Performance Tuning

ANJU MUNOTH

Examine memory use

- Can use the metrics in the memory section of the serverStatus document to understand how MongoDB is using system memory:

```
db.serverStatus().mem
"mem" : {
   "bits" : <int>,
   "resident" : <int>,
   "virtual" : <int>,
   "supported" : <boolean>,
   "mapped" : <int>,
   "mappedWithJournal" : <int>
},
```

Examine memory use

- mem.resident: Roughly equivalent to the amount of RAM in megabytes that the database process uses
- mem.mapped: The amount of memory that the database maps, in megabytes

To see if we've exceeded the capacity of our system, we can compare the value of mem.resident to the amount of system memory.

- If mem.resident exceeds the value of system memory and there's a large amount of unmapped data on disk, we've most likely exceeded system capacity.
- If the value of mem.mapped is greater than the amount of system memory, some operations will experience page faults.

Tune the WiredTiger cache

- MongoDB will reserve 50 percent of the available memory for the WiredTiger data cache.
- The size of this cache is important to ensure WiredTiger performs adequately.
- It’s worth taking a look to see if you should alter it from the default.
- A good rule of thumb is that the size of the cache should be big enough to hold the entire application working set.
- How do we know whether to alter it? Look at the cache usage statistics:

```
db.serverStatus().wiredTiger.cache
{
   "tracked dirty bytes in the cache" : <num>,
   "tracked bytes belonging to internal pages in the cache" : <num>,
   "bytes currently in the cache" : <num>,
   "tracked bytes belonging to leaf pages in the cache" : <num>,
   "maximum bytes configured" : <num>,
   "tracked bytes belonging to overflow pages in the cache" : <num>,
   "bytes read into cache" : <num>,
   "bytes written from cache" : <num>,
   "pages evicted by application threads" : <num>,
   "checkpoint blocked page eviction" : <num>,
   "unmodified pages evicted" : <num>,
   "page split during eviction deepened the tree" : <num>,
   "modified pages evicted" : <num>,
   "pages selected for eviction unable to be evicted" : <num>,
   "pages evicted because they exceeded the in-memory maximum" : <num>,
   "pages evicted because they had chains of deleted items" : <num>,
   "failed eviction of pages that exceeded the in-memory maximum" : <num>,
   "hazard pointer blocked page eviction" : <num>,
   "internal pages evicted" : <num>,
   "maximum page size at eviction" : <num>,
   "eviction server candidate queue empty when topping up" : <num>,
"eviction server candidate queue not empty when topping up" :
<num>,
    "eviction server evicting pages" : <num>,
    "eviction server populating queue, but not evicting pages" : <num>,
    "eviction server unable to reach eviction goal" : <num>,
    "internal pages split during eviction" : <num>,
    "leaf pages split during eviction" : <num>,
    "pages walked for eviction" : <num>,
    "eviction worker thread evicting pages" : <num>,
    "in-memory page splits" : <num>,
    "in-memory page passed criteria to be split" : <num>,
    "lookaside table insert calls" : <num>,
    "lookaside table remove calls" : <num>,
    "percentage overhead" : <num>,
    "tracked dirty pages in the cache" : <num>,
    "pages currently held in the cache" : <num>,
    "pages read into cache" : <num>,
    "pages read into cache requiring lookaside entries" : <num>,
    "pages written from cache" : <num>,
    "page written requiring lookaside records" : <num>,
    "pages written requiring in-memory restoration" : <num>
}
```

