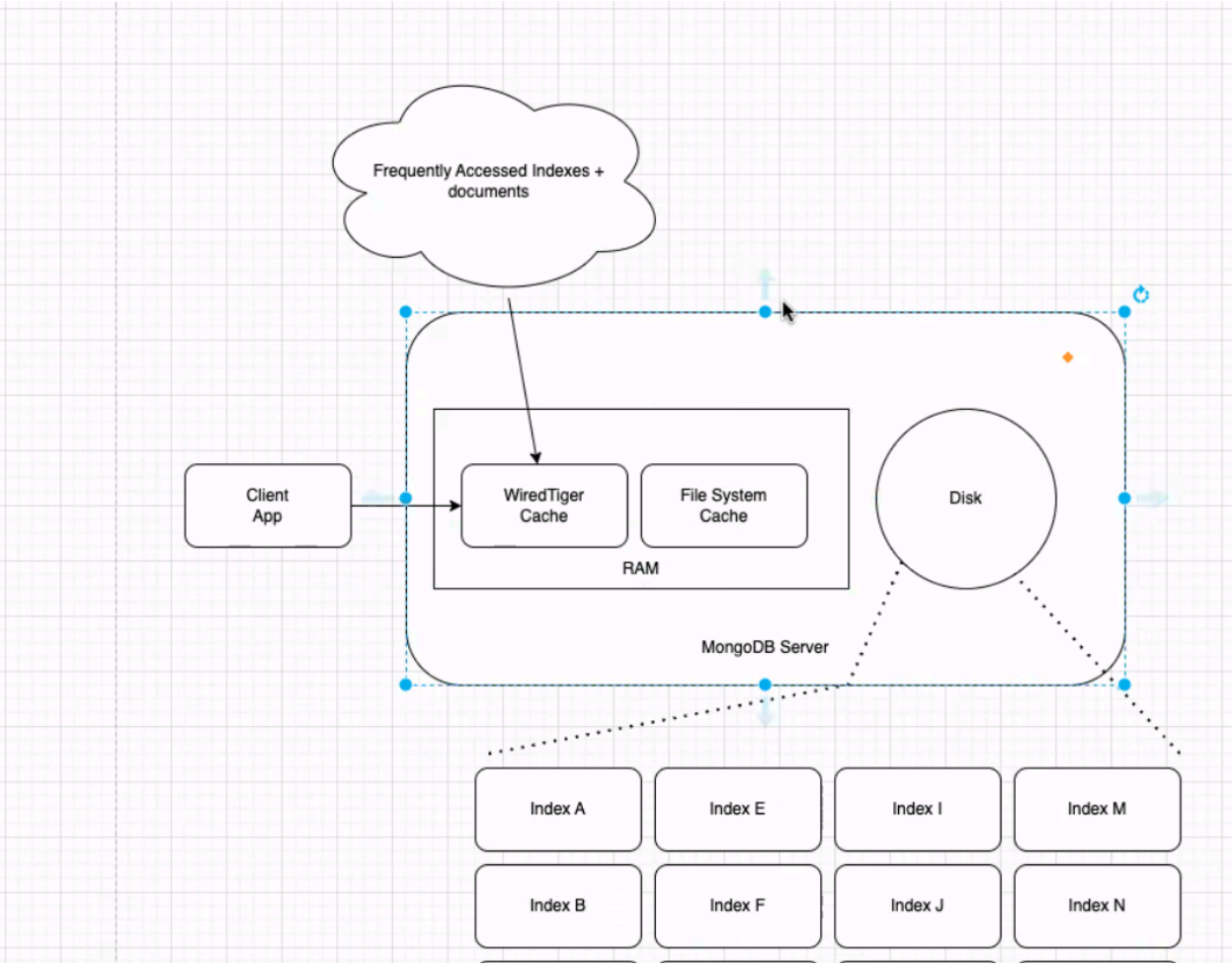
MDB200

index



The default WiredTiger cache size in MongoDB is the larger of **50% of the RAM minus 1 GB, or 256 MB**.

