which will allow us to find the number of retweets for each reply tweet, and hence sort it in descending order to return the reply tweet with the most retweets.



```
  "alreadyHasObj": 0,
"inputStage": {
  "stage": "IXSCAN",
  "nReturned": 10000,
  "executionTimeMillisEstimate": 0,
  "works": 10001,
  "advanced": 10000,
  "needTime": 0,
  "needYield": 0,
  "saveState": 11,
  "restoreState": 11,
  "isEOF": 1,
  "keyPattern": {
    "replyto_id": 1
  },
  "indexName": "replyto_id_1",
```

Figure 2 Index scan examining the entire collection when using "$exists: true"

Within the first stage, careful consideration must be made with regards to the exact match condition that is to be used. In our implementation, it is notable that we chose to use "$ne: null" instead of "$exists: true". This is because using the exists operator to look for true will actually result in the whole collection being examined, even though replyto_id index is still technically used. This is because in non-sparse index like replyto_id_1, documents which do not have the field replyto_id are considered to have null value. Hence, looking up for replyto_id if it exists will still cause all documents in the collection to be examined because of the way general index treats documents where that field does not exist, being considered as having value of null in that field. In contrary, using "$ne: null" allows us to exploit this behaviour within general indexes, since we know that documents which do not have this field will be treated with having null value, and hence be able to filter through documents without this field effectively, and resulting in only 1098 documents being examined.

In the following lookup stage, retweet_id field will then be used to lookup for tweets in which the reply tweet ID is considered to be the parent of the retweet. This stage results in just 218 documents being examined, meaning that, out of the 1098 documents returned from the previous stage, those 1098 tweets actually have only been retweeted 218 times.

The project stage then calculates the size of the array field retweets, and consequently returns the same number of documents, 1098, to be passed to the next stage. Lastly, it is noted that the sort and limit stages are executed together as one stage, and because there is a limit of only 1, then global sorting is not required, resulting in memory usage of less than 1KB.

Query 3

In this query, the objective is to return the top five hashtags which are appearing as the first hashtag mentioned within general or reply tweets in the dataset. As such, hashtags embedded in retweets should not be considered. Moreover, the requirement of this query also needs to group the hashtags in a case-insensitive manner, which will need to be considered for the implementation.

For this query workload, we note that we start making use of the hash_tags.text field. As such, we consider making an index since it will be used for checking within the match stage to filter through tweets to only those with hashtags embedded, and also future queries which might be interested in looking up particular hashtags, such as Query 4. In this instance, the chosen implementation is to create a partial index for hash_tags.text. Using a partial index fulfills our requirements of being able to quickly filter through documents based on specific hashtags, and also in this particular query, which is to find the documents which does have hashtags embedded within. The other important consideration of using a partial index vs a general index however, is the storage saving, whereby a partial index only takes 28KB in comparison to 60KB of a general index on the same field.

The first stage of the query consists of a match, with an "and" condition that the tweet has the hash_tags field, and are also either a reply tweet or a general tweet. We can see that this stage makes use of an index scan on retweet_id to find the general tweets, and an index scan on replyto_id to find the reply tweet for the winning query plan. One of the rejected plan is to use replyto_id for checking both the general tweets and reply tweets. An intersection of both retweet_id and replyto_id indexes has also been rejected. Overall, the total documents examined is 4,238 with 241 tweets being returned. In determining the winning query plan, the threshold for the number of documents is 101 in this instance. The other two were only able to return 19 and 13 respectively, within the same timeframe the winning query is able to return 101. Another aspect to note, is that the projection stage which is set to be executed after, is treated within the same stage as denoted by "transformBy" in the execution plan.

It is also worth noting that within the match stage, it is required to have the condition ensuring that the hash_tags field exists. Otherwise, within our groupings, tweets which do not have a hashtag will be grouped together and treated to have the hashtag of an empty string. In order to prevent tweets with no hashtags from being grouped together, then this condition must exist within our match.

