COMP5338 -ASSIGNMENT-1

Name: Pooja Vijay Mahajan
SID: 510282930

INTRODUCTION:

In this assignment, the Twitter datasets comprising of 'tweets' and 'user' objects have been explored. A detailed analysis has been done to gauge the performance of each query along with their alternative implementations. The next section discusses the motivation behind usage of index. Then after studying the different features of the data, this report entails the execution statistics to compare the implementations of different workloads.

For each target query, the following key points are highlighted:

1. Performance analysis using execution statistics
2. Detailed evaluation of each stage with a focus on usage of index and memory
3. An overview of documents used in different stages

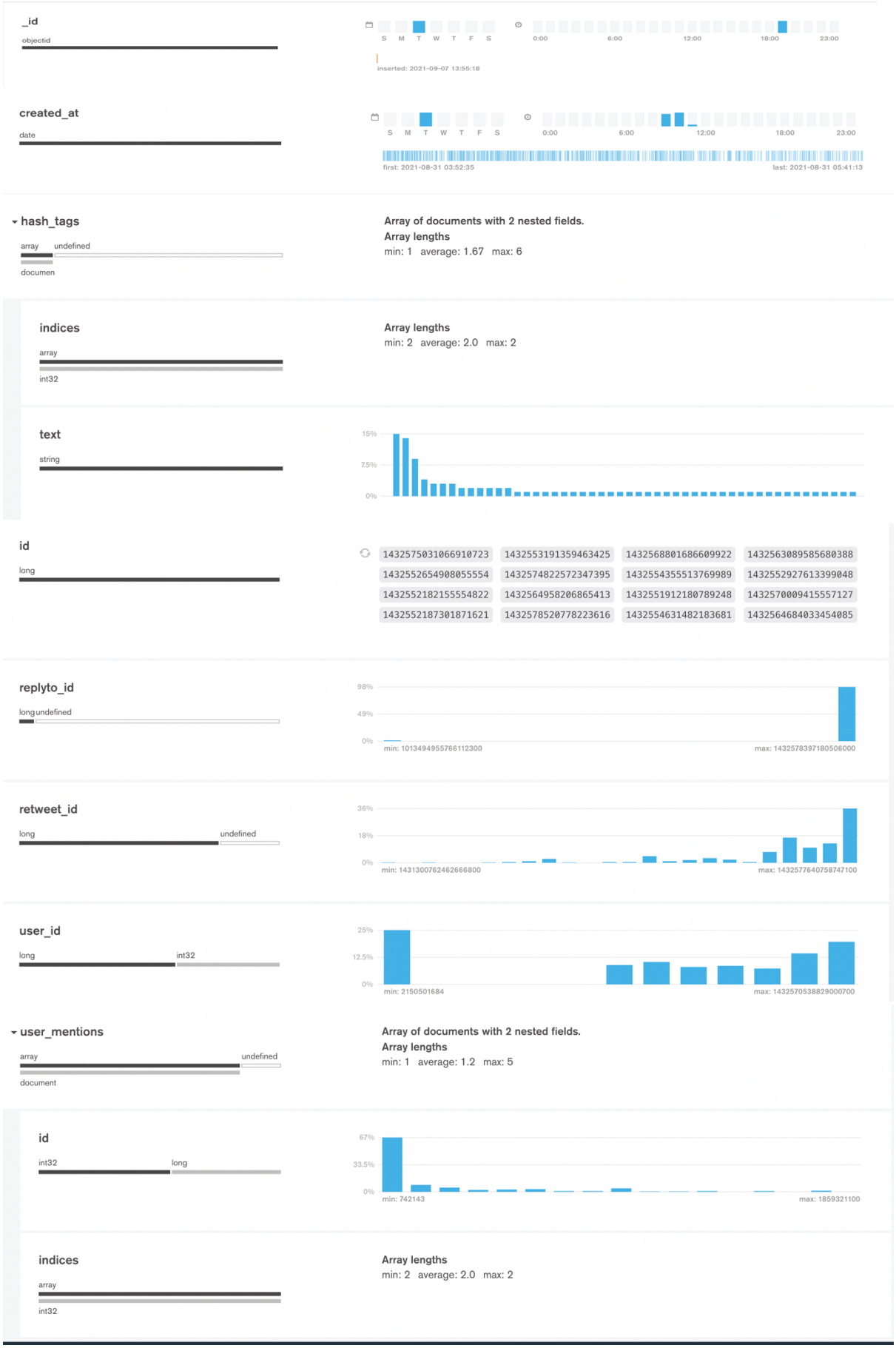
All the inferences have been backed up by relevant screenshots collected from MongoDB's explain( ) method output.

MOTIVATION BEHIND USAGE OF INDEX

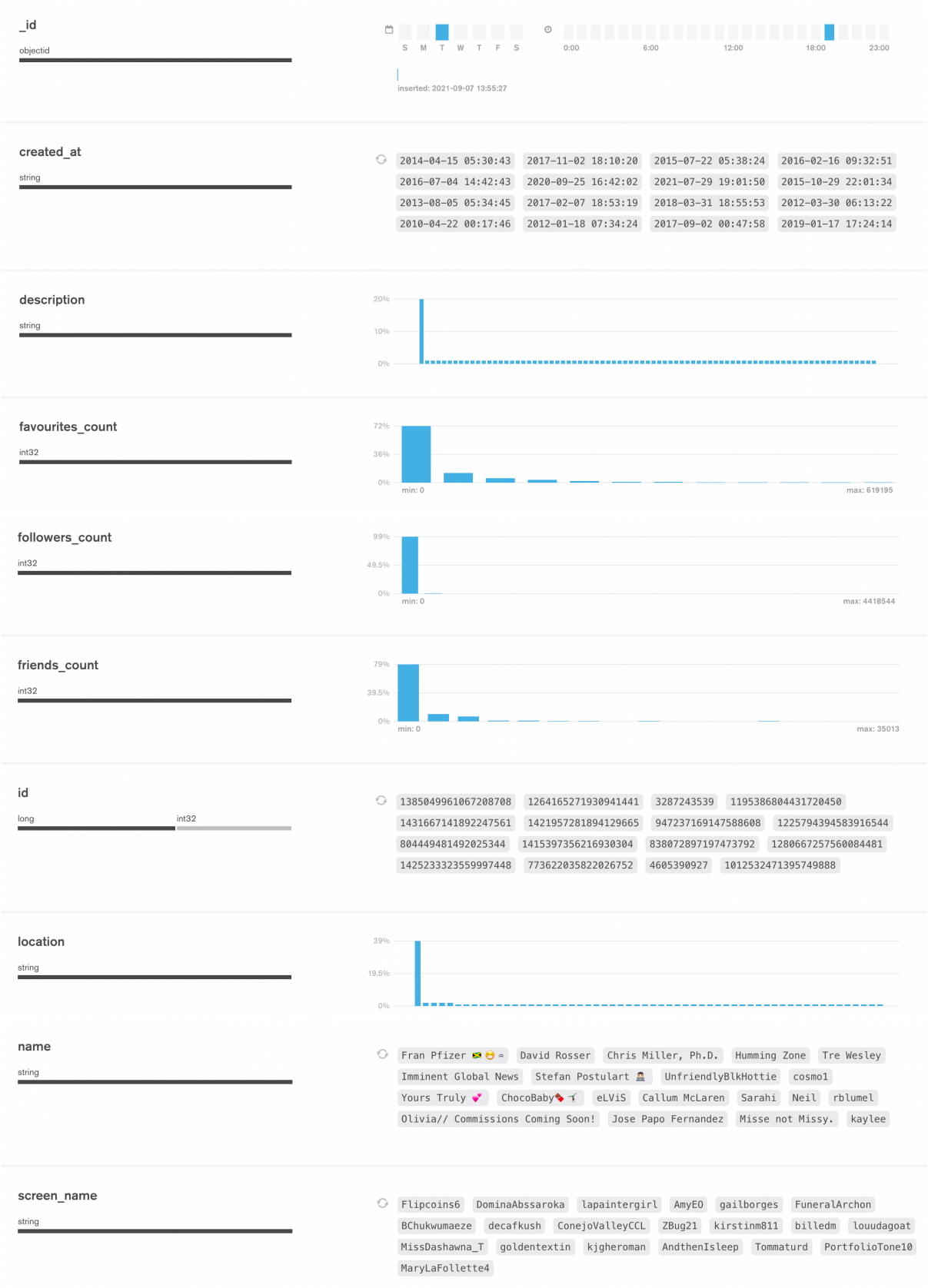
What happens without an index?
When a collection or table is not using an index, the database will have to search through every row in the collection to find the rows matching the condition in the query. If the data ever increases in size, then it will certainly increase the execution time.