How do Indexes work

Data in an index is ordered so it can be searched efficiently.

Values in an index point to document identity (not _id).

As a result, if the document moves the index doesn't change.

cursor.showrecordid() can show the document identity(record id)

Index Prefix Compression

MongoDB Indexes use a special compressed format

Each entry is a delta from the previous one

If there are identical entries, they need only one byte

As indexes are inherently sorted, this makes them much smaller

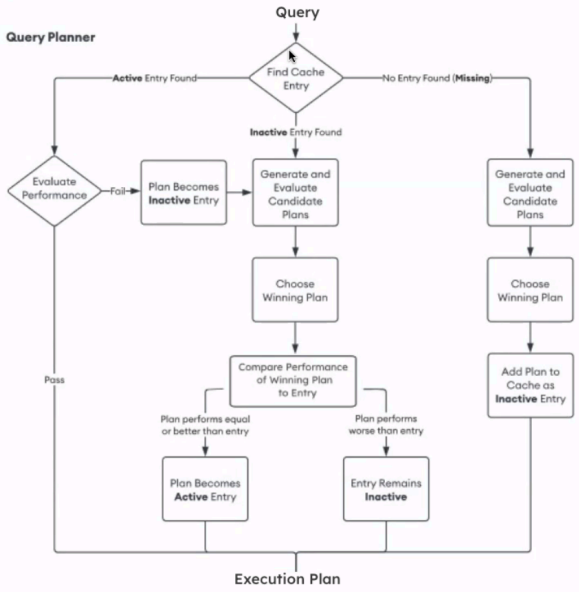
Smaller indexes require less RAM

Using and choosing Indexes

MongoDB checks in the **PlanCache** to see if an optimal index has been chosen before.

If not:

- Picks all candidate indexes
- Runs query using them to score which is most efficient
- Adds its choice of best index to the PlanCache



11

Using and choosing Indexes

Plan cache entries are automatically evicted when:

- Using that index becomes less efficient
- A new relevant index is added
- The server is restarted

Can also be manually cleared using:

- db.<collectionName>.getPlanCache().clear()

Query explain results show information on the Query Plan and execution statistics

clear cache wont help to change to other plan unless create new index()

Explain verbosity

`queryPlanner` shows the winning query plan but does not execute query.

`executionStats` executes query and gathers statistics.

`allPlansExecution` runs all candidate plans and gathers statistics.

If you do not specify a verbosity - the default is `"queryPlanner"`

find(xxx).explain("queryPlanner") verbosity inside

Single-Field Indexes

Optimize finding values for a given field

- Specified as field name and direction
- The direction is irrelevant for a single field index
- The field itself can be any data type.

Avoid indexing an Object type:

- Indexes the whole object as a comparable blob
- Can use it for range searches, but better to index individual fields as less error prone

Indexing an Array is covered later

Index Demonstration

Here we see two stages:

IXSCAN (Look through the index) returning a list of 11 documents identities

FETCH (Read known document) getting the document itself

The total time was one millisecond

```
MongoDB> db.listingsAndReviews.find({
      number_of_reviews:50}).explain("executionStats")
...
    winningPlan: {
      queryPlan: {
        stage: 'FETCH',
        planNodeId: 2,
        inputStage: {
          stage: 'IXSCAN',
          planNodeId: 1,
          keyPattern: {
            number_of_reviews: 1
          },
          indexName: 'number_of_reviews_1',
...
      executionStats: {
            nReturned: 11,
            executionTimeMillis: 1,
            totalKeysExamined: 11,
            totalDocsExamined: 11,
...
```

create index better do rolling update(secondary first, switch current primary to secondary, then update again)

recommend have 4 index most for one collection

Index Limitations

Up to 64 indexes per collection - Avoid being close to this upper bound

Write performance degrades to unusable between 20 and 30

4 indexes per collection is a good recommended number

https://www.mongodb.com/docs/manual/reference/limits/#:~:text=A single collection can have no more than 64 indexes

one index can affect 10% write performance

In MongoDB's explain output, the needYield field indicates how many times the query execution was paused to yield to other operations. This mechanism ensures that long-running queries do not monopolize system resources, allowing MongoDB to maintain overall performance and responsiveness.