The subsequent group stage is then used to group together the same hashtags, irrespective of case, and perform a count, to allow us to capture the number of instances each specific hashtag has been mentioned as the first hashtag in a general or reply tweet. This stage used up 25KB of memory and returns 110 documents for the next stage.

Finally, the sort and limit stage are again executed as one stage, using up approx. 3.2KB of memory. Again, due to the use of limit within this stage, no global sorting is required, since only 5 documents are needed to be returned.

<u>Query 4</u>

This query workload requires the return of the top five users profile with the highest followers count, whom are embedded as a user mention within a tweet that contains a particular hashtag to be provided as a parameter. An alternative implementation will also be considered and compared against the original implementation to be described below.

The first stage within the aggregation pipeline is a match stage. This ensures that only documents with the target hash_tag are passed through the subsequent stages. The advantages of having a partial index on hash_tag.text can also be re-iterated, since in this query, we are only interested in obtaining tweets containing a particular hashtag, hence tweets which do not contain a hashtag do not need to be considered, and hence partial index with a filter on that field existing is effective. It is also worthwhile noting that without the index, a complete collection scan would have been performed on the 10,000 documents searching for the hashtag. Instead, the index allows us to retrieve the documents containing the required key, 49 documents being examined and 49 being passed to the next stage. This stage also includes an implicit projection, keeping only the user_id value of the tweets, since the remaining tweets are not required for the subsequent stage.

An unwind stage then follows, to flatten the document considering that each tweet may contain multiple user mentions. This prepares the data structure for the group stage which follows. The number of documents returned by this stage is 37, suggesting that there are a number of tweets which do not contain a user mention. The group stage which follows does the grouping by the ID of the user mention, allowing us to get the unique list of users mentioned. This stage takes approximately 4KB of memory.

At this stage, we now have each unique user_id that was mentioned on a tweet containing the target hashtag. The lookup stage then makes use of the _id index of the users collection to extract the fields we are interested within the users profile, namely: followers_count, location, and name. Since _id field is indexed by default within MongoDB, the lookup against the users collection is efficient, resulting in just 12 documents being examined, and returning 22 documents in total.

A sort and limit stage then follows the lookup, which again is treated as a combined stage, and uses approximately 5.5KB of memory. A replaceRoot stage follows thereafter, where we move the details captured within the profile array field, to the root of the document being returned.

As previously mentioned, an alternative implementation of the query is also created, and will be used to compare against the original implementation. The alternative implementation has the first three stages being identical as the original, yielding similar results within the execution plan. Except in this instance, within the lookup stage, we only perform the projection on the followers_count, and explicitly exclude the _id field, since we only want to embed the followers count within the new "user_follower_count" field. This is done in a bid to reduce the amount of data being passed through, unlike the original implementation, whereby we include the name and location fields for users whom may not even be on the top five in terms of followers count. With this approach, because only the followers_count field is included, we are then able to perform a sort and limit on the top five users by followers_count, and only after that will we fill the required profile details of name and location by doing another lookup stage.

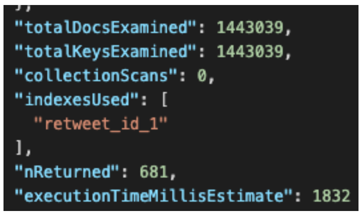
With this alternative implementation, the sort stage took roughly 3.8KB of memory, in comparison with the original query which included data on other profile details, and using up approx. 4.4KB of memory. This improvement in memory usage, however, is marginal, and in this instance can be considered negligible. However, if there are any further changes in requirements, such as the inclusion of a new "description" field within users profiles which could be of significant size, then it is possible that the sort stage in the original implementation would take considerably more memory, since those profile fields will be included during the sorting process. In such scenario, then it may be ideal to switch implementation to this alternative query which only fills the users profile details after obtaining which users are the top five by follower count.

