Stanbic IBTC Bank Plc

MongoDB Consulting Report

Santhosh S Kashyap <santhosh.skashyap@mongodb.com>, MongoDB Inc.
June 2023

Participants:

- Ernest Oduba, Stanbic IBTC Bank Plc
- Nnamdi Eze, Stanbic IBTC Bank Plc
- Michael Okwuwa, Stanbic IBTC Bank Plc
- Santhosh S Kashyap, Consulting Engineer, MongoDB Inc.

This document summarizes the discussions and recommendations from 2 days of “Optimize, Accelerate” remote consultation with “Stanbic IBTC Bank Plc” on 05-June-2023 and 06-June-2023.

Each recommendation is assigned a level from 1 to 3. The levels correspond to the following priorities:

1. Implement this recommendation *immediately*.

    Not implementing this recommendation incurs the risk of data loss, system unavailability, or other significant problems that may cause outages in the future.

2. Implement this recommendation *as soon as possible*.

    These recommendations are less severe than level 1, but typically represent significant problems that could affect a production system. Consider these issues promptly.

3. Consider this recommendation.

    While this suggestion may improve some aspects of the application, it is not critical and may reflect a much larger change in the application code or MongoDB deployment. Consider these modifications as part of the next revision of the application.

1. Background

1.1 Application

Stanbic IBTC is a financial service holding company in Nigeria with subsidiaries in Banking, Stock Brokerage, Investment Advisory, Asset Management, Investor Services, Pension Management, Trustees Insurance Brokerage and life Insurance businesses.

1.2 Environment

Application is developed using C#. Following are the details of the MongoDB deployment focussed as part of this engagement:

Mongodb Deployment

Server	IP Address	Notes
pngmobiledb01v.ng.sbicdirectory.com	10.234.18.142	Deployment (Region 1)
pngmobiledb02v.ng.sbicdirectory.com	10.234.18.143	Deployment (Region 1)
pngarbiter.ng.sbicdirectory.com	10.234.18.141	Deployment (Region 1)
yngmobiledb01v.ng.sbicdirectory.com	10.234.179.212	Deployment DR (Region 2)
yngmobiledb02v.ng.sbicdirectory.com	10.234.179.213	Deployment DR (Region 2)

Ops manager App DB

Server	IP Address	Notes
pngmondbops01.ng.sbicdirectory.com	10.234.19.125	Application Server (Region 1)
pngmondbops02.ng.sbicdirectory.com	10.234.19.126	MongoDB Prod (Region 1)
yngmondbops01.ng.sbicdirectory.com	10.234.178.54	MongoDB DR (Region 2)
yngmondbops02.ng.sbicdirectory.com	10.234.178.55	MongoDB DR (Region 1)

2. Goals of the Consulting Engagement

The primary goal of the engagement was to address the issues that the team is currently facing on the production Atlas cluster and answer all the questions that the team came up with regarding MongoDB best practices. Various topics discussed are as follows:

- Ops Manager Case Debugging
- Indexing Strategy

3. Recommendations

3.1 Ops Manager Monitoring Ping Fail [Priority 1]

During the consultation, the team mentioned that they have an open case where the monitoring agent was showing it was not able to collect the information and the ping ops manager. Upon debugging we found that the active/primary monitoring agent selected was node *yngmobiledb01v.ng.sbicdirectory.com.*

This node is part of APP DB deployment, since this was a part of APPDB it did not have access to other nodes in the ops manager project. Since the node was working correctly the ops manager did not change the active monitoring agent. Hence ops manager monitoring pings failed.

pngarbiter.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *standby*	
Backup - *standby*	
Automation - *active*	
pngmobiledb01v.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *standby*	
Backup - *standby*	
Automation - *active*	
pngmobiledb02v.ng.sbicdirectory.com	**11.0.14.7064**

Monitoring - *standby*	
Backup - *active*	
Automation - *active*	
pngmondbops02.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *standby*	
pngmongdb01.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *standby*	
Automation - *active*	
pngmongdb02.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *standby*	
Automation - *active*	
pngmongdb03.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *active*	
Automation - *active*	
yngmobiledb01v.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *standby*	
Backup - *standby*	
Automation - *active*	
yngmobiledb02v	**11.0.14.7064**

Monitoring - *standby*	
Backup - *standby*	
Automation - *active*	
yngmondbops01.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *standby*	
yngmondbops02.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *standby*	
yngmongdb01.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *standby*	
Automation - *active*	
yngmongdb02.ng.sbicdirectory.com	**11.0.14.7064**
Monitoring - *standby*	
Automation - *active*	

To solve this problem we discussed 3 possible solutions.