What happens with an index?
After we create an index for a database, it causes the database to create a data structure (a tree) and store a pointer which is basically a reference to the original information and the indexed column itself. So instead of scanning all the documents, now the database will scan only for records. Indexing results in fast performance because it arranges the data in sorted order.

A brief overview of the schema design (tweets):



A brief overview of the schema design (users):

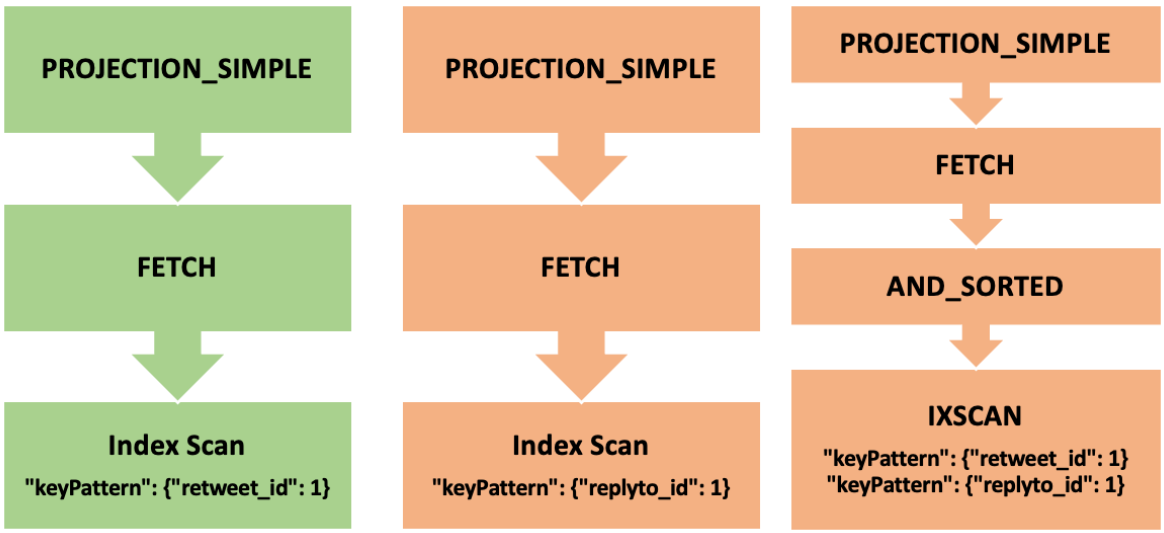


PERFROMANCE ANALYSIS OF QUERY IMPLEMENTATIONS:

Query 1: Find the general tweets with atleast one reply and one retweet.

This query is running against the namespace “a1.tweets_v2”. Fives stages are involved to fetch the required data from the database’s storage engine. The parsedQuery section highlights this “match” condition and passes only those tweets which neither have a replyto_id nor a retweet_id.

This query has 3 possible query plans. Out of which one is the winning plan and the other two are part of the rejected plans. Based on the findings of the query planner (including strings “queryHash: 149552B0” and “ planCacheKey: 13BBFF09”), a cached entry plan is created which comprises of the winning plan. The query optimizer will execute this cache query to generate the result documents.



WINNING PLAN REJECTED PLAN REJECETD PLAN

FIGURE 1: Query plans

Interpreting the winning plan:

The query stage involves different type of operations used to formulate the winning plan which will be discussed in the next section. It is a tree like structure which contains multiple parent and child stages. It uses the index as shown in “keyPattern”: {“retweet_id”:1}

queryPlanner.winningPlan.stage = "PROJECTION_SIMPLE""

It selects the data attribute 'id' and returns 1641 documents. It takes 8ms because it is using an index.

queryPlanner.winningPlan.stage = "FETCH""

These ids are then used in the 'FETCH' stage to retrieve the documents filtering only those which comply with the 'match' condition of being a general tweet. It matches 2262 documents and returns 1641 documents within 1ms.

queryPlanner.winningPlan.stage = "IXSCAN""

IXSCAN implies that MongoDB doesn't have to iterate over all the documents (collection scan), instead it will use an index to find the documents. It uses retweet_id index to execute the query. It contains the index bounds apart from the information related to the index.

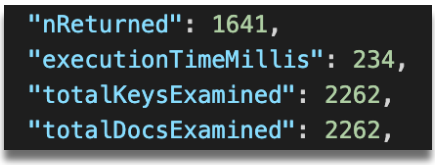
queryPlanner.winningPlan.inputStage.indexName = "retweet_id_1"

The name of this index is "retweet_id_1" where _1 implies that it is used in ascending order. The database reads the single-key index created on retweet_id and passes the documents containing retweet_id to the next stage.

queryPlanner.winningPlan.inputStage.direction = "forward"

It implies that during the execution, the database will traverse the index in a forward direction.

Execution statistics of the winning plan:



executionStats.nReturned = 1641

This is the total number of documents returned that match the query condition, ie. There are 1641 general tweets in the documents.

executionStats.totalKeysExamined = 2262

The index was used, and the total number of keys examined in the index were 2262.

executionStats.totalDocsExamined = 2262

The number of elements in the index (documents) examined during the query execution are 2262.

nReturned /totalKeysExamined = 1641/2262 ~ 72% hit ratio

The number of key examined during the query execution are 2262. For each key examined, 0.72 document was returned.

totalDocsExamined /totalKeysExamined = 2262/2262 ~ 100% hit ratio

The total number of documents examined here are equivalent to the number of elements examined in the index implying that it is an efficient query.

executionStats.executionTimeMillis = 234

It takes approximately 234milliseconds to scan 2262 documents.