Query 5

In this query, the intended output is the number of general tweets published by users with neither location nor description information in their profiles. In considering the execution for this query, we decide to make two partial indexes, one for the description field, and the other for location field in the users collection. This will be used to check which one will be in the winning plan for the match stage within the aggregation query, and hence avoid a full collection scan. It is also important to note that the partial index that we will create is specifically looking for a blank screen within those fields, since we know that for this dataset, the profiles without profile or location default with a blank string, instead of a null value, and that these fields are present for all profile documents.

The winning plan for the match stage is an index scan on the description index. There are two rejected plans, one of which is the index scan on location, as well as the intersect of the two indexes. The winning plan in this stage is determined as the one that was able to return the first 101 documents, while the execution plan by location index was only able to return 40 during that same time, and the intersect plan returning 27 documents.

After the match stage is the lookup. The intention of this stage is to join the users who have no location and description in their profile, as from the previous match stage, and putting in the general tweets that they've made. This means that the lookup will be on the _id field of the users collection against the user_id field of the tweets collection. It is important to note that the creation of a user_id index on the tweets collection is paramount for this query to run efficiently, since this would otherwise result in the lookup using the retweet_id index, which will result in every user being checked if they are the creator of every single non-retweeted tweet. This results in 1,443,039 documents being examined which equates to 1,443,039 / 681 or 2,119 documents in the tweets collection being checked per user – where 681 is the number of users from the previous stage. With a user_id index being created on the tweets collection, we will instead only have 941 documents being examined in total for this lookup stage, which would be the tweets that those 681 users made.


```
}],
"totalDocsExamined": 1443039,
"totalKeysExamined": 1443039,
"collectionScans": 0,
"indexesUsed": [
  "retweet_id_1"
],
"nReturned": 681,
"executionTimeMillisEstimate": 1832
```
Figure 3 Inefficient lookup stage without user_id index on tweets collection

The project stage following the lookup then creates a new field which aggregates the number of general tweets per user. After which, a grouping is performed on the entire collection, denoted by the use of "_id: null", which allows us to perform a sum on the number of tweets. This consumes less than 1KB of memory. The final stage is then the project, which allows us to format the final output, as specified within the sample output, specifically, removing the _id field.

Query 6

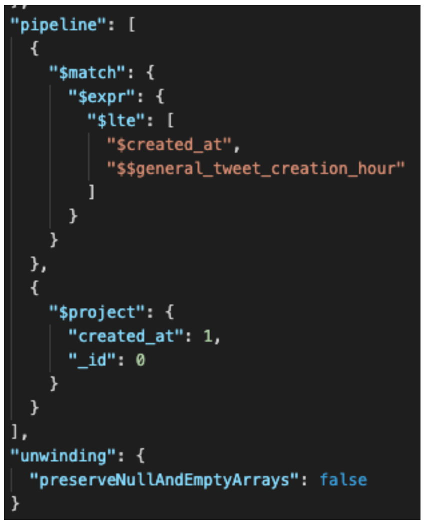
The purpose of this query is to find the general tweet that receives the most number of retweets within the first hour of it being published. The expected outcome is therefore the tweet ID and the number of retweets counted. As in Query 4, an alternative implementation will be provided, as comparison against the original implementation, and to find the approach that is more efficient for the requirement.

In the match stage of the query, the purpose is to find the tweets which are neither retweets nor reply tweets i.e. general tweets. The winning plan is the use of index scan on the retweet_id index with an implicit projection of _id and created_at fields, as denoted by transformBy, resulting in 1021 tweets being returned from this stage. The rejected plans are replyto_id index scan and an intersection between replyto_id and retweet_id indexes, similar to previous query workloads.

We then have a lookup stage, in which we try to find the retweet of each general tweet which have been done within the hour of the parent tweet being created, as marked in the created_at field. This involves using the created_at field and adding 60*60*10000 (which are in milliseconds) to ensure that the retweets that we are matching with in the retweets array field, are within 60 minutes of the parent tweet being created. A new index on the created_at field will be utilised in this query as this allows MongoDB to quickly filter through the retweets that have been created within the 60 minutes timeframe. Without the index, the number of documents examined is 7042, compared to using the created_at index, which reduces the number of documents examined to just 6,680. As a result of also having an index on the created_at index, the lookup stage actually makes use of two indexes: retweet_id index used to find the retweets for every parent tweet from the previous stage, and created_at index, used to filter through the retweets ensuring its' within an hour.