3.1.1 Move the APP DB to another project

It is a best practice to only have clusters which have relation with each other in a given project. Since APPDB did not have any direct relationship with mongodb deployments it is recommended that we move the app monitoring project to its own contained project. Here this will separate the APPDB nodes to another project thus its node will never be an active monitoring agent for database cluster deployment.

For creating a new project please refer here.
Please refer here for steps for migration.

3.1.2 Give access to all nodes to connect internally

One of the other ways we can solve this problem is to provide access for each node to access each other. Since all the nodes are now able to access each other the above issue would not occur.

3.1.3 Remove monitoring to nodes where access is not given

One of the other solutions we discussed was the possibility of removing monitoring from nodes which do not have access to each other. This will make sure the node which doesn't have access to each other will never be able to become an active monitoring agent thus solving the above problem.

3.1.4 What is an active monitoring agent?

We can activate Monitoring on multiple MongoDB Agents to distribute monitoring assignments and provide failover. Ops Manager distributes monitoring assignments among up to 100 running MongoDB Agents. Each MongoDB Agent running active Monitoring monitors a different set of MongoDB processes. One MongoDB Agent running active Monitoring per project is the primary Monitor. The primary Monitor reports the cluster's status to Ops Manager. As MongoDB Agents have Monitoring enabled or disabled, Ops Manager redistributes assignments. If the primary Monitor fails, Ops Manager assigns another MongoDB Agent running active Monitoring to be the primary Monitor.

If we run more than 100 MongoDB Agents with active Monitoring, the additional MongoDB Agents run as standby MongoDB Agents. A standby MongoDB Agent is idle, except to log its status as a standby and periodically ask Ops Manager if it should begin monitoring.

> Please note: Only one host can monitor a deployment at a time. On the Server tab, the host that is monitoring the deployment displays Monitoring - active. Any other host with Monitoring activated displays Monitoring - standby.

4. Other Notes, Questions & Answers

4.1 Indexing strategy

4.1.1 General Indexing Strategies

During the consultation, the team wanted to understand the indexing strategies and how to implement them optimally. Hence indexing strategies were discussed. When adding or modifying your approach to indexing, consider the following notes:

- A compound index can be utilized to satisfy multiple queries. For example, if there is a compound index like `{ a:1, b:1, c:1 },` this can be utilized to satisfy all the following queries -

  - `db.coll.find({ a:3 })`
  - `db.coll.find({ a:3, b:5 })`
  - `db.coll.find({ a:3, b:5, c:8 })`

- The order of the index keys is important for a compound index. In general, the following rule of thumb can be used for the key order in compound indexes: First use keys on fields on which there is an **equality** match in the query (these are usually the most "selective" part of the query), then the **sort** keys, and then **range** query keys. We call this the ESR (Equality-Sort-Range) rule. Consider the following example:

  - If the Query shape looks like - `{ a:5, b: {$gt : 5} }.sort({ c:1 })`
  - Index field order should be - `{ a:1, c:1, b:1 }`

  The reason for this structure is that for an equality match, it is expected that it will only select a subset of the index keys (therefore more selective), compared to sorting which will scan all of the index keys for an index.

- Remove indexes that are not used because every index creates overhead for write operations. The $indexStats command can be used for getting the statistics about the usage of the indexes or index usage can be checked using MongoDB Compass. Atlas UI can also be used to check index usage.

- For the fastest processing, ensure that your indexes fit entirely in RAM so that the system can avoid reading the index from the disk. For more information, refer to Ensure Indexes Fit in RAM.

- When possible use covered queries. A covered query is a query that can be satisfied entirely using an index and does not have to examine any documents (it will not show a FETCH stage in the winning plan of explain-results).