Tune the WiredTiger cache

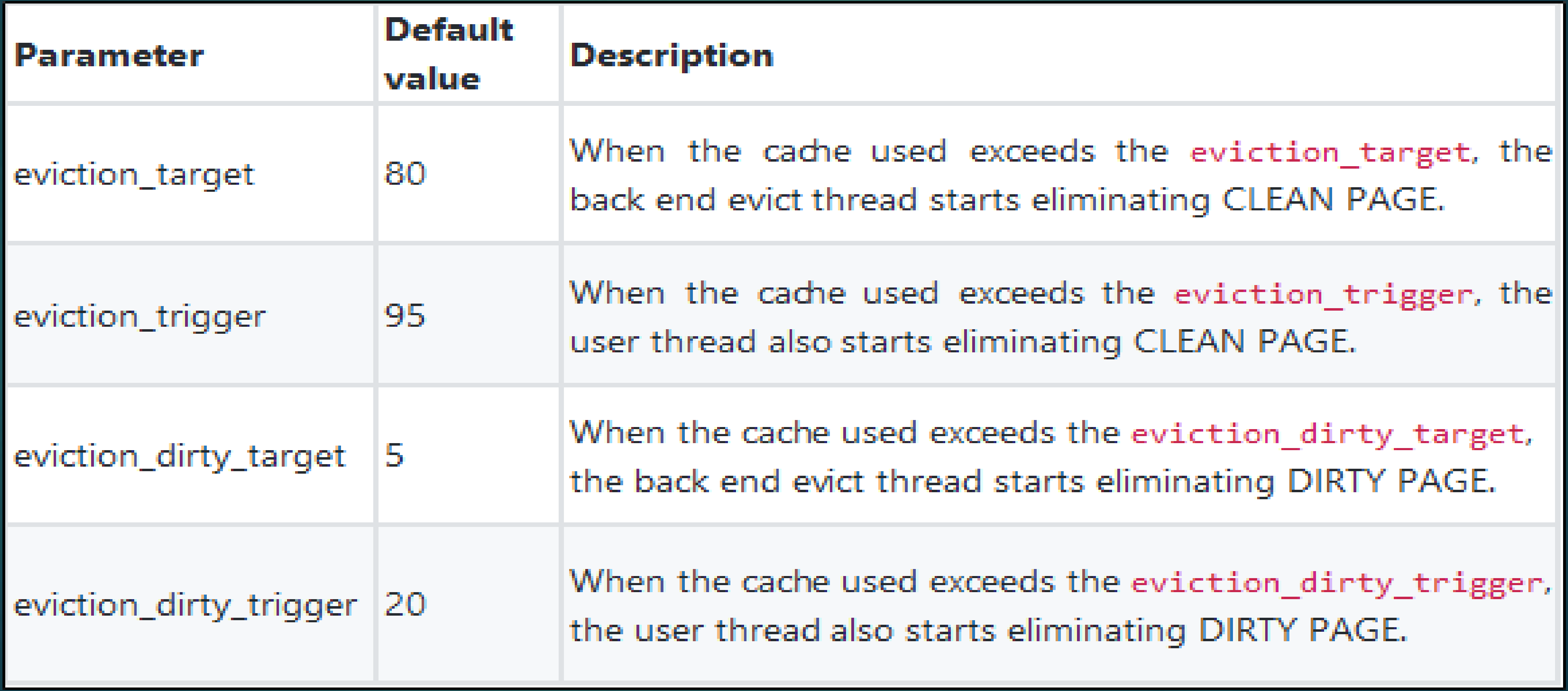
There's a lot of data here, but we can focus on the following fields:

- wiredTiger.cache.maximum bytes configured: This is the maximum cache size.
- wiredTiger.cache.bytes currently in the cache – This is the size of the data currently in the cache. This should not be greater than the maximum bytes configured.
- wiredTiger.cache.tracked dirty bytes in the cache – This is the size of the dirty data in the cache. This value should be less than the bytes currently in the cache value.
- Looking at these values, we can determine if we need to up the size of the cache for our instance.
- Additionally, we can look at the **wiredTiger.cache.bytes read** into cache value for read-heavy applications. If this value is consistently high, increasing the cache size may improve overall read performance.

Storage Engine Cache

- MongoDB 3.2 and after use as a default the WiredTiger storage engine.
- The upper limit of the memory used by the WiredTiger engine can be set by cacheSizeGB, and the general recommended setting is about 60% of the system's usable memory (default setting).
- To cite an example, if cacheSizeGB is set at 10 GB, you can think of it as if the total amount of memory allocated through tcmalloc by the WiredTiger engine does not exceed 10 GB.
- In order to control the use of memory, WiredTiger begins elimination when the memory use approaches a certain threshold, to prevent user requests from being blocked when the memory use is full.
- At present there are 4 configurable parameters that support the eviction policy of the WiredTiger storage engine (in general it does not require revision)

Storage Engine Cache



Parameter	Default value	Description
eviction_target	80	When the cache used exceeds the `eviction_target`, the back end evict thread starts eliminating CLEAN PAGE.
eviction_trigger	95	When the cache used exceeds the `eviction_trigger`, the user thread also starts eliminating CLEAN PAGE.
eviction_dirty_target	5	When the cache used exceeds the `eviction_dirty_target`, the back end evict thread starts eliminating DIRTY PAGE.
eviction_dirty_trigger	20	When the cache used exceeds the `eviction_dirty_trigger`, the user thread also starts eliminating DIRTY PAGE.