Understanding needYield:

- **Purpose:** MongoDB is designed to handle concurrent operations efficiently. When a query runs for an extended period, it may periodically yield control to allow other operations to proceed. The needYield counter tracks these yield occurrences during the query's execution.
- **Implications:** A higher needYield value suggests that the query was long-running or resource-intensive, necessitating multiple yields. While yielding helps maintain system balance, frequent yields might indicate a need to optimize the query or its associated indexes to enhance performance.

Example Scenario:

Consider a collection with a large dataset where a query performs a full collection scan. During this scan, MongoDB may yield multiple times to accommodate other operations. The explain output for such a query would reflect a higher needYield count, signaling the potential benefit of adding appropriate indexes to reduce the query's execution time and the necessity to yield.

Optimizing Queries with High needYield Counts:

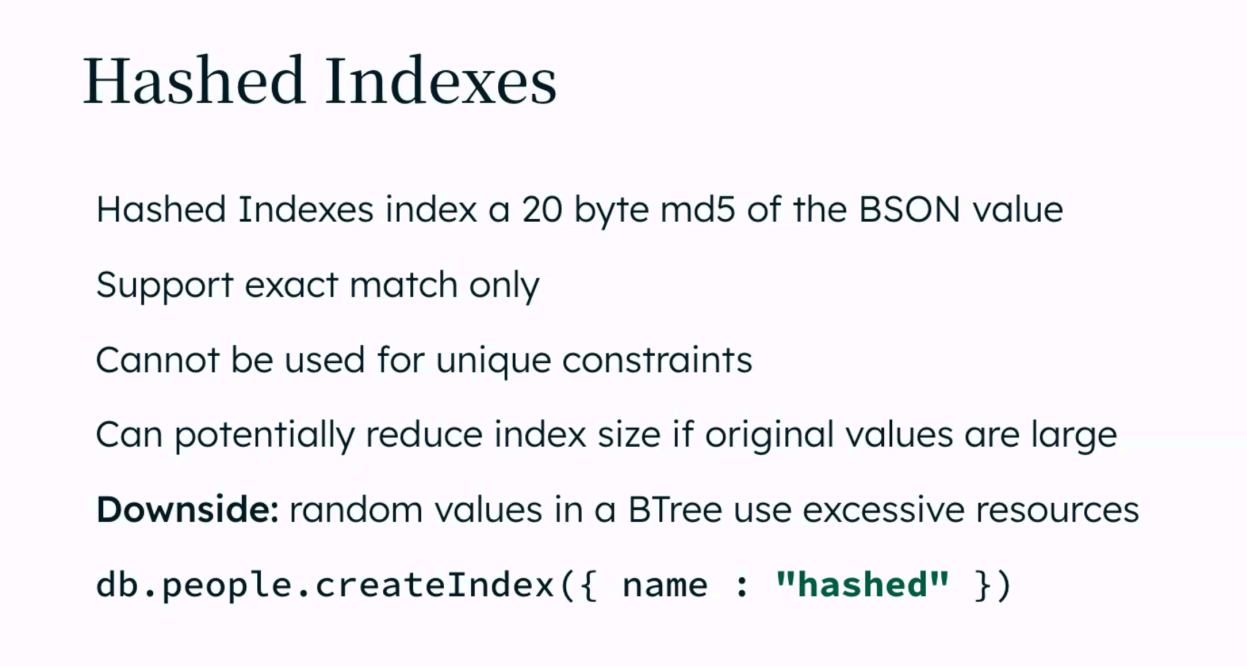
- **Indexing:** Ensure that the query utilizes indexes effectively. Proper indexing can significantly reduce the need for full collection scans, thereby decreasing the needYield count.
- **Query Refinement:** Review and optimize the query structure to make it more efficient, potentially reducing execution time and resource consumption.
- **Resource Allocation:** Assess server resources and workload distribution to ensure that the system can handle concurrent operations without excessive yielding.