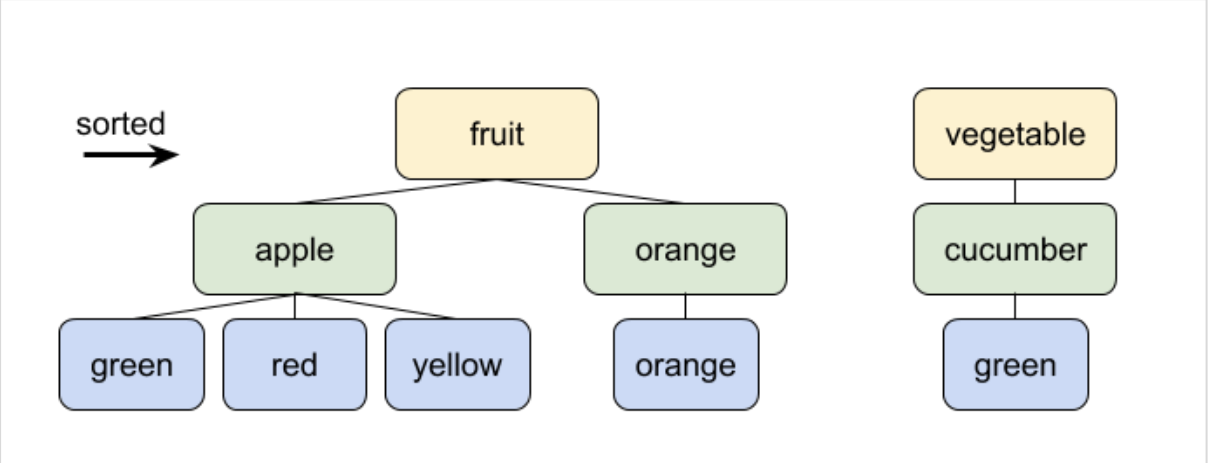
- Use indexes to sort query results. For more information please refer to the link -Use Indexes to Sort Query Results.

- You can make MongoDB use an index with hint() if the optimal index is not used for some reason. Use caution while using *.hint()* because changes to the database's query optimizer may be negated by forcing an index selection with hints.

4.1.2 Create compound indexes to help with sorting

It's common to find indexes that are optimized for querying and others for sorting. Compound indexes should be structured to optimize for both whenever possible. A compound index can benefit both a query and a sort when created properly.

Order matters in a compound index. Indexes are, by definition, a sorted structure. All levels of a tree are sorted in ascending or descending order. Some queries and sorts can use an index while others cannot, all depending on the query structure and sort options. It may help to visualize an index as a generic tree structure to help see why.



The diagram above shows an index on { type: 1, name: 1, color: 1 }. This index will work efficiently for the following queries:

- db.food.find({ type: "fruit" }).sort({ name: 1 })
- db.food.find({ type: "fruit", name: "apple" }).sort({ color: 1 })
- db.food.find({}).sort({ type: 1 })
- db.food.find({ type: "fruit", name: "apple", color: "green" })

All of the examples queries above leverage the fact that every “level” of the tree is sorted. It’s important to note that name and color are both sorted under their respective tree. For example, apple and orange are sorted correctly even though cucumber comes before orange in a basic sort because they do not share a common parent.

The same index can be used for the following queries, but less efficiently:

- db.food.find({ type: "fruit" }).sort({ color: 1 })
- db.food.find({ name: "apple" }).sort({ type: 1 })

Both examples above can use the index to satisfy either the equality or the sort but not both.

Looking at the first example, equality on fruit will eliminate half of the tree structure (i.e. vegetable), but an in-memory sort of color is still required. Orange comes before yellow when sorting only by color, but those colors don’t contain a common parent so they are not sorted for this example. An in-memory sort is now required to sort by color.

The second example has an equality on name and sort on type. The index is already sorted on type so it can be used for that portion, but the entire tree must be traversed to eliminate all entries with the name apple.