Storage Engine Cache

- Under this rule, in an instance where MongoDB is operating normally, the cache used will generally be 0.8 * cacheSizeGB and under, and there is no major problem if this is occasionally exceeded;
- if used > = 95% or dirty >= 20% occurs and is sustained for a time, this means that the memory elimination pressure is very large, and the users' request threads will be blocked from participating in page elimination; when a request is delayed the pressure increases, and at this time "increasing memory" or "switching to a faster disk to raise IO capabilities" can be considered.

Monitor the number of current connections

- Unless system limits constrain it, MongoDB has no limits on incoming connections.
- But there’s a catch. In some cases, a large number of connections between the application and database can overwhelm the database.
- This will limit its ability to handle additional connections.
- Can find the connections metrics in the connections section of the serverStatus document:

```
> db.serverStatus().connections
{
   "current" : <num>,
   "available" : <num>,
   "totalCreated" : NumberLong(<num>)
}
```

Monitor the number of current connections

- connections.current: The total number of current clients connected to this instance
- connections.available: The total number of unused connections available to clients for this instance
- If connection issues are a problem, there are a couple of strategies for resolving them. First, check whether the application is read-heavy.
- If it is, increase the size of the replica set and distribute the read operations to secondary members of the set.
- If this application is write-heavy, use sharding within a sharded cluster to distribute the load.
- Application- or driver-related errors can also cause connection issues.
- For instance, a connection may be disposed of improperly or may open when not needed, if there's a bug in the driver or application.
- See this if the number of connections is high, but there's no corresponding workload.

Watch replication performance

- Replication is the propagation of data from one node to another.
- It’s key to MongoDB being able to meet availability challenges.
- As data changes, it’s propagated from the primary node to secondary nodes. Replication sets handle this replication.

Monitor replication lag

- In a perfect world, data would be replicated among nodes almost instantaneously.
- But we don't live in a perfect world.
- Sometimes, data isn't replicated as quickly as we'd like. And, depending on the time it takes for a replication to occur, we run the risk of data becoming out of sync.
- This is a particularly thorny problem if the lag between a primary and secondary node is high and the secondary becomes the primary.
- Because the replication didn't occur quickly enough, data will be lost when the newly elected primary replicates to the new secondary.

Monitor replication lag

- Can use the **db.printSlaveReplicationInfo() or the rs.printSlaveReplicationInfo()** command to see the status of a replica set from the perspective of the secondary member of the set.
- The following is an example of running this command on a replica set with two secondary members:

```
source: m1.example.net:27017
syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
0 secs (0 hrs) behind the primary
source: m2.example.net:27017
syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
0 secs (0 hrs) behind the primary
```

Monitor replication lag

- output of this command shows how far behind the secondary members are from the primary.
- This number should be as low as possible.
- However, it’s really going to be based on the application’s tolerance for a delay in replication.
- Should monitor this metric closely.
- In addition, administrators should watch for any spikes in replication delay.
- If replication lag is consistently high or it increases at a regular rate, that’s a clear sign of environmental or systemic problems.
- Always investigate these issues to understand the reasons for the lag.

Monitor replication state

- As mentioned above, replica sets handle replication among nodes.
- One replica set is primary. All others are secondary.
- Under normal conditions, the assigned node status should rarely change.
- If a role change does occur—that is, a secondary node is elected primary—we want to know immediately.
- The election of a new primary usually occurs seamlessly. Still, you should understand what caused the status change.
- That’s because it could be due to a network or hardware failure.
- In addition, it’s not normal for nodes to change back and forth between primary and secondary.

Database profiler

- Can gather additional detailed performance information using the built-in database profiler.
- If additional performance tuning needs to happen,can use the profiler to gain a deeper understanding of the database’s behavior.
- Profiler collects information about all database commands that are executed against an instance.
- Includes the common create, read, update, and delete operations.
- Also covers all configuration and administration commands.
- However, there’s a catch. Enabling the profiler can affect system performance, due to the additional activity.

