**Performance:**

**1)Introduction:**

The performance of computer system are depended on varity of factors which includes the performance of the underlying hardware, CPU, disk and memory. One we have chosen our configuration then it gonna be our algorithms which determine the performance and db based application it gonna be the algorithms that satisfy our queries.

There are 2 ways we can impact the latency and throught-put of database quries.

1. Is to add indexes to the collections.
2. Distribute load on multiple servers using sharding.

This week we will talk about performance and to make matter little more interesting, in Mongo 3.0, they have introduced plugable storage engines. Storage Engine: is a software that controls how a data is stored on disk.

We will also discuss, how the choice of storage engine impact performance.

Performance is basically the donmain of the DataBase Adminstrators. But, good developers understand performance and write their application with performace in mind. This will also help to debug the performance related problems within the applications.

**2)Storage Engine : Introduction :**

This week is mostly about indexing and the very end of the week, we gonna loook at sharding, which is distributing database quries across multiple servers. Before going to that, we have understand the concept of Storage engine inside a database. Because in mongo db 3.0, they offer plugable storage engine.

**What is Storage Engine ?**

Storage Engine is a interface between the actual persistance storage (which we called it as Disks) and the database itself(MognoDB, which we mean MongoDb server). MongoDb server talks to the data storage through Storage Engine.

As programmers, we use Java driver(any other varient) to communicate with the mongoDb server using wired protocal and when mognoDb server wants to do CRUD operations then it talks to Storage Engine, which will then talks to the disks. All the different structures that holds the documents, indexes and metadata invoving documents are stored into persistence storage by this storage engine.

Storage Engine may decide to use memory(RAM) to optimise the process, i.e actually disks are very slow, so the idea of database is to store stuff persistently. So the storage engine has control over memory on the computer and can decide what to keep on the memory and what to take out of memory and what to persiste into disks and when.

Now, MongoDb offer a plugable storage Engine architecture where we can use more than one. The idea is based on the requirements we can plug in such Storage Engine with those performance characteristics.

There are two main storage engines that ship on with MongoDb.

1. MMAP: This is the default, when we start the mongoD without any options and this is the storage engine for mongo for a while.
2. Wired Tiger:2014, mognoDb acquired WiredTiger

**What a storage engine is not(does not handle)?**

1. If we have a bunch of mongo servers all running on a clustor, then the storage engine does not effect the communication between these mongoDB servers.
2. The storage engine does not affect the API that the db presents to the programmers. But there gonna be some difference in the performance based on the Storage Engine.

**Quiz:**

Bottom of Form

The storage engine directly determines which of the following? Check all that apply.

* Format of indexes
* Architecture of a cluster
* The data file format
* The wire protocol for the drivers

**3) MMAP Storage Engine:**

It is called as MMAPV1. It’s the original storage engine of mongoDb. And it uses the MMAP system call under the covers in order to implement storage management.

**MMAP system Calls**: In a unix machine, if do “man mmap”. It show the documentation that says “allocate memory, or map files or devices into memory”.

Mongo Db a place to keep the documents and it puts the documents inside files. And to do that, it initially allocate, let’s say, a large file i.e 100gb file on disk. And this may or may not be physically contiguous on the actual disk, because there are some algorithms that occur beneath that layer that control the actual allocation of space on a disk. But for the standpoint it is 100gb continious file.

If mongoDb calls the MMAP system call, it can map this 100gb file, into 100gb of virtual memory. Not to do that, of course, we have to be on a 64 bit machine, bcz we could never get a 100gb of virtual memory space on 32 bit machine. We will be limited to 3-4 gb.

All these page sized. So pages on an operating system are either 4k or 16k large. And the operating system will decide, what can fit in memory(all green). And so when we go to read a document, if it hits a page that’s in memory, then you get it. And if it hits the page that not in memory, then the OS has to bring it from disk to memory before we can access it. And that;s the basics of the way the MMAP storage engine works.

Now, MMAP storage engine offers

1) **collection level concurency or collection level locking** . Each collection in mongoDb is its own file, if you look in data/db. So what that means is that if you have 2 different Operation going on the same time and if they are on the same collection. Then one’s going to have to wait for the other one if they are write because it’s multiple reader, single writer lock that it takes. So only one write can happen to a particular collection.