By monitoring the needYield metric within the explain output, database administrators and developers can gain insights into query performance and take

proactive steps to optimize long-running operations, leading to a more efficient and responsive MongoDB environment.

Key Requirements for a Covered Query:

1. **Indexed Fields in Query Filter:** All fields specified in the query's filter must be included in an index.
2. **Indexed Fields in Query Projection:** All fields specified in the query's projection (i.e., the fields to return) must also be part of the same index.
3. **Exclusion of _id Field:** By default, MongoDB includes the _id field in query results. If the _id field is not part of the index, it must be explicitly excluded in the projection to achieve a covered query.



Multikey Indexes

- A multikey index is an index that has an array as one of the fields being indexed
  - Can index primitives, documents, or sub-arrays
  - Are created using `createIndex()` just as single-field indexes
- An index entry is created on each unique value found in an array

```
db.race_results.insertMany([
{ "lap_times" : [ 3, 5, 2, 8 ] },
{ "lap_times" : [ 1, 6, 4, 2 ] },
{ "lap_times" : [ 6, 3, 3, 8 ] }
 ])
db.race_results.find( { lap_times : 1 } )
db.race_results.find( { "lap_times.2" : 3 } )
```

"lap_times.2" the third index postion same as [2] for array

```
db.blog.drop()
db.blog.insertMany([
{"comments": [{ "name" : "Bob", "rating" : 1 },
              { "name" : "Frank", "rating" : 5.3 },
              { "name" : "Susan", "rating" : 3 } ]},

{"comments": [{ name : "Megan", "rating" : 1 } ] },
```

```
{"comments": [{ "name" : "Luke", "rating" : 1.4 },
              { "name" : "Matt", "rating" : 5 },
              { "name" : "Sue", "rating" : 7 } ] }
])
db.blog.createIndex( { "comments" : 1 } )
db.blog.createIndex( { "comments.rating" : 1 } )
db.blog.find( { "comments" : { "name" :"Bob", "rating": 1 }})。find the first one
db.blog.find( { "comments" : { "rating" : 1 } } ).  no result due to no content/i,index
db.blog.find( { "comments.rating" : 1 } ) find the result, due to create index
```

first field need to match

ESR guideline

Field Order Matters

Equality First

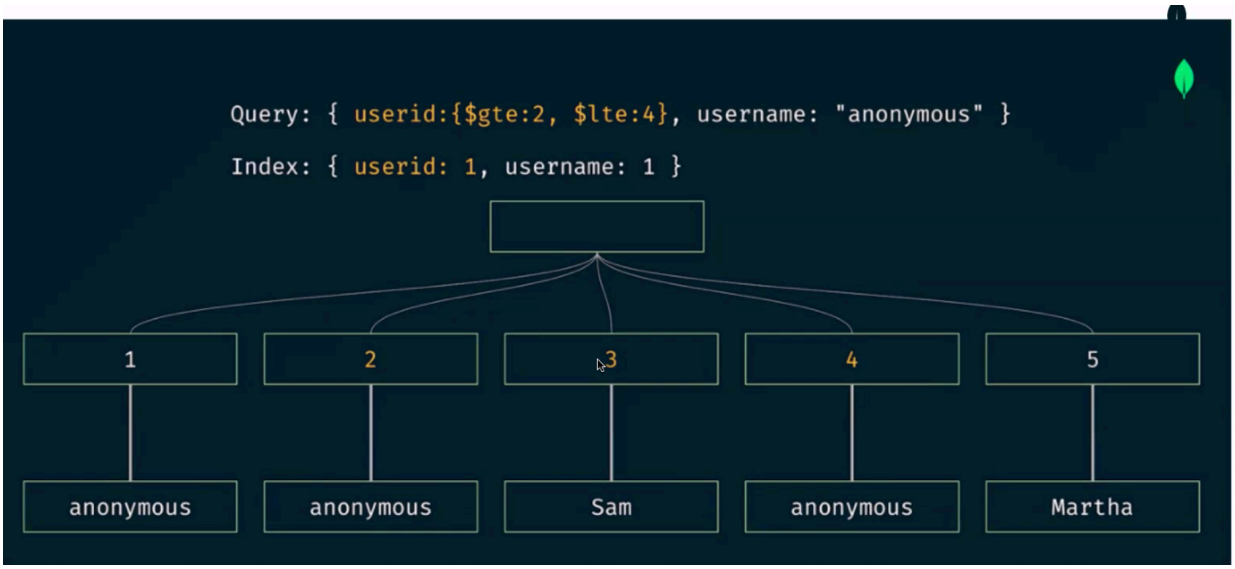
- What fields, for a typical query, are filtered the most
- Selectivity is NOT cardinality, selective can be a boolean choice
- Normally Male/Female is not selective (for the common query case)
- Dispatched versus Delivered IS selective though