Configuring the MongoDB profiler

- Profiler stores the queries in db.system.profile.
- Can be queried just like any other database, so you can find a lot of information quickly.
- As well as the **source of the query** or command, there is a summary of the execution stats that tell you, amongst other information:
- ❑ when the query was run
- ❑ how fast it ran
- ❑ how many documents were examined
- ❑ the type of plan it used
- ❑ whether it was able to fully-use and index
- ❑ what sort of locking happened

db.getProfilingStatus()

- Can check your current profiling status using the command:

  db.getProfilingStatus()

- Can then change the status of the MongoDB profiler according to what we want it to do.

- Level 0 – The profiler is off and does not collect any data. This is the default profiler level.
- Level 1 – The profiler collects data for operations that take longer than the value of slowms, which you can set.
- Level 2 – The profiler collects data for all operations.

db.setProfilingLevel()

- Using the mongo shell or a GUI-integrated shell run the following command:

db.setProfilingLevel(2)

- which, in my case, returns:
- '{ "was" : 0, "slowms" : 30, "ok" : 1 }'

db.setProfilingLevel(1, { slowms: 30 })

Sort queries by when they were recorded, showing just commands

```
db.system.profile.find(
{
"command.pipeline": { $exists: true }
},
 {
"command.pipeline":1
})
.sort({$natural:-1}).pretty();
```

Find all queries doing a COLLSCAN because there is no suitable index

- // all queries that do a COLLSCAN

```
db.system.profile.find(
{
"planSummary":{$eq:"COLLSCAN"},
"op" :{ $eq:"query“}
}).sort({millis:-1})
```

Find any query or command doing range or full scans

- // any query or command doing range or full scans

```
db.system.profile.find({"nreturned":{$gt:1}})
```

Find the source of the top ten slowest queries

- Can show the source, nicely formatted, of the top ten slowest queries.
- This query lists the source of the top slowest queries in the order of slowest first.

```
db.system.profile.find(
{"op" : {$eq:"query"}},
{"query" : 1,"millis":1}
).sort({millis:-1},{$limit:10}).pretty()
```

Find the source of the top ten slowest aggregations

- // the source of the top ten slowest commands/aggregations

```
db.system.profile.find(
{"op" : {$eq:"command"}},
 {"command.pipeline" : 1,"millis": 1}
).sort({millis:-1},{$limit:10}).pretty()
```

Find all queries that take more than ten milliseconds, in descending order, displaying both queries and aggregations

- //find all queries that take more than ten milliseconds, in descending order,
- displaying both queries and aggregations

```
db.system.profile.find(
{"millis":{$gt:10}})
.sort({millis:-1})
```

find all queries and aggregations that take more than ten milliseconds, in descending

- //find all queries and aggregations that take more than ten milliseconds, in descending
- order, displaying either the query or aggregation

```
db.system.profile.find(
{ "millis": { $gt: 10 } },
 { millis: NumberInt(1),
"query": NumberInt(1), "command.pipeline": 1 })
.sort({ millis: -1 })
```

- To return the most recent 10 log entries in the system.profile collection, run a query similar to the following:

```
db.system.profile.find().limit(10).sort( { ts : -1 } ).pretty()
```

- To return all operations except command operations ($cmd), run a query similar to the following:

- db.system.profile.find({ op: { $ne : 'command' } }).pretty()

- To return operations for a particular collection, run a query similar to the following. This example returns operations in the mydb database’s test collection:

- db.system.profile.find({ ns : 'mydb.test' }).pretty()

- To return information from a certain time range, run a query similar to the following:

```
db.system.profile.find({
  ts : {
    $gt: new ISODate("2012-12-09T03:00:00Z"),
    $lt: new ISODate("2012-12-09T03:40:00Z")
  }
}).pretty()
```

- The following example looks at the time range, suppresses the user field from the output to make it easier to read, and sorts the results by how long each operation took to run:

```
db.system.profile.find({
  ts : {
    $gt: new ISODate("2011-07-12T03:00:00Z"),
    $lt: new ISODate("2011-07-12T03:40:00Z")
  }
}, { user: 0 }).sort( { millis: -1 } )
```

Show the Five Most Recent Events

- On a database that has profiling enabled, the show profile helper in the mongo shell displays the 5 most recent operations that took at least 1 millisecond to execute.
- Issue show profile from the mongo shell, as follows:
- show profile

- MongoDB Profiler Cheat Sheet.docx

system.profile.op

- The type of operation.