2) **In place updates:** so if a document is sitting a page and we do an update to it, then this will try to update it right in place. And if can’t be update it, then what it will do is, it will it as a whole and then it will move it somewhere else i.e to other page in the memory with more memory.

3) **Power of 2 sizes:**  In order to avoid the moving of documents while updating the docs, we use this while allocation of the initial size to the docs. What it mean is, if we are trying to create a 3 byte document, then we are going to get a 4 bytes, similary 7-8 and 19-32 bytes. In this way, its more likely that we can grow the docs a little bit and that space that opens up, that we can re-use it more easily.

This is all about MMAP storage engine. But notice that, OS is what makes the decision about what is in memory versus is in disk, we can’t do much in controlling that. OS are pretty smart about managing memory, so db does not involve in the decision of what winds up in physical memory versus what winds up in disks.

**Virtual Memory?????**

Lecture Notes

In this video, Andrew talks about [Power of Two Sized Allocations](http://docs.mongodb.org/manual/core/storage/#power-of-2-allocation), and uses an example that has record spaces allocated for as few as 4 bytes; in fact, the minimum record space in MongoDB 3.0 is 32 bytes.

## Quiz:

Which of the following statements about the MMAPv1 storage engine are true? Check all that apply.

* MMAPv1 offers document-level locking
* MMAPv1 automatically allocates power-of-two-sized documents when new documents are inserted
* MMAPv1 is built on top of the mmap system call that maps files into memory
* MongoDB manages the memory used by each mapped file, deciding which parts to swap to disk.

**4) Wired Tiger:**

This storage is not turned on by default inside the mongoDB in 3.0, but it offers some interesting features, and for a lot of workloads, it is faster.

* Document Level Concurrency: It is not called as document level locking because it actually a lock free implementation which has a optimistic concurrency model where the storage engine assumes that two writes are not going to be to the same document, and if they are to the same document, then one of those writes is unwound and has to try again, and its invisible to the actual application. But we do get the document level concurrency versus the collection level concurrency and that’s a huge win.
* Compression: Allow compression of document of the data and indexes. Wired tiger itself manages the memory, that is used to access the files. So the file is brought in pages, and the pages can be of varying sizes. Wired tiger decides which blocks to keep in memory and which block to moved back to disks. As wired tiger manages memory that wired tiger for instance, compress.
* **No InPlace updates:**  For every update, the page is rewritten the document in a new place. This is how they handle Document level concurrency. So overall its often faster.

To start a mongoDb with wiredTiger then

mongoD –dbpath WT –storageEngine wiredTiger

WiredTiger can’t open files created MMAP v1. So we have to create a separate db folder, passes as a parameter –dbpath.

Db.foo.insert({“apple” : “ball”})

Db.foo.stats()

This will have a key says wiredTiger.

## Quiz:

**Which of the following are features of the WiredTiger storage engine?**

* In-place update of documents.
* Power-of-two document padding
* Document-level concurrency
* Compression
* Turbocharged(something related to car not for wired tiger).

**5) Indexes :**

In this class, we discuss about indexes and impact of indexes on DB performance.

Let us assume a bunch of documents and documents might have the format the form shown here. The collection is a set of documents which are stored in a arbitrary order. And whether you’re talking about NMAP v1 and Wired Tiger, there might be no particular order to the documents on the disks.