For more information on indexing performance, please refer to the following blog
Performance Best Practices: Indexing

4.1.3 Explain Plans

We can get insights about query/aggregation performance using explain plans. We can run an explain plan on a query using .explain() on a query/aggregation. By looking at the execution plan, the following can be determined:

- Access paths being used for fetching the data
- Various stages that go into execution
- How data is being sorted
- What indexes are being used and which fields the indexes are being used for

Using the `explain` method, an explainable object can be constructed to get the `executionStats` as below.

```
db.collection.explain("executionStats").<query/aggregation>
```

Sample Execution plan results

```
"executionStats" : {
   "executionSuccess" : <boolean>,
   "nReturned" : <int>,
   "executionTimeMillis" : <int>,
   "totalKeysExamined" : <int>,
   "totalDocsExamined" : <int>,
   "executionStages" : {
      "stage" : <STAGE1> // Executed second
      "nReturned" : <int>,
      "executionTimeMillisEstimate" : <int>,
      "works" : <int>,
      "advanced" : <int>,
      "needTime" : <int>,
      "needYield" : <int>,
      "saveState" : <int>,
      "restoreState" : <int>,
      "isEOF" : <boolean>,
      ...
      "inputStage" : {
         "stage" : <STAGE2>, // First Executed
         "nReturned" : <int>,
         "executionTimeMillisEstimate" : <int>,
```

```
            ...
            "inputStage" : {
               ...
            }
         }
      },
}
```

Each stage passes its results (i.e. documents or index keys) to the parent node. The leaf nodes access the collection or the indices. The internal nodes manipulate the documents or the index keys that result from the child nodes. The root node is the final stage from which MongoDB derives the result set.

Stages are descriptive of the operation; e.g.

- COLLSCAN for a collection scan
- IXSCAN for scanning index keys
- FETCH for retrieving documents
- SORT in-memory sort

The key metrics to note are

- nReturned: Shows the number of documents returned
- executionTimeMillis: Total time required in milliseconds for the execution of the query
- keysExamined: Number of index entries scanned
- docsExamined: Number of documents scanned from disk
- executionStages: Shows the complete execution path the query took

In a performant system, *nReturned*, *keysExamined* and *docsExamined* should be the same because we should not scan more than what we want to be returned from the database. In other words, there should be a 1:1 ratio between the parameters. Please see Explain Results for more information on explain output.

4.1.4 "$or" clauses and indexes

When evaluating the clauses in the $or expression, MongoDB either performs a collection scan or, if all the clauses are supported by indexes, MongoDB performs index scans. That is, for MongoDB to use indexes to evaluate an $or expression, all the clauses in the $or expression must be supported by indexes. Otherwise, MongoDB will perform a collection scan.

When using indexes with $or queries, each clause of an $or can use its own index. Consider the following query:

```
db.inventory.find( { $or: [ { quantity: { $lt: 20 } }, { price: 10 } ] } )
```

The following indexes need to be created to support the above query-

```
db.inventory.createIndex( { quantity: 1 } )
db.inventory.createIndex( { price: 1 } )
```

In a performant system, *nReturned*, *keysExamined,* and *docsExamined* should be the same because we should not scan more than what we want to be returned from the database. In other words, there should be a 1:1 ratio between the parameters. Please see Explain Results for more information on the explain output.

4.2 Mongolyser

Using the mongolyser tool you can detect, diagnose and anticipate any bottlenecks, issues and red flags inside your MongoDB implementation. The information that the tool examines includes in depth log analysis, query health and efficiency analysis, index analysis and much more. This tool only runs administrative commands and does not contain any DDL or DML commands. It is also worth mentioning that no analytics data leaves the system running the tool.

We helped the team in downloading and installing the latest release from here. This tool is still evolving and will entail new and deeper insights in near future.

Note 1: It is to be noted that the mongolyser tool is ***not*** officially supported by MongoDB. It is to be considered as a 3rd party tool from an unverified source.

Note 2: Certain admin commands being used inside mongolyser's analysis engine, under specific conditions, can cause performance impact. Hence it is recommended to run these analyses in Least User Activity (LUA) hours.