The possible values are:

- command
- count
- distinct
- geoNear
- getMore
- group
- insert
- mapReduce
- query
- remove
- update

- system.profile.ns
- The namespace the operation targets. Namespaces in MongoDB take the form of the database, followed by a dot (.), followed by the name of the collection.

- system.profile.command
- A document containing the full command object associated with this operation.

system.profile.command

- For example, the following output contains the command object for a find operation on a collection named items in a database named test:

```
"command" : {
  "find" : "items",
  "filter" : {
    "sku" : 1403978
  },
  ...
  "$db" : "test"
}
```

- system.profile.originatingCommand
- For "getmore" operations which retrieve the next batch of results from a cursor, the originatingCommand field contains the full command object (e.g. find or aggregate) which originally created that cursor.

- system.profile.cursorid
- The ID of the cursor accessed by a query and getmore operations.

system.profile.keysExamined

- The number of index keys that MongoDB scanned in order to carry out the operation.
- keysExamined is available for the following commands and operations:
- aggregate
- find (OP_QUERY and command)
- findAndModify
- count
- distinct
- getMore (OP_GET_MORE and command)
- mapReduce
- delete
- update

system.profile.docsExamined

- The number of documents in the collection that MongoDB scanned in order to carry out the operation.
- docsExamined is available for the following commands and operations:
- aggregate
- find (OP_QUERY and command)
- findAndModify
- count
- distinct
- getMore (OP_GET_MORE and command)
- mapReduce
- delete
- update

system.profile.hasSortStage

- hasSortStage is a boolean that is true when a query cannot use the ordering in the index to return the requested sorted results; i.e. MongoDB must sort the documents after it receives the documents from a cursor.
- The field only appears when the value is true.
- hasSortStage is available for the following commands and operations:
- find (OP_QUERY and command)
- getMore (OP_GET_MORE and command)
- findAndModify
- mapReduce
- aggregate

system.profile.usedDisk

- A boolean that indicates whether any aggregation stage wrote data to temporary files due to memory restrictions.
- Only appears if usedDisk is true.

system.profile.ndeleted

- The number of documents deleted by the operation.

system.profile.ninserted

- The number of documents inserted by the operation.

system.profile.nMatched

- The number of documents that match the system.profile.query condition for the update operation.

system.profile.nModified

- The number of documents modified by the update operation.

system.profile.upsert

- A boolean that indicates the update operation’s upsert option value. Only appears if upsert is true.

system.profile.fromMultiPlanner

- A boolean that indicates whether the query planner evaluated multiple plans before choosing the winning execution plan for the query.
- Only present when value is true.

system.profile.replanned

- A boolean that indicates whether the query system evicted a cached plan and re-evaluated all candidate plans.
- Only present when value is true.

system.profile.replanReason

- New in version 4.4.
- A string that indicates the specific reason a cached plan was evicted.
- Only present when value for replanned is true.

system.profile.keysInserted

- The number of index keys inserted for a given write operation.

system.profile.writeConflicts

- The number of conflicts encountered during the write operation; e.g. an update operation attempts to modify the same document as another update operation.

system.profile.locks

- The system.profile.locks provides information for various lock types and lock modes held during the operation.

system.profile.storage

- The system.profile.storage information provides metrics on the storage engine data and wait time for the operation.
- Specific storage metrics are returned only if the values are greater than zero.

system.profile.storage

- system.profile.storage.data.bytesRead --Number of bytes read by the operation from the disk to the cache.
- system.profile.storage.data.timeReadingMicros ---Time in microseconds that the operation had to spend to read from the disk.
- system.profile.storage.data.bytesWritten ---Number of bytes written by the operation from the cache to the disk.

system.profile.storage

- system.profile.storage.data.timeWritingMicros ---Time in microseconds that the operation had to spend to write to the disk.

- system.profile.storage.timeWaitingMicros.cache ---Time in microseconds that the operation had to wait for space in the cache.