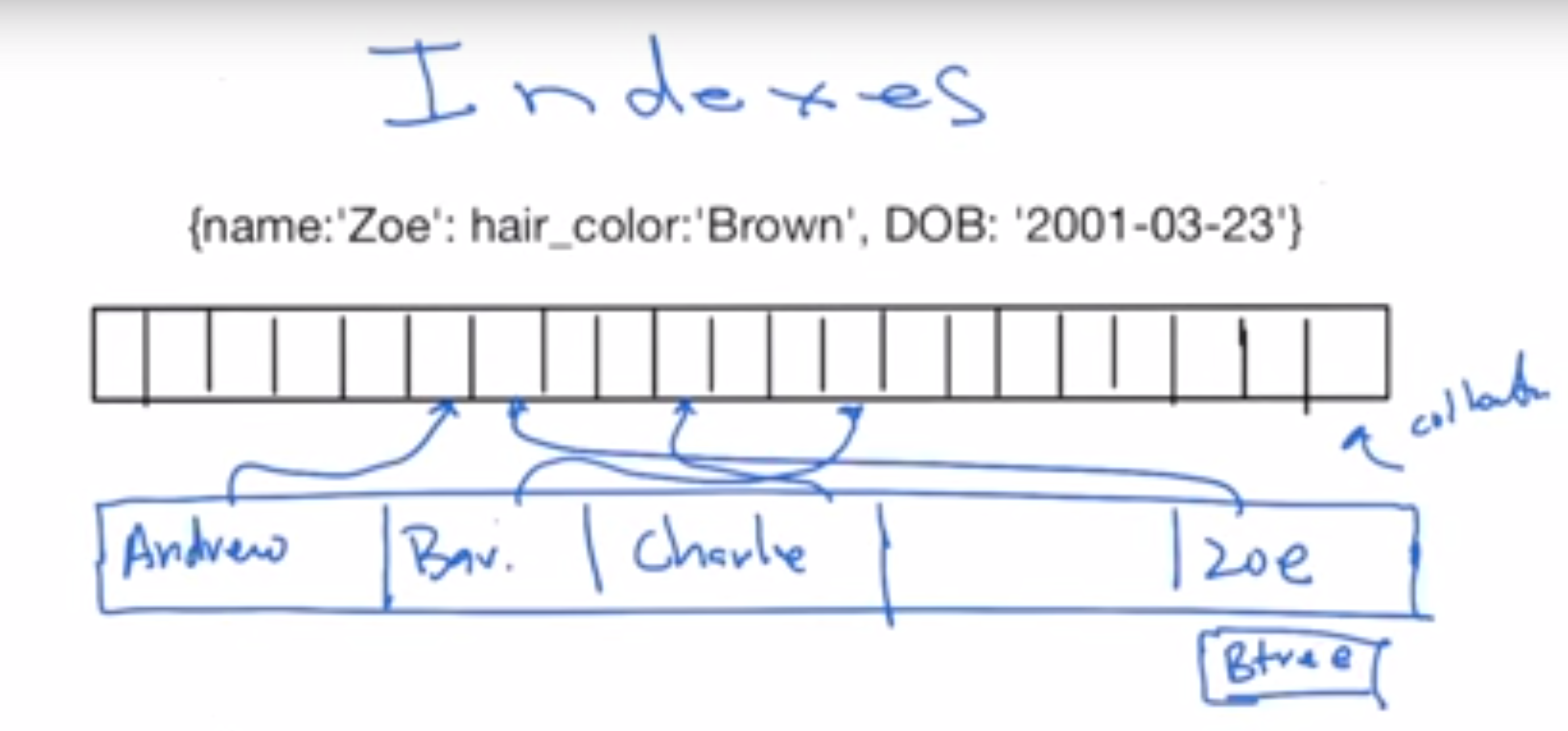
Now, if there is no index and we wanted to find all the documents where, lets say, name was zoe, we would need to scan every document in the collection. And there could be millions of those and this collecion scan or relational scan(Called in relational system) is just death to performance. It is probabily the single greatest factor whether or not your queries are going to perform well, more important than the speed of CPU, memory is whether or not we can use some sort of indexing to avoid looking at the entire collecion.