4.3 Atlas Metrics

During the consultation the team wanted to understand their cluster performance hence we discussed Atlas Metrics. The Atlas Metrics tab provides you with essential information that could help you understand the current load on the cluster as well as give you deep insights into how well your workload is getting handled. This not only helps diagnose bottlenecks but also allows informed decisions on sizing the cluster correctly by targeting the resource bottleneck.

Few metrics that are good to focus on when monitoring MongoDB:

- *Opcounters*: Number of queries, updates, inserts, deletes, getmore (other) commands per second. It gives a basic overview of what the system is doing
- *Cache Activity*: This metric represents the WiredTiger cache activity i.e at what rate the data is coming into the cache(read into cache) and at what rate data is moving out of the cache and written to disk(written from cache). If there is high cache activity

with corresponding spikes in Util% and Disk IOPS metric, it signifies that the RAM is not sufficient and the disk is being used heavily.

- *Replication Oplog Window*: This is the amount of time in write operations that the oplog covers, this window represents how long a replicating secondary can be down before losing its ability to catch up to the primary. This should not be allowed to drop below 24 hours.
- *Query targeting:* The ratio of the number of keys and documents scanned to the number of documents returned, averaged by second. A high level of this metric means that the query subsystem has to perform many inefficient scans in order to fulfill queries. This situation typically arises when there are no appropriate indexes to support queries.
    - We saw a high query targeting ratio here hence we discussed indexing strategy to reduce this.
- *Scan and Order:* The average rate of in-memory sort operations performed by the database per second. In-memory sorts can be very expensive as they require large result sets to be buffered. They can be avoided by using appropriate indexes.
- *IOPS*: Displays input operations per second. Monitor whether disk IOPS approaches the maximum provisioned IOPS. Determine whether the cluster can handle future workloads.
- *Disk Queue Depth: Displays the average length of the queue of requests issued to the disk partition that MongoDB uses. Monitor disk queue depth to identify potential issues and bottlenecks.*
    - Here we saw that there was a high number of queue depths. Here we discussed the possibility of increasing the number of IOPS (current limit is around 1K).
- *Normalized System CPU:* Displays the CPU usage of all processes on the node, scaled to a range of 0-100% by dividing by the number of CPU cores. Monitor CPU usage to determine whether data is retrieved from disk instead of memory.

4.4 Write Concerns

Write concern describes the level of acknowledgment requested from MongoDB for write operations. If not provided, the default write concern is `w:1`, in which the write is acknowledged to the application as soon as the data has been written in the memory of the primary node.

With this configuration, MongoDB does not provide the application a strict guarantee about the durability of the data, if the primary encounters an outage before the write has been replicated, the write operation could be rolled back or lost. To be sure that the data you write

will never be rolled back, the application should use the `w: majority` write concern. Since the application will receive the acknowledgment only when the data has been replicated to the majority of the voting nodes, this can increase the write response time depending on the replication lag between the primary and secondary.

Some applications may have varying requirements regarding durability. Clients may adjust write concerns to ensure that the most important operations persist successfully to an entire MongoDB deployment. Clients can adjust write concerns for other less critical operations to ensure faster performance. Refer to the documentation for further information on the Write Concern.

4.5 MongoDB Ops Manager Version Upgrade

During the consultation, we discussed the upgrade of the ops manager. It is to be noted that the current mongodb ops manager version will not be supported from July. Here we just basic steps and we did not perform an upgrade due to lack of time. The team is recommended to avail another consulting session for actual upgrade of ops manager.

4.5.1 Upgrade Path

Upgrade to the highest version of the current major version (5.x → 5.0.21) once the upgrade is successful, upgrade to the latest version in (6.x).

```
5.x → 5.0.21 → 6.0.14
```

For more information please refer here

5. Questions

5.1 How to get the count of the collections efficiently with no filters?

We can use estimatedDocumentCount to get the count of documents from metadata of the collection.

```
db.collection.estimatedDocumentCount( <options> )
```

Please note: After an unclean shutdown, the count may be incorrect.

6. Recommended Further Consulting and Training

MongoDB offers a comprehensive set of instructor-led training courses covering all aspects of building and running applications with MongoDB. Instructor-led training is the fastest and

best way to learn MongoDB in depth. Both public and private training classes are available - for more information or to enroll in classes, please see Instructor-Led Training.