Analyzing the stages:

Stage 1: $cursor

No of documents returned :1	Estimated Time of execution :197ms
Total documents examined:2262	Total Keys Examined: 2262

The method explain( ) explains the execution of other commands. It is a helper on shell cursor object. The first stage ($cursor) starts with the query planner executing the parsedQuery to match the tweets which don't have replyto_id and retweet_id. After filtering general tweets, the winning plan is executed with project, fetch and index scan in order. It returns 1641 general tweets (documents) in 19ms.

Stage 2: $facet

No of documents returned :1	Estimated Time of execution :197ms
Total documents examined:892	Total Keys Examined: 892
Index used	
retweet_id_1	replyto_id_1
Memory used	
totalOutputDataSizeBytes=4157	totalOutputDataSizeBytes=573

In the next stage ($facet), 2 aggregation pipelines are created using lookup namely 'forReply' and 'forRetweet'. One is for joining documents for id=retweet_id as retweetData and the other for id=replyto_id as replytoData. This stage returns 1 document in 197ms.

In the forRetweet pipeline, retweet_id_1 index is used. 892 documents are examined in this pipeline by 892 keys which is a single index lookup for each document. 892 documents are returned within 128ms. They are further added to the set based on their 'id', and thus 1 document is returned. It takes a total size of 4157 bytes (4KB) of memory for all the documents as an output by the $group stage.

Similarly, in the forReply pipeline, replyto_id_1 index is used. 21 documents are examined in this pipeline by 21 keys which is a single index lookup for each document. 21 documents are returned within 66ms It takes a total size of 573 bytes of memory for all the documents as an output by the $group stage. The resources consumed in the forReply pipeline are less as compared to forRetweet pipeline because the number of reply tweets are 621 and the number of retweets is 7738.

Stage 3&4: $unwind

No of documents returned :1	Estimated Time of execution :197ms

The resulting documents are added to a set using their ids followed by two stages of unwind. The unwind stage splits the documents such that for each id, the corresponding reply/retweet is added. It takes 197ms to execute this stage.

Stage 5: $project

No of documents returned :1	Estimated Time of execution :197ms

In the project stage, an intersection of both the sets ("forRetweet.setArr", "forReplyto.setArr") are taken using setIntersection to return the number of general tweets having both replies and retweets in 194ms.

Query 2: Find the reply tweet that has the most retweets in the data set.

This query is running against the namespace “a1.tweets_v2”. Fives stages are involved to fetch the required data from the database’s storage engine. The parsedQuery section highlights this “match” condition and passes only reply tweets (those tweets that have a replyto_id)

This query has only 1 possible query plans which is the winning plan. Based on the findings of the query planner (including strings “queryHash: 98BF093B” and “ planCacheKey: FCF9B63F”), a cached entry plan is created which comprises of the winning plan. The query optimizer will execute this cache query to generate the result documents.

Interpreting the winning plan:

The query stage involves different type of operations used to formulate the winning plan which will be discussed in the next section. It is a tree like structure which contains multiple parent and child stages. It uses the index as shown in “keyPattern”: {“replyto_id”:1}as well as “keyPattern”: {“retweet_id”:1}

```
"executionStages": {
  "stage": "PROJECTION_SIMPLE",
  "nReturned": 621,
  "executionTimeMillisEstimate": 0,
  "works": 10001,
  "advanced": 621,
  "needTime": 9379,
  "needYield": 0,
  "saveState": 11,
  "restoreState": 11,
  "isEOF": 1,
  "transformBy": {
    "_id": 1,
    "id": 1,
    "retweetData": 1,
    "retweet_count": 1,
    "text": 1
  },
```

```
"inputStage": {
  "stage": "FETCH",
  "filter": {
    "replyto_id": {
      "$exists": true
    }
  },
  "nReturned": 621,
  "executionTimeMillisEstimate": 0,
  "works": 10001,
  "advanced": 621,
  "needTime": 9379,
  "needYield": 0,
  "saveState": 11,
  "restoreState": 11,
  "isEOF": 1,
  "docsExamined": 10000,
  "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
  },
```

queryPlanner.winningPlan.stage = “PROJECTION_SIMPLE”"
It selects the data attribute ‘_id’,’id’,retweetData and retweet_count and returns 621 documents. It takes 0ms to execute this part.

queryPlanner.winningPlan.stage = “FETCH”"

These ids are then used in the 'FETCH' stage to retrieve the documents filtering only those which comply with the 'match' condition of being a reply tweet. It matches 10000 documents and returns 621 documents within 1ms.

queryPlanner.winningPlan.stage = "IXSCAN""
It uses replyto_id_1 index to execute the query. It contains the index bounds apart from the information related to the index.

queryPlanner.winningPlan.inputStage.indexName = "replyto_id_1"
The name of this index is "replyto_id_1" where _1 implies that it is used in ascending order. The database reads the single-key index created on replyto_id and passes the documents containing replyto_id to the next stage.

queryPlanner.winningPlan.inputStage.direction = "forward"
It implies that during the execution, the database will traverse the index in a forward direction.

Execution statistics of the winning plan:


```
"executionStats": {
  "executionSuccess": true,
  "nReturned": 621,
  "executionTimeMillis": 85,
  "totalKeysExamined": 10000,
  "totalDocsExamined": 10000,
```


executionStats.nReturned = 621
This is the total number of documents returned that match the query condition, ie. There are 621 reply tweets in the documents.

executionStats.totalKeysExamined = 10000
The index was used, and the total number of keys examined in the index were 10000. These include the index entries of the reply tweets.

executionStats.totalDocsExamined = 10000
The number of elements in the index (documents) examined during the query execution are 2262.

nReturned /totalKeysExamined = 621/10000 ~ 6.21% hit ratio
The number of key examined during the query execution are 2262. For each key examined, 0.06 document was returned.

totalDocsExamined /totalKeysExamined = 2262/2262 ~ 100% hit ratio
The total number of documents examined here are equivalent to the number of elements examined in the index implying that it is an efficient query.

executionStats.executionTimeMillis = 85

It takes approximately 85 milliseconds to scan 10000 documents.

Analyzing the stages:

Stage 1: $cursor

No of documents returned :621	Estimated Time of execution :38ms
Total documents examined:10000	Total Keys Examined: 10000

The first stage ($cursor) starts with the query planner executing the parsedQuery to match the tweets which have replyto_id. After filtering reply tweets, the winning plan is executed with project, fetch and index scan in order. It returns 621 reply tweets (documents) in 38ms. It uses the index on replyto_id to execute this part.

Stage 2: $lookup

No of documents returned :621	Estimated Time of execution :83ms
Total documents examined:27	Total Keys Examined: 27
Index used: retweet_id_1	

In the next stage ($lookup), all the documents containing retweet_id are associated with their corresponding ‘id’ using the index retweet_id_1. 27 documents were examined using

27 index keys ie. single index lookup for each document. This stage returns 621 documents in 38ms as retweetData.

Stage 3: $addField

No of documents returned :621	Estimated Time of execution :83ms

The size of the array (retweetData) was stored in a variable called retweet_count. It returned 621 documents within 83ms.