**What is an index?**

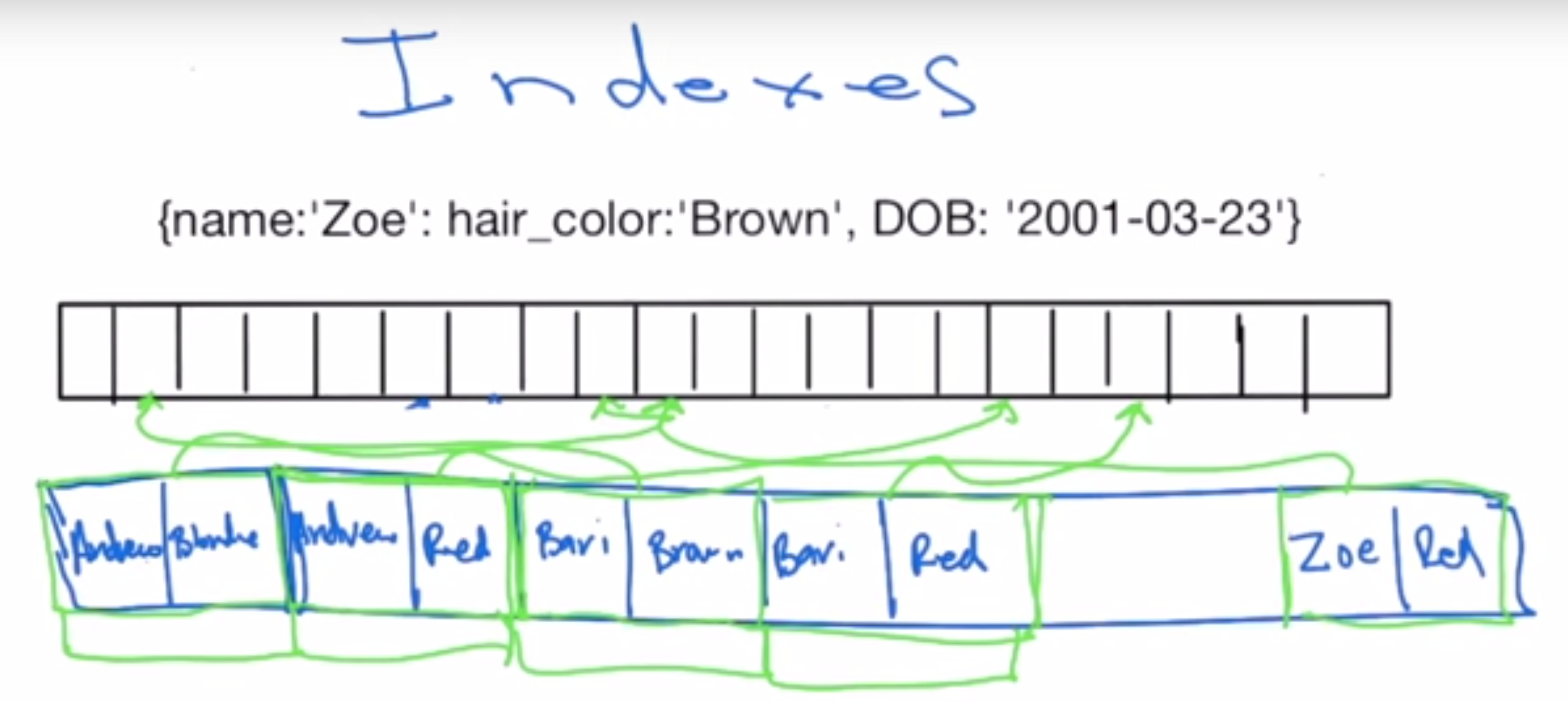
An index is an ordered set of things. So imagine that we have a index on, lets say the name. Then the index of the name is a ordered list of things that are order in some particular order like dictionary order. And each of these index has a pointer to the physical records i.e documents on the disks. So the nice thing if something is ordered is that we can search it easily. Bcz it was actually a linear list like this (which is not in a typical Db) and we use binary search to search the records and it would take you log(2) of number of items.

In real db’s or mongo DB, the actual way these indexes are structured are called B-Tree.

We are going to indexes on the fields that we are expecting to query on because that gonna make querying much faster.



we will not only search on only name, but also include other fields like hair color. The index on name and hair color would be represented as (name,hair-color).



These documents are in the order of name, hair-color.

* let us consider a index which include columns a,b,c then it is represented as (a,b,c). Then searching will be fine if the search is primarily on a but if the search is primarily on any other field like b,c then it will be bit difficult.
* Indexing is not free bcz when where we change the document that effects the index then we have to update the index i.e eventually we have to write it on memory and then on the disks. Indexes are not represented linearly they are represented a B-Tree and maintaining of B-Tree will takes time. So has a result, if we have a collecion and we have indexes on the collection, and then if the writes effects the index, the writes wil will be slower then if there are no indexes.
* Indexes actually slow down your writes. But ur reads will be much faster.
* One tactic is that, if we have a huge document to be written to the BD, in such a case we have to first write the docs and then add indexes to it. This will reduce the time in indexing while writing.
* As indexing slow down, we can’t use indexes on each and every field of the document and this will also makes us use a lot of disks space.

**Lecture Notes:**

In this video, Andrew mentions that while reads are much faster with indexes, writes to a document will happen slower. This is true, but it's worth noting a caveat here: combination operations, such as update and deletion operations, where you find the document you want and then perform a write, will benefit from the index when you're performing the query stage, and then may be slowed by the index when you perform the write. Usually you're still better off having an index, but there are some special cases where this may not be true.