Project stage is then used to capture the count of retweets on each parent tweet from the previous stage, which also returns 1021 documents. Finally, Sort and limit stage thereafter follows, with a value of only 1, since only the parent tweet with the most retweets is required to be displayed. The memory usage is therefore under 1KB for this final stage.

In the alternative implementation of the query, we will be focusing our analysis on the use of the unwind and group stage combination, instead of using a single project stage to aggregate the number of retweets per parent tweet, used in the original implementation. The previous stages are the same, with a match stage to figure out which ones re the parent tweets, with the retweet_id index being the winning query plan. The lookup stage is also the same, except in this instance, the unwind stage has been bundled in with this stage, meaning that the results are already separated from the retweets. 6,680 documents have been returned here in contrast to 1,021 when we just used project stage to get the size of the array.



```
"pipeline": [
  {
    "$match": {
      "$expr": {
        "$lte": [
          "$created_at",
          "$$general_tweet_creation_hour"
        ]
      }
    }
  },
  {
    "$project": {
      "created_at": 1,
      "_id": 0
    }
  }
],
"unwinding": {
  "preserveNullAndEmptyArrays": false
}
```

Figure 4 Unwind stage being bundled in with the lookup stage

This then allows the group stage within our alternative implementation to group by the parent tweet in the form of _id, and perform a count based on the number of rows matching each parent tweet. It is worth noting that because the input of this stage is 6,680 documents, much larger than the output of the lookup stage from the original implementation, the memory usage of this group stage is quite sizeable using 37KB. The increase in number of documents returned from the lookup is due to the unwinding. As before, the sort and limit is then used to capture the parent tweet with the highest number of retweets.

The key takeaway in the analysis of this query is that the use of a single project stage to perform the sum of an array field, is much more efficient than performing an unwind to flatten the array field, and doing the sum on the group stage, since the group stage will consume considerable memory from the flattened document created during the unwind.

Conclusion

Through the analysis on the execution of various query workloads examined within this report, there are various key findings that have been found which support the idea of optimal query implementations.

The most important of which is the use of indexes on fields which are used within lookup stages. Best demonstrated in Queries 1 & 5, without an index on the foreignField, a collection scan will likely be performed resulting in the number of documents being examined to be a multiple of the size of the foreign collection.

The difference between the use of $exists: true and $ne: null are also observed, particularly in Query 2, whereby an index on a field which may not exist for some documents, results in poor performance when matching by $exists: true, whereby, using $ne: null results in a much more efficient use of the index. This supports the notion that fields which do not exists in all documents are treated to have null value within the index, as such checking for exists true merely returns all documents in the collection.

Careful consideration must also be made when choosing between the creation of a partial index or general index. Query 3 exemplifies that creating an index on hash_tags.text is only beneficial if we make it a partial filter on the documents in which the field exists. This can result in considerable space savings.

There are also optimisation approaches which may seem counter-intuitive, such as the use of the two user lookups instead of one in Query 4. In that query workload, we see that it may be worthwhile limiting the amount of data being passed through the aggregation stage, by using projection to limit to just the followers_count within the first lookup, and only after the sort and limit stage do we populate the name and location fields as done in the alternative implementation. This makes for a marginal reduction in memory usage during the sort & limit stage when compared to the original implementation, although as previously discussed, it may be sufficient to simply use the simpler approach done in the original implementation, of loading the details required from the first lookup.

Finally, the use of a single project stage to perform calculations on array fields is confirmed to be more efficient than doing an unwind and a group stage. This is because of the size of the document produced by the unwinding, which is then passed onto the group stage, resulting in greater consumption of memory.