Stage 4: $sort

No of documents returned :1	Estimated Time of execution :83ms
Memory consumed (totalDataSizeSortedBytesEstimate) ~0 bytes	

In the sort stage, the documents in retweetData array were arranged in a descending order on the basis of retweet_count and a limit was applied for 1 document, thus it returned 1 document within 83ms with almost 0 bytes of memory consumption (assuming we have small sample data)

Stage 5: $project

No of documents returned :1	Estimated Time of execution :83ms

In the project stage, 1 document was returned within 83ms. It contained the id of the reply tweet and its retweet count.

Query 3: Find the top 5 hashtags appearing as the FIRST hashtag in a general or reply tweet, ignoring the case of the hashtag.

This query is running against the namespace “a1.tweets_v2”. Four stages are involved to fetch the required data from the database’s storage engine. The parsedQuery section highlights this “match” condition and passes only those general tweets and retweets which contains hash_tags.

This query has only 1 possible query plans which is the winning plan. Based on the findings of the query planner (including strings “queryHash: DA30D830” and “ planCacheKey: 2E6AFC44”), a cached entry plan is created which comprises of the winning plan. The query optimizer will execute this cache query to generate the result documents.

Interpreting the winning plan:

The query stage involves different type of operations used to formulate the winning plan which will be discussed in the next section. It is a tree like structure which contains multiple parent and child stages. It uses the index as shown in “keyPattern”: {“retweet_id”:1}

queryPlanner.winningPlan.stage = “PROJECTION_DEFAULT”

It selects the data attribute ‘firstHashTag.text’ and ‘hash_tags’ and returns 211 documents. It takes 0ms to execute this part.

queryPlanner.winningPlan.stage = “FETCH”

A filter is applied in this stage to match the condition of not being a retweet (ie. the document should either be a general tweet or reply tweet) and those documents containing hash_tags. It matches 2262 documents and returns 211 documents within 0ms.

queryPlanner.winningPlan.stage = “IXSCAN”

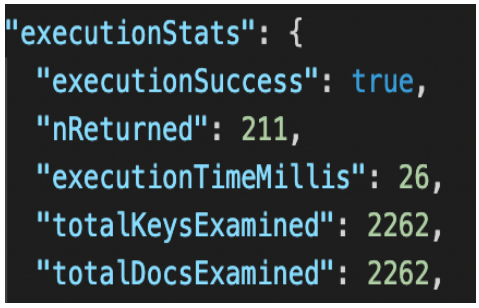
It uses retweet_id_1 index to execute the query. It contains the index bounds apart from the information related to the index. It matches 2262 documents and returns 2262 documents within 0ms.

queryPlanner.winningPlan.inputStage.indexName = “retweet_id_1”

The name of this index is “retweet_id_1” where _1 implies that it is used in ascending order. The database reads the single-key index created on retweet_id and passes the documents containing retweet_id to the next stage.

queryPlanner.winningPlan.inputStage.direction = "forward"
It implies that during the execution, the database will traverse the index in a forward direction.

Execution statistics of the winning plan:



executionStats.nReturned = 211
This is the total number of documents returned that match the query condition, ie. There are 621 general tweets and retweets which contain hashtags in the documents.

executionStats.totalKeysExamined = 2262
The index was used, and the total number of keys examined in the index were 2262. These include the index entries of the retweets.

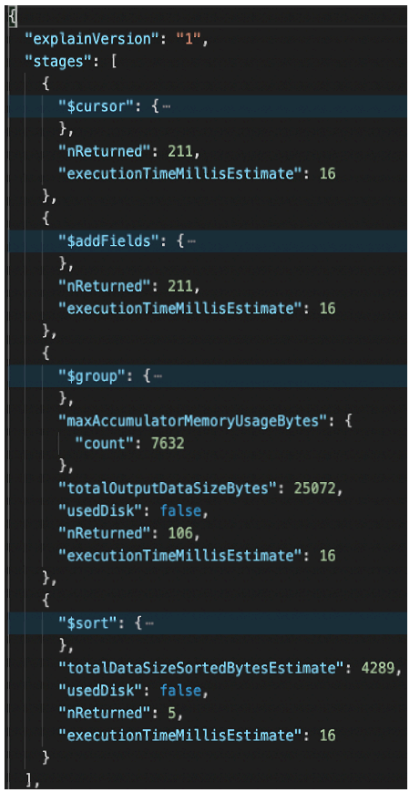
executionStats.totalDocsExamined = 2262
The number of elements in the index (documents) examined during the query execution are 2262.

nReturned /totalKeysExamined = 211/2262 ~ 9.32% hit ratio
The number of key examined during the query execution are 2262. For each key examined, 0.09 document was returned.

totalDocsExamined /totalKeysExamined = 2262/2262 ~ 100% hit ratio
The total number of documents examined here are equivalent to the number of elements examined in the index implying that it is an efficient query.

executionStats.executionTimeMillis = 26
It takes approximately 26 milliseconds to scan 2262 documents.

Analyzing the stages:



Stage 1: $cursor

No of documents returned :211	Estimated Time of execution :16ms
Total documents examined:2262	Total Keys Examined: 2262

The first stage ($cursor) starts with the query planner executing the parsedQuery to match the tweets which don't have retweet_id and have hash_tags. After filtering general tweets and reply tweets, the winning plan is executed with project, fetch and index scan in order. It returns 211 documents in 16ms. It uses the index on retweet_id to execute this part.

Stage 2: $addFields

No of documents returned :211	Estimated Time of execution :16ms

In the next stage ($addFields), the first hash_tags are extracted and added in the array firstHashTag. This stage returns 211 documents in 16ms as firstHashTag.

Stage 3: $group

No of documents returned :106	Estimated Time of execution :16ms
Memory consumed (totalDataSizeSortedBytesEstimate) ~ 25072 bytes	

In the group stage, the texts of firstHashTag is converted into lower case and their count is added up. It consumed 25072 bytes (25KB) of memory. It returned 106 documents within 16ms.

Stage 4: $sort

No of documents returned :5	Estimated Time of execution : 16ms
Memory consumed (totalDataSizeSortedBytesEstimate) ~4289 bytes	

In the sort stage, the documents in firstHashTag array were arranged in a descending order by count with a limit of 5. Thus, it returned 5 documents within 16ms with 4289 bytes (4.2KB) of memory consumption.

Query 4: For a given hash_tag, there are many tweets including that hash_tag. Some of those tweets mention one or many users. Among all users mentioned in those tweets, find the top 5 users with the most followers_count.

This query is running against the namespace “a1.tweets_v2”. Fives stages are involved to fetch the required data from the database’s storage engine. The parsedQuery section highlights this “match” condition and passes only those tweets which neither have a replyto_id nor a retweet_id.

This query has 1 possible query plans which is the winning plan. Based on the findings of the query planner (including strings “queryHash: 7815A034” and “ planCacheKey: 923AB29A”), a cached entry plan is created which comprises of the winning plan. The query optimizer will execute this cache query to generate the result documents.