He mentions that indexes in mongodb are in btrees. This is true for MMAP (and therefore for MongoDB prior to 3.0), but it does depend on your storage engine. For example, when you are using WiredTiger, as of MongoDB 3.0, indexes are implemented in b+trees. Again, you can find details in Wikipedia (links provided here for your convenience).

## Quiz:

Which optimization will typically have the greatest impact on the performance of a database.?

* Adding more memory so that the working set fits in memory.
* Adding a faster drive so that operations that hit disk will happen more quickly.
* Replacing your CPU with a faster one (say one 2x as fast)
* Adding appropriate indexes on large collections so that only a small percentage of queries need to scan the collection.

**6) Creating Indexes:**

**Explain() :** This command will show the secrets of what database is doing while executing the query. explain

command runs on top of a collection.

db.students.explain().find( { students\_id : 5 } )

we can check this for update or any othere command. If we look at the output of explain, the secret is at “winningplan” section.

**createIndex():**  1 is ascending and -1 is decending. On the first run this will take time.

db.students.createIndex( { fieldName : order }) :: ex::db.students.createIndex( { students\_id : 1 }) that is 1 is ascending.

**CompoundIndex:**

db.students.createIndex( { students\_id : 1 , class\_id : -1})

the ordering of indexes does not matter for querying but at the time of sorting, it will be helpful.

## 7) Discovering (and deleting) indexes:

## getIndexes (): we have seen, how to create index on a collection. How would you discover what indexes are already on the collection?. Well we can do that with the get indexes command.

## db.students.getIndexes();

## this command will return a document which is an array of indexes. By default, every collection has a index on “\_id” and we cannot delete this index at all. And the array includes rest other indexes on the collecions .

## The getIndexes is a call that you use in MongoDB 3.0 and will work on both wiredTiger and MMAP v1. And in earlier versions of the database and we could look at a special collection system that indexes to see what indexes existed. That does not work in WiredTiger.

## dropIndex(): To get rid of a index on a collection, we need this.

## db.students.dropIndex( { students\_id : 1 })

## The argument should be the same as we have used while creating the index.

## 8) Multi-Key Indexes:

## Indexes on Arrays are called multi-Key Indexes. We can create a index on array by using createIndex method similar to non array filed. This will create index for all the elements in the array.

## We can also create index with the combination of “array fields” and “Non array fields”.

## For example, let us consider the documents:

## {

## name: ‘pranith’,

## tags : [‘reading’,’playing’,’ golf’] ,

## color:’red’,

## location : [‘bangalore’,’warangal’]

## }

## we can create index on (tags).

## Also we can create index in combination (tags, color), (tags, location) etc.

## 🡪 MultiKey indexes will become multikey index when a database realizes that the there is a document with an array as one the keys of the index.

## 🡪 There are some restrictions on multi-key index. In particular, we can’t have two items of a compound index can’t be arrays. For instance (tags, location) index is not possible, if there exists a document, which has array element for tags and location, because an explosion of index points that create it because it has to create index points of the Cartesian product of the items in the arrays, and mongoDB doesn’t permit that.

## 🡪 while creating a compound index, if there is no document in the collection then the index will be created fine. But later if we try to enter array values for both the keys of compound index then mongoDb will not allow.

## Let us consider the scenarios: Let us assume that we have created index on a and b for the collection ” indexCol”, which is a empty collection.

## Case 1 : db.indexCol.insert( { “a” : 1 , “b” : 1} ) // this will enter fine. At this point, the indexes are normal indexes. They are not multiKey because, there is no document with arrays in a or b.

## Case 2 : db.indexCol.insert( { “a” : 1 , “b” : {1,2,3}} ) // this will enter fine. At this point, the indexes are upgraded to multikey indexes. They are multiKey because, there is one document with arrays in the element b.

## Case 3 : db.indexCol.insert( { “a” : {1,2,3} , “b” : {1,2,3}} ) // mongo will throw error because, multikey does allow on two array indexes.

## Case 4 : db.indexCol.insert( { “a” : {1,2,3} , “b” : 1 } ) // this will enter fine bcs, a is array and be is scalar. Any combination is fine.

## Point To Note: Once the index is upgraded to multi-key , even after all the documents are removed , the indexes will be multi-key only.

## 9) Dot Notation and Multi-Key:

## We can use dot notation to reach deep into a document and index something in a sub-document of the main document. And also doing this with things that are arrays—so combining multi-key with dot notation.