- system.profile.storage.timeWaitingMicros.schemaLock ---Time in microseconds that the operation (if modifying the schema) had to wait to acquire a schema lock.
- system.profile.storage.timeWaitingMicros.handleLock ---Time in microseconds that the operation had to wait to acquire the a lock on the needed data handles.

system.profile.storage

- system.profile.nreturned ----The number of documents returned by the operation.

- system.profile.responseLength -- The length in bytes of the operation’s result document. A large responseLength can affect performance.
- To limit the size of the result document for a query operation, you can use any of the following:

- Projections
- The limit() method
- The batchSize() method

system.profile.storage

- system.profile.protocol --The MongoDB Wire Protocol request message format.
- system.profile.millis ---The time in milliseconds from the perspective of the mongod from the beginning of the operation to the end of the operation.
- system.profile.planSummary ---A summary of the execution plan.

system.profile.execStats

- A document that contains the execution statistics of the query operation.
- For other operations, the value is an empty document.
- The system.profile.execStats presents the statistics as a tree; each node provides the statistics for the operation executed during that stage of the query operation.
- system.profile.execStats.stage
  - The descriptive name for the operation performed as part of the query execution; e.g.
- COLLSCAN for a collection scan
- IXSCAN for scanning index keys
- FETCH for retrieving documents
- system.profile.execStats.inputStages --An array that contains statistics for the operations that are the input stages of the current stage.

System.profile

- system.profile.ts ---The timestamp of the operation.
- system.profile.client ---The IP address or hostname of the client connection where the operation originates.
- system.profile.appName ---The identifier of the client application which ran the operation. Use the appName connection string option to set a custom value for the appName field.

System.profile

- system.profile.allUsers ----An array of authenticated user information (user name and database) for the session.
- system.profile.user ---The authenticated user who ran the operation. If the operation was not run by an authenticated user, this field’s value is an empty string.

Enable Profiling for an Entire mongod Instance

- For development purposes in testing environments, you can enable database profiling for an entire mongod instance.
- The profiling level applies to all databases provided by the mongod instance.
- To enable profiling for a mongod instance, pass the following options to mongod at startup.

```
mongod --profile 1 --slowms 15 --slowOpSampleRate 0.5
```

- Alternatively, can specify operationProfiling in the configuration file.

How to detect the queries that lead to CPU load spikes

- db.currentOp()
- Database Profiler
- Explain()
- db.serverStatus()

db.currentOp()

- db.currentOp() function lists the currently running queries with very detailed information.
- Also includes the duration they have been running (secs_running).
- Can go to MongoDB console, run this function and analise the output,
- Can start with a filter (secs_running) to reduce the output, e.g:

db.currentOp({“secs_running”: {$gte: 3}})

- **Going to give a wealth of information about the database on which query is running, the actual query, the time taken, locks, plan summary, user who is giving the query…..**

db.currentOp()

- most important keys to analize:
- active: means the query is ‘in progress’ state
- secs_running: query’s duration, in seconds
- ns: a collection name against you perform the query.
- query: the query body.

db.serverStatus()

- Returns a document that provides an overview of the database process's state.
- This command provides a wrapper around the database command serverStatus.

db.serverStatus()

Instance Information

- "host" : <string>,
- "advisoryHostFQDNs" : <array>,
- "version" : <string>,
- "process" : <"mongod" | "mongos">,
- "pid" : <num>,
- "uptime" : <num>,
- "uptimeMillis" : <num>,
- "uptimeEstimate" : <num>,
- "localTime" : ISODate(""),

db.serverStatus()

asserts

```
"asserts" : {
  "regular" : <num>,
  "warning" : <num>,
  "msg" : <num>,
  "user" : <num>,
  "rollovers" : <num>
},
```

- A document that reports on the number of assertions raised since the MongoDB process started. While assert errors are typically uncommon, if there are non-zero values for the asserts, you should check the log file for more information

db.serverStatus()

- connections

```
"connections" : {
   "current" : <num>,
   "available" : <num>,
   "totalCreated" : <num>,
   "active" : <num>,
   "exhaustIsMaster" : <num>,
   "awaitingTopologyChanges" : <num>
},
```

A document that reports on the status of the connections. Use these values to assess the current load and capacity requirements of the server.