Interpreting the winning plan:

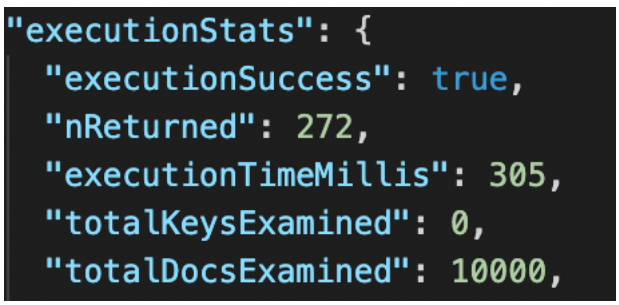
The query stage involves different type of operations used to formulate the winning plan which will be discussed in the next section. It is a tree like structure which contains multiple parent and child stages. Since the second stage is COLLSCAN, it implies that it does not use any index and is iterating through all the documents in the database.

queryPlanner.winningPlan.stage = "PROJECTION_SIMPLE""
It selects the data attribute 'id', 'created_at', 'retweetData.total' to be projected and returns 272 documents. It takes 3ms to execute this query.

queryPlanner.winningPlan.stage = "COLLSCAN""
It implies that it is not using any index and iterates through all the documents in the database. It returns 272 documents. It takes 3ms to execute this query.

queryPlanner.winningPlan.inputStage.direction = "forward"
It implies that during the execution, the database will traverse in a forward direction.

Execution statistics of the winning plan:



executionStats.nReturned = 272

This is the total number of documents returned that match the query condition, ie. There are 272 tweets in the documents.

executionStats.totalKeysExamined =0

Since the index was not used, the total number of keys examined in the index were 0.

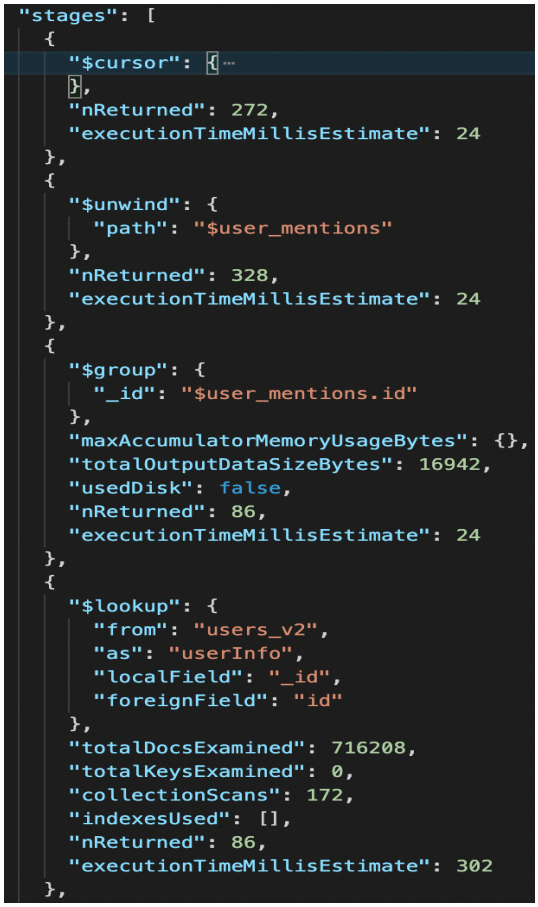
executionStats.totalDocsExamined = 10000

Since the index was not used, all the 10000 documents were examined.

executionStats.executionTimeMillis = 305

It takes approximately 305 milliseconds(0.3 seconds) to scan 2262 documents.

Analyzing the stages:



Stage 1: $cursor

No of documents returned :272	Estimated Time of execution :24ms
Total documents examined:10000	

The method explain() explains the execution of other commands. It is a helper on shell cursor object. It returns 272 tweets (documents) in 24ms.

Stage 2: $unwind

No of documents returned :328	Estimated Time of execution :24ms

In the next stage all the documents were unwinded. This stage returns 328 documents in 24ms.

Stage 3: $group

No of documents returned :86	Estimated Time of execution : 24ms
Memory consumed (totalDataSizeSortedBytesEstimate) ~16942 bytes	

In the group stage, returned 86 documents within 24ms with almost 16942 bytes(16.9KB) of memory consumption.

Stage 4: $lookup

No of documents returned :86	Estimated Time of execution :302ms
Documents examined: 716208	

In the lookup stage, 71608 documents were returned within 302ms.

Stage 5: $sort

No of documents returned :5	Estimated Time of execution :302ms
Memory consumed (totalDataSizeSortedBytesEstimate) ~30398 bytes	

In the lookup stage, 5 documents were returned within 302ms consuming 30398 bytes(30KB).

Stage 6: $unwind

No of documents returned :5	Estimated Time of execution :302ms

In the next stage all the documents were unwinded. This stage returns 5 documents in 24ms.

Stage 7: $project

No of documents returned :5	Estimated Time of execution :302ms

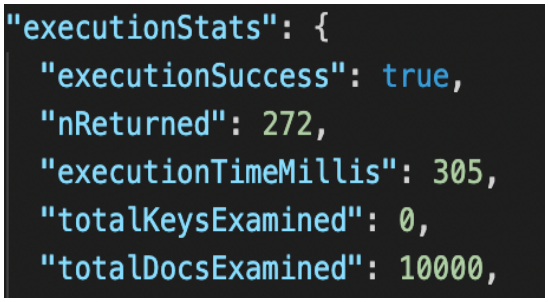
In the next stage all the documents were projected. This stage returns 5 documents in 302ms.

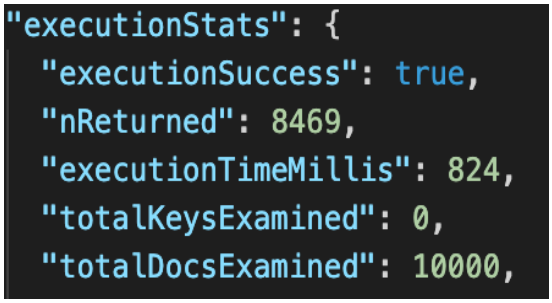
Stage 8: $limit

No of documents returned :5	Estimated Time of execution :302ms

This stage returns 5 documents in 202ms.

COMPARISON OF QUERY 4 WITH IT'S ALTERNATIVE IMPLEMENTATION





Query 4 (FIRST IMPLEMENTATION) Query 4 (SECOND IMPLEMENTATION)

A few key differences that can be traced between both the implementations of query 4:

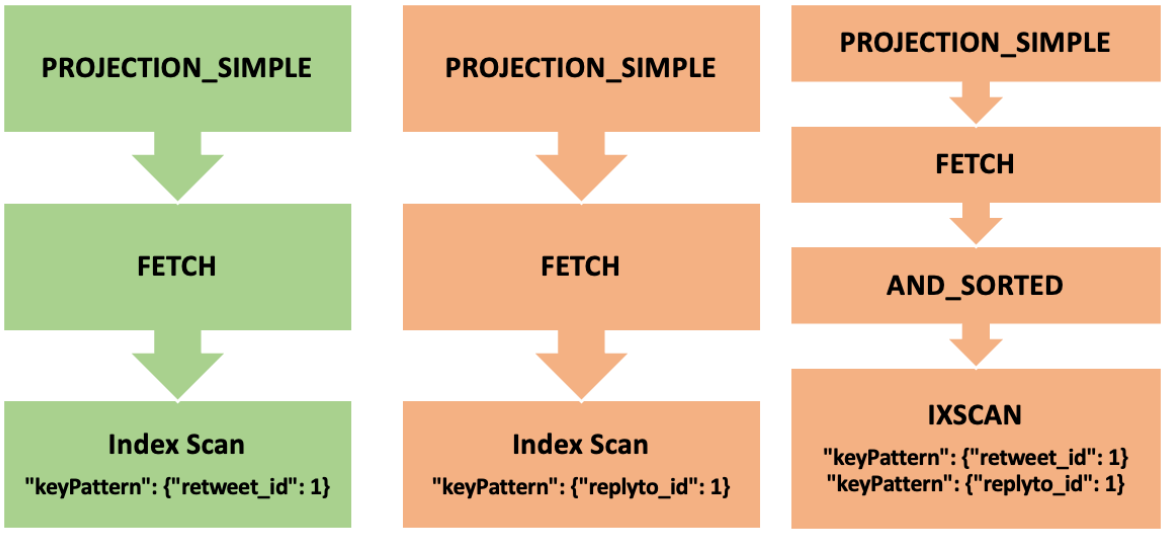
Query 4 (FIRST IMPLEMENTATION)	Query 4 (SECOND IMPLEMENTATION)
No of stages : 8	No of stages : 12
Execution Time : 305ms	Execution Time : 825ms

1. The first implementation is better than the second implementation because it is in best practices to use $match before $lookup. $lookup is an expensive operation and must be used towards the end of the aggregation pipeline if possible. In the first stage we are filtering the documents as per the criteria and then using lookup on the remaining documents. This is less expensive in terms of "memory used" and "time consumed" than the second implementation.
2. The number of stages in the pipeline should be as less as possible to improve the efficiency of the query. First implementation has more stages than second implementation which makes the former a better alternative.
3. The execution time of the second implementation is twice than that of first implementation.
4. Indexes are not used in both the implementations.

Query 5: Find the number of general tweets published by users with neither location nor description information.

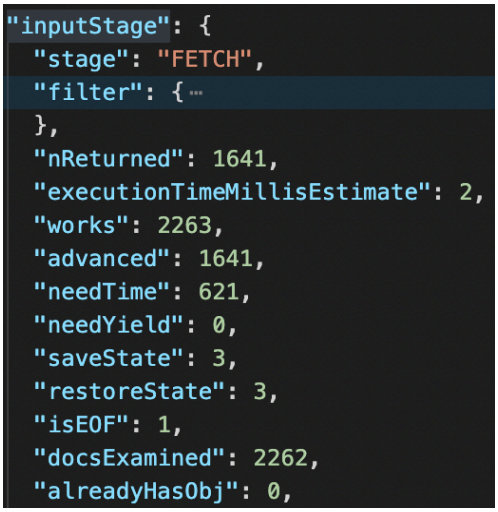
This query is running against the namespace "a1.tweets_v2". Fives stages are involved to fetch the required data from the database's storage engine. The parsedQuery section highlights this "match" condition and passes only those tweets which neither have a replyto_id nor a retweet_id.

This query has 3 possible query plans. Out of which one is the winning plan and the other two are part of the rejected plans. Based on the findings of the query planner (including strings “queryHash: 4C9B65E0” and “ planCacheKey: 2B89823B”), a cached entry plan is created which comprises of the winning plan. The query optimizer will execute this cache query to generate the result documents.



Interpreting the winning plan:

The query stage involves different type of operations used to formulate the winning plan which will be discussed in the next section. It is a tree like structure which contains multiple parent and child stages. It uses the index as shown in “keyPattern”: {“retweet_id”:1}



queryPlanner.winningPlan.stage = “PROJECTION_SIMPLE”"

It selects the data attribute ‘id’ to be projected and returns 1641 documents. It takes 2ms because it is using an index.

queryPlanner.winningPlan.stage = “FETCH”"

These ids are then used in the ‘FETCH’ stage to retrieve the documents filtering only those which comply with the ‘match’ condition of being a general tweet. It matches 2262 documents and returns 1641 documents within 2ms.

queryPlanner.winningPlan.stage = "IXSCAN'"
IXSCAN implies that MongoDB doesn't have to iterate over all the documents (collection scan), instead it will use an index to find the documents. It uses retweet_id index to execute the query. It contains the index bounds apart from the information related to the index.

queryPlanner.winningPlan.inputStage.indexName = "retweet_id_1"
The name of this index is "retweet_id_1" where _1 implies that it is used in ascending order. The database reads the single-key index created on retweet_id and passes the documents containing retweet_id to the next stage.

queryPlanner.winningPlan.inputStage.direction = "forward"
It implies that during the execution, the database will traverse the index in a forward direction.

Execution statistics of the winning plan:


```
"executionStats": {
  "executionSuccess": true,
  "nReturned": 1641,
  "executionTimeMillis": 5764,
  "totalKeysExamined": 2262,
  "totalDocsExamined": 2262,
```


executionStats.nReturned = 1641
This is the total number of documents returned that match the query condition, ie. There are 1641 general tweets in the documents.

executionStats.totalKeysExamined = 2262
The index was used, and the total number of keys examined in the index were 2262.

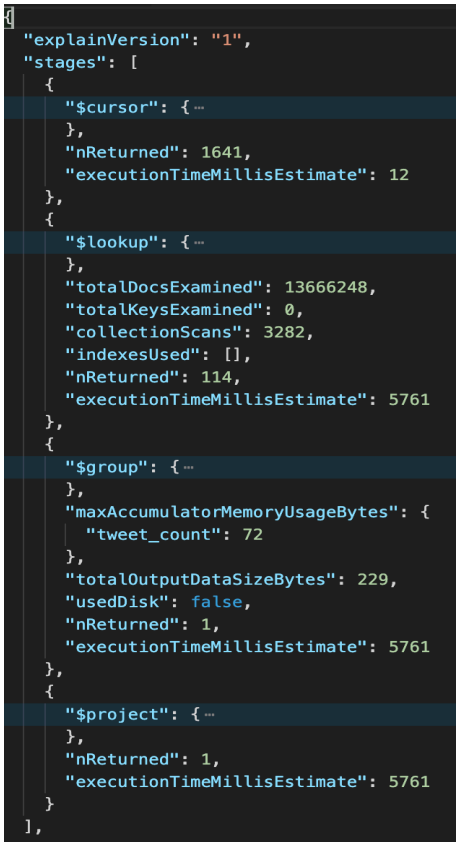
executionStats.totalDocsExamined = 2262
The number of elements in the index (documents) examined during the query execution are 2262.