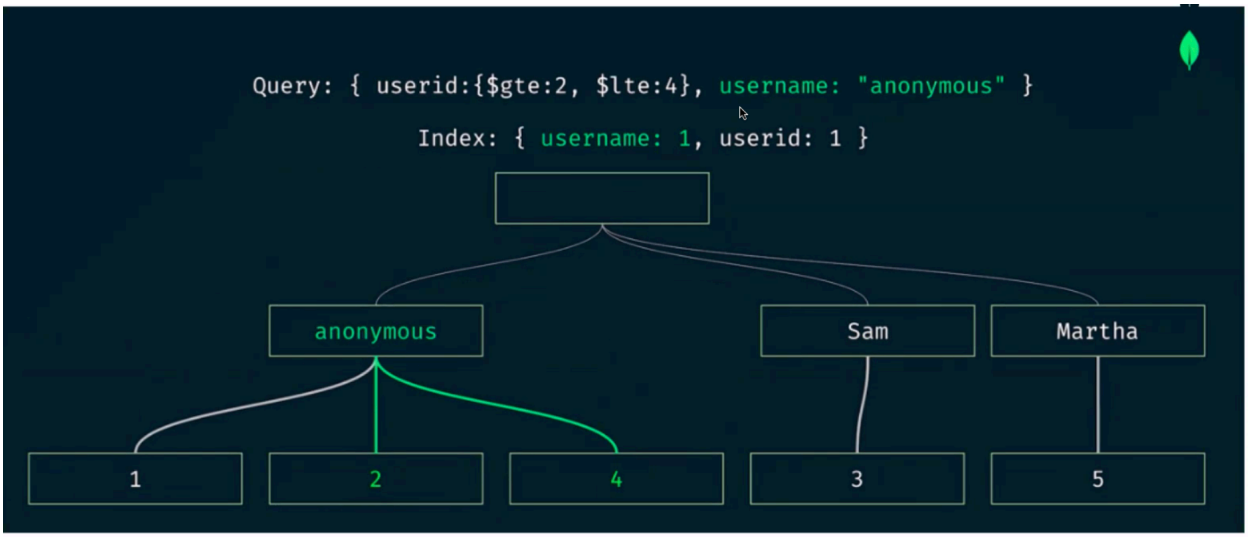
Then **S**ort and **R**ange

- Sorts are much more expensive than range queries when no index is used
- The directions matter when doing range queries

What use cases would the following Compound Index support?

```
db.orders.createIndex({order_status:1, postcode:1, order_date:-1, number_of_items: -1})
```





In Production: Rolling Build

In production, many maintenance tasks are done in a Rolling Manner

Rolling updates are fast with minimal impact on production.