electionMetrics

The electionMetrics section provides information on elections called by this mongod instance in a bid to become the primary:

```
"electionMetrics" : {
   "stepUpCmd" : {
      "called" : <NumberLong>,
      "successful" : <NumberLong>
   },
   "priorityTakeover" : {
      "called" : <NumberLong>,
      "successful" : <NumberLong>
   },
   "catchUpTakeover" : {
      "called" : <NumberLong>,
      "successful" : <NumberLong>
   },
   "electionTimeout" : {
      "called" : <NumberLong>,
      "successful" : <NumberLong>
   },
   "freezeTimeout" : {
      "called" : <NumberLong>,
      "successful" : <NumberLong>
   },
```

Election metrics contd.

```
 "numStepDownsCausedByHigherTerm" :
<NumberLong>,
   "numCatchUps" : <NumberLong>,
   "numCatchUpsSucceeded" : <NumberLong>,
   "numCatchUpsAlreadyCaughtUp" :
<NumberLong>,
   "numCatchUpsSkipped" : <NumberLong>,
   "numCatchUpsTimedOut" : <NumberLong>,
   "numCatchUpsFailedWithError" :<NumberLong>,
   "numCatchUpsFailedWithNewTerm" :
<NumberLong>,

"numCatchUpsFailedWithReplSetAbortPrimaryCatch
UpCmd" : <NumberLong>,
   "averageCatchUpOps" : <double>
}
```

db.serverStatus()

- And many more…..

Analyze locking performance

- Databases operate in an environment that consists of numerous reads, writes, and updates.
- Any one of a hundred clients can trigger any of these activities.
- They're often not sequential, and they frequently use data that another request is in the middle of updating.
- For example, if a client attempts to read a document that another client is updating, conflicts can occur.
- And this can cause lost or unexpectedly altered data.
- To get around this issue and maintain consistency, databases will lock certain documents or collections.
- When a lock occurs, no other operation can read or modify the data until the operation that initiated the lock is finished.
- This prevents conflicts. But it can also severely degrade the database's performance.

Analyze locking performance

- Consider another example. If a read operation is waiting for a write operation to complete, and the write operation is taking a long time, additional operations will also have to wait.
- In the case of a large write or read, that alone can be enough to noticeably degrade database performance.
- If the server is unresponsive for too long, it can cause a replica state change, which can lead to further cascading problems.

How to view lock metrics

- MongoDB provides some useful metrics to help determine if locking is affecting database performance.
- globalLock and locks sections of the db.serverStatus() command output:

```
db.serverStatus().globalLock
db.serverStatus().locks
```

Global lock

```
db.serverStatus().globalLock
{
   "totalTime" : <num>,
   "currentQueue" : {
      "total" : <num>,
      "readers" : <num>,
      "writers" : <num>
   },
   "activeClients" : {
      "total" : <num>,
      "readers" : <num>,
      "writers" : <num>
   }
}
```

Global lock

- globalLock.currentQueue.total: This number can indicate a possible concurrency issue if it’s consistently high.
  - This can happen if a lot of requests are waiting for a lock to be released.
- globalLock.totalTime: If this is higher than the total database uptime, the database has been in a lock state for too long.

db.serverStatus().locks

```
{
   <type> : {
      "acquireCount" : {
      <mode> : NumberLong(<num>),
      ...
   },
   "acquireWaitCount" : {
      <mode> : NumberLong(<num>),
      ...
   },
   "timeAcquiringMicros" : {
      <mode> : NumberLong(<num>),
      ...
   },
   "deadlockCount" : {
      <mode> : NumberLong(<num>),
      ...
   }
}
```

Locks

- locks.<type>.acquireCount: Number of times the lock was acquired
- locks.<type>.acquireWaitCount: Number of times the locks.acquireCount encountered waits because of conflicting locks
- locks.<type>.timeAcquiringMicros: Cumulative wait time for lock acquisitions, in microseconds
- locks.deadlockCount: Number of times the lock acquisitions have encountered deadlocks

Locks

- Each of these possible lock types tracks the above metrics:

- Global: Global locks
- MMAPV1Journal: Locks to synchronize journal writes
- Database: Database locks
- Collection: Collection-related locks
- Metadata: Metadata-related locks
- Oplog: Operational log locks