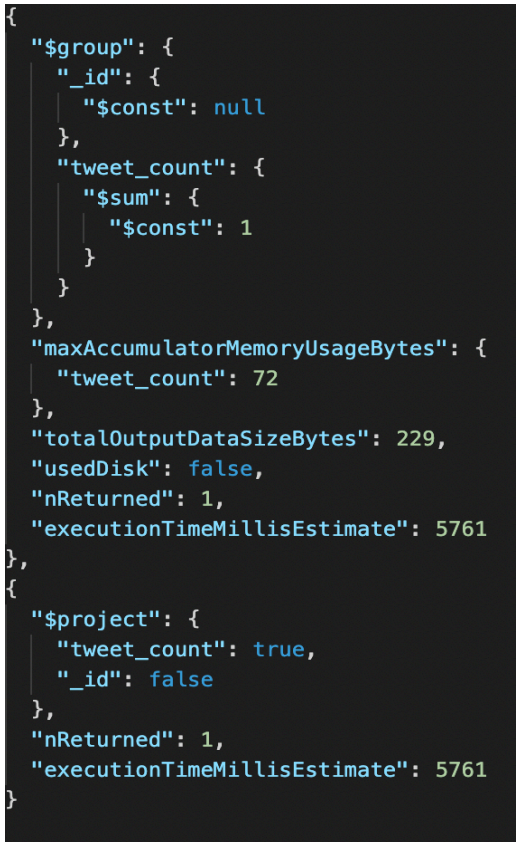
nReturned /totalKeysExamined = 1641/2262 ~ 72% hit ratio
The number of key examined during the query execution are 2262. For each key examined, 0.72 document was returned.

totalDocsExamined /totalKeysExamined = 2262/2262 ~ 100% hit ratio
The total number of documents examined here are equivalent to the number of elements examined in the index implying that it is an efficient query.

executionStats.executionTimeMillis = 5764
It takes approximately 5764 milliseconds(5.7 seconds) to scan 2262 documents.

Analyzing the stages:





Stage 1: $cursor

No of documents returned :1641	Estimated Time of execution :12ms
Total documents examined:2262	Total Keys Examined: 2262

The method explain( ) explains the execution of other commands. It is a helper on shell cursor object. The first stage ($cursor) starts with the query planner executing the parsedQuery to match the tweets which don't have replyto_id and retweet_id. After filtering general tweets, the winning plan is executed with project, fetch and index scan in order. It returns 1641 general tweets (documents) in 12ms.

Stage 2: $lookup

No of documents returned :114	Estimated Time of execution :5761ms
Total documents examined: 13666248	Total Keys Examined: 0

In the next stage ($lookup), 1 aggregation pipeline is created namely 'userInfo'. It associates id from users to the corresponding user_id in tweets as userInfo. It then filters those documents in which do not have the location or description followed by unwinding. This stage returns 1 document in 197ms.

Stage 3: $group

No of documents returned :1	Estimated Time of execution :5761ms

Memory ("totalOutputDataSizeBytes": 229)

The resulting documents are then grouped together to calculate the sum of the tweet_count. This stage consumes 229 bytes of memory and 5761ms (5.7ms) to execute.

Stage 5: $project

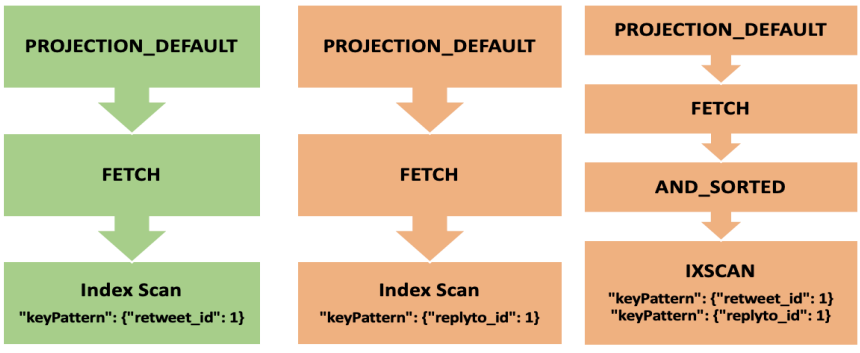
No of documents returned :1	Estimated Time of execution :197ms

In the project stage, the variable tweet_count is projected. It returns 1 document within 197ms of execution.

Query 6: Find the general tweet that receives most retweets in the first hour after it is published.

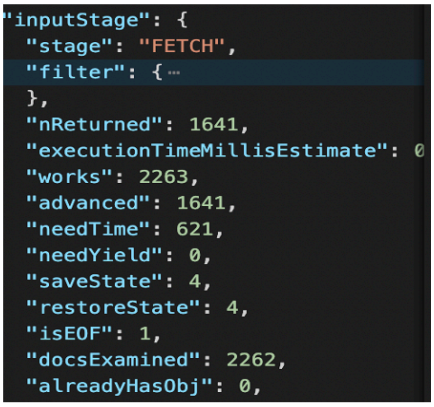
This query is running against the namespace “a1.tweets_v2”. Fives stages are involved to fetch the required data from the database’s storage engine. The parsedQuery section highlights this “match” condition and passes only those tweets which neither have a replyto_id nor a retweet_id.

This query has 3 possible query plans. Out of which one is the winning plan and the other two are part of the rejected plans. Based on the findings of the query planner (including strings “queryHash: F3615BCF” and “ planCacheKey: 43AE9BFC”), a cached entry plan is created which comprises of the winning plan. The query optimizer will execute this cache query to generate the result documents.



Interpreting the winning plan:

The query stage involves different type of operations used to formulate the winning plan which will be discussed in the next section. It is a tree like structure which contains multiple parent and child stages. It uses the index as shown in “keyPattern”: {“retweet_id”:1}



queryPlanner.winningPlan.stage = "PROJECTION_DEFAULT'"
It selects the data attribute 'id', 'created_at', 'retweetData.total' to be projected and returns 1641 documents. It takes 0ms to execute this query.

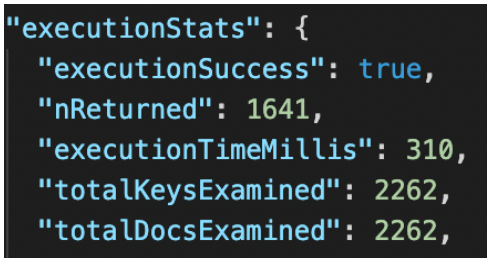
queryPlanner.winningPlan.stage = "FETCH'"
These ids are then used in the 'FETCH' stage to retrieve the documents filtering only those which comply with the 'match' condition of being a general tweet. It matches 2262 documents and returns 1641 documents within 0ms.

queryPlanner.winningPlan.stage = "IXSCAN'"
IXSCAN implies that MongoDB doesn't have to iterate over all the documents (collection scan), instead it will use an index to find the documents. It uses retweet_id index to execute the query. It contains the index bounds apart from the information related to the index.

queryPlanner.winningPlan.inputStage.indexName = "retweet_id_1"
The name of this index is "retweet_id_1" where _1 implies that it is used in ascending order. The database reads the single-key index created on retweet_id and passes the documents containing retweet_id to the next stage.

queryPlanner.winningPlan.inputStage.direction = "forward"
It implies that during the execution, the database will traverse the index in a forward direction.

Execution statistics of the winning plan:



executionStats.nReturned = 1641

This is the total number of documents returned that match the query condition, ie. There are 1641 general tweets in the documents.

executionStats.totalKeysExamined = 2262

The index was used, and the total number of keys examined in the index were 2262.

executionStats.totalDocsExamined = 2262

The number of elements in the index (documents) examined during the query execution are 2262.

nReturned /totalKeysExamined = 1641/2262 ~ 72% hit ratio

The number of key examined during the query execution are 2262. For each key examined, 0.72 document was returned.