- Can include creating a new index.
- Change performed on temporarily offline secondary
- Then secondary added back to the replica set
- Secondary catches up using the transaction log

In Production: Hidden Indexes

Hidden indexes allow to prevent the DB using an index without dropping it

- Creating an Index is expensive and takes time
- Dropping an Index no longer we think is no needed could be a risk
- Hiding the index temporarily lets us test if it is OK to drop it

Hidden indexes:

- Are no longer used in any operations like find() or update()
- Are still updated and can be re-enabled at any time
- Still apply unique constraints
- If it is a TTL index, documents will still be removed

Logging

Accessing the Database Logs

The Database log file is stored as a text file on the server host

If hosted in Atlas, use the GUI/API to download

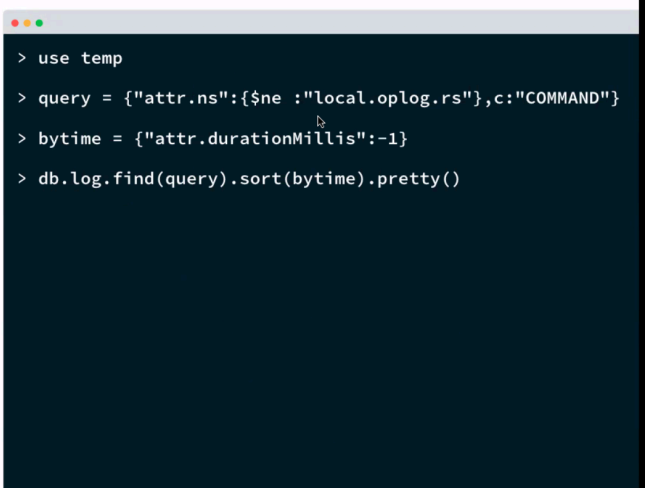
Access the last 1024 lines of the log via getLog

File System	/var/log/mongodb/mongod.log
Hosted in Atlas	Download from the UI or with API
Database	Access with the **getLog** command

Finding slow operations using log

Queries appear in the log if their duration exceeds the default slow operation threshold (100ms)

Can see how long they took, their query plan, and many other values

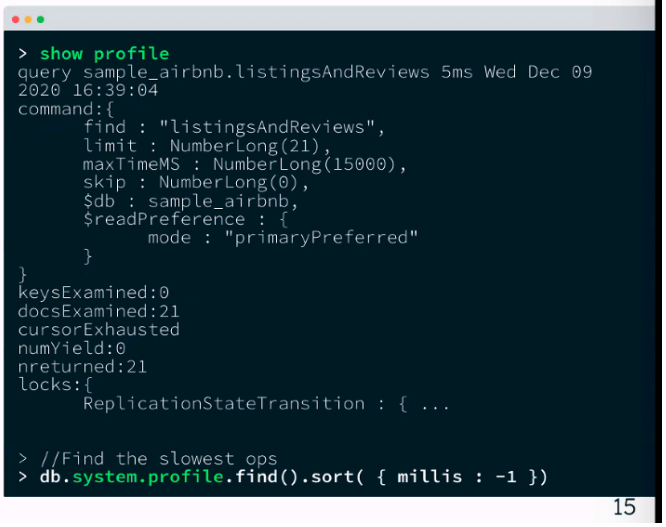


get slow operation

Query Profiler Data

The **show profile** helper in the mongo shell displays the five most recent operations that match the **setProfilingLevel** filters.

We can also query or aggregate the **system.profile** collection in any database



15

normalized cpu 40%-70% ideal

how to monitor mongodb

https://www.mongodb.com/resources/products/capabilities/how-to-monitor-mongodb-and-what-metrics-to-monitor

MongoDB Sizing Script

https://gist.github.com/TravWill-Mongo/32f9738b9a6768e634126a9616ae38d1

https://www.mongodb.com/docs/manual/administration/production-notes/