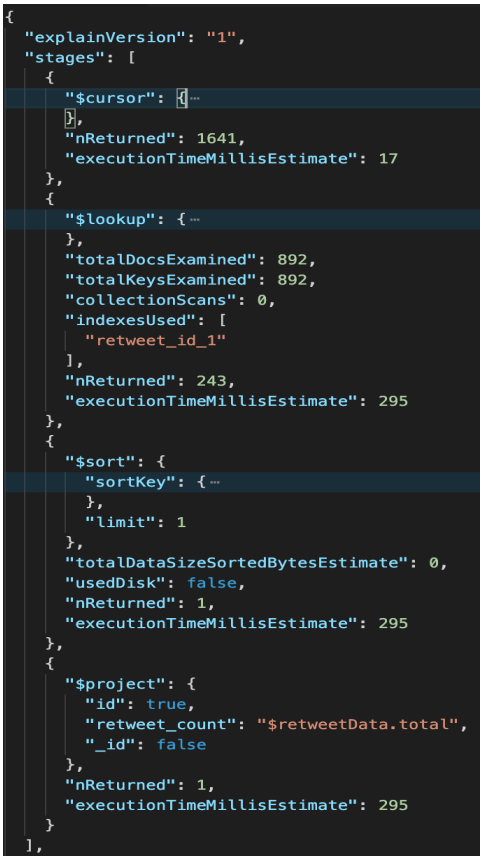
totalDocsExamined /totalKeysExamined = 2262/2262 ~ 100% hit ratio

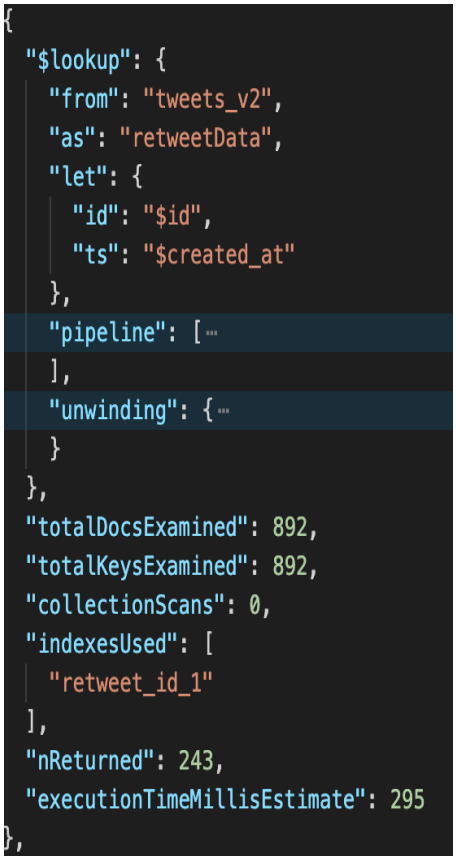
The total number of documents examined here are equivalent to the number of elements examined in the index implying that it is an efficient query.

executionStats.executionTimeMillis = 310

It takes approximately 310 milliseconds(0.3 seconds) to scan 2262 documents.

Analyzing the stages:





Stage 1: $cursor

No of documents returned :1641	Estimated Time of execution :17ms
Total documents examined:2262	Total Keys Examined: 2262

The method explain() explains the execution of other commands. It is a helper on shell cursor object. The first stage ($cursor) starts with the query planner executing the parsedQuery to match the tweets which don't have replyto_id and retweet_id. After filtering general tweets, the winning plan is executed with project, fetch and index scan in order. It returns 1641 general tweets (documents) in 12ms.

Stage 2: $lookup

No of documents returned :243	Estimated Time of execution :295ms
Total documents examined: 892	Total Keys Examined: 892
Index used: retweet_id_1	

In the next stage ($lookup), an array named retweetData was created. It creates a pipeline where it matches if id is equal to retweet_id and the date at which the retweet is created is within one hour of time at which the general tweet was created. This is followed by unwinding. This stage returns 243 documents in 295ms.

Stage 3: $sort

No of documents returned :1	Estimated Time of execution : 295ms
Memory consumed (totalDataSizeSortedBytesEstimate) ~0 bytes	

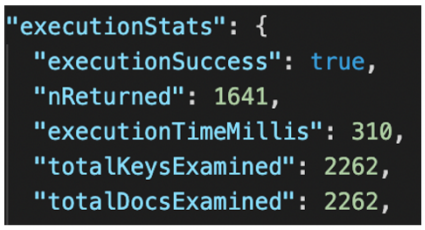
In the sort stage, the documents in retweetData array were arranged in a descending order by count with a limit of 1. Thus, it returned 1 document within 295ms with almost 0 bytes of memory consumption.

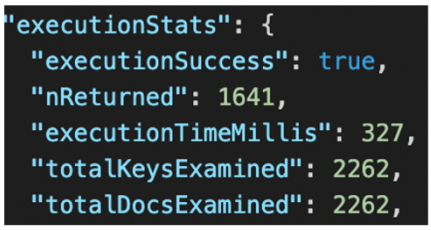
Stage 5: $project

No of documents returned :1	Estimated Time of execution :295ms

In the project stage, the variable retweetData is projected. It returns 1 document within 295ms of execution.

COMPARISON OF QUERY 6 WITH IT'S ALTERNATIVE IMPLEMENTATION





Query 6 (FIRST IMPLEMENTATION) Query 6 (SECOND IMPLEMENTATION)

A few key differences that can be traced between both the implementations of query 6:

Query 6 (FIRST IMPLEMENTATION)	Query 6 (SECOND IMPLEMENTATION)
No of stages : 4	No of stages : 5
Execution Time : 310ms	Execution Time : 327ms
Stages: 1. $match 2. $lookup 3. $unwind 4. $sort 5. $project 6. $limit	Stages: 1. $addField 2. $lookup 3. $match 4. $unwind 5. $sort 6. $project 7. $limit

1. The first implementation is better than the second implementation because it is in best practices to use $match before $lookup. $lookup is an expensive operation and must be used towards the end of the aggregation pipeline if possible. In the first stage we are filtering the documents as per the criteria and then using lookup on the remaining

documents. This is less expensive in terms of “memory used” and “time consumed” than the second implementation.

2. The number of stages in the pipeline should be as less as possible to improve the efficiency of the query. First implementation has more stages than second implementation which makes the former a better alternative.
3. The execution time of the second implementation is more than that of first implementation.
4. Indexes are used in both the implementations as part of the winning plan (retweet_id_1).

CONCLUSION:

The database does several things to try and optimize it, but in terms of execution directly, it is going to basically split into two parts. It's going to have the first part where it goes and retrieves the candidate documents from the underlying collection and its gonna have the second part, which is basically any additional transformations that can be performed in the initial retrieval of the documents. A few things to consider when encountering poor performance are data model and sizing query, query patterns, indexes and hardware configuration.

The bottom-line is that indices will increase the efficiency of the queries by reducing the number of rows during the search. However, it is not encouraged to use indexing when we rarely run the queries in a backup MongoDB database. It is usually useful in those instances when the speed of accessing the data matters.