## This is a pretty cool feature of mongoDb and it’s a little tricky to explain how it works and it may not works as we thing it works.

## db.students.createIndex({ “students.score” : 1 });

## The above example create index for score array in a student docs. Use getIndexes to get the indexes after executing. This will take time.

## Let us write a query to find an student doc’s that has a score of greater than 98 for the subject exam.

## The doc structure is like

## {

## name:

## scores : [{

## type:

## score:

## }]

## }

## db.students.find({‘scores’:{$elemMatch:{‘type’:’exam’,score:{‘$gt’, 98}}}});

## $elemMatch: this operator allow us to match the condition for a particular documents.

## The above query will first executes on score because we have already created a index. Once the result is ready, Mongo will scan the result docs and for the match of exam .

## Similar, any query which has some condition on field that are already indexed then mongo will first work on that index data. This will reduce the result set and then later work on the other field criteria.

## 10) Index Creation Option, unique:

## In mongo we can create a unique key index i.e duplicates are not allowed on that key. For this we have to pass an extra document parameter to the createIndex method we used earlier. And after this, we cannot have 2 docs with same key for that fields.

## db.students.createIndex( {‘thing’ : 1},{unique : true} )

## Note: we cannot create unique-key index on the collection, which already has duplicate key in that field. To create it we have to delete the duplicate entries and then we can create it. And also, once the unique-key index is created, then we can’t add duplicate keys for that field. In both the cases, mongo will throw a exception.

## To see, if the index is unique or not, then we can trigger getIndexes(), the document response for this method include a field ‘unique’. If the value of unique is true then it is a unique index. However, the response for \_id does not show the unique key as true but even it is unique index.

## 11) Index Creation, Sparse:

## Sparse Indexes are the indexes that can be used when the index key is missing from some of the documents.

## Let us look at the four documents below : first and second document has a,b,c as elements where 3rd and 4th document has only a,b and c missing.

## 

## And also notice that all the a values and b values are unique and also c values.

## If we wanted to create a unique index on “A” that would be no problem bcz a values are unique in all the docs. Unique index on b is also not a problem.

## But unique index on c is a problem because the value of c in 3rd and 4th documents is null and unique index on c will throw a exception and hence it will violate unique constraint. We should have one single docs with c value as null in it. But multiple null is not allowed for a unique index.

## db.students.createIndex( {‘c’ : 1},{sparse : true} )

## In this case, we can create indexes by specifying the sparse option. Sparse option tell the Db server that it should not index the documents that missing the key. In this case, 3rd and 4th doc are not indexed.

## Note: Sparse indexes are not used for sorting. if we are sorting on a key that is sparse indexed, then the mongo Db server will not use the index of this key. Because, if it uses this index then it will misses some docs which are not indexed as the value is null for those docs. For example, In the above example if we sort by c then it will use the space index because 3rd and 4th are not at all indexed.

## Quiz:

## What are the advantages of a sparse index? Check all that apply.

## The index will be smaller than it would if it were not sparse.

## You can gain greater flexibility with creating Unique indexes.

## The index can be used to sort much more quickly in all cases.

## Your indexes can be multikey only if they are sparse.

## 12) Index Creation, Background:

## Last concept in this topic is if we can create indexes in foreground or background.

|  |  |
| --- | --- |
| Fore-Ground Index creation | Back-Ground Index creation |
| It is the default in mongoDb. |  |
| It is relative fast | Little slower, relatively to the fore ground index creation. |
| It blocks all writers and readers in the DB that has these collections. Even though we have collection level locking in MMAP v1 and Document level concurrency in Wired Tiger, still all the writers are reader is blocked while indexing fore ground. This is not suggested in prod environment. | Don’t block readers and writers. We can have only one background index going at a time. After that next one queue and wait on per a Database level. |

## Basically background is fairly the better option to create index, but however, its still fairly high load.

## The other way of creating index more efficiently is to create the index on a different server than the one that we are using to serve most of the queries. Lets us assume that we have a cluster of mongoDb servers that connected over a network. We can temporarily remove one of the servers and redirect the request to other servers, then create index of data on foreground (as it is fast) on this server. Once the Index creation is done, we will bring back the server online n connectivity to other server and accepts new request.

## Syntax: db.students.createIndex( {‘c’ : 1},{background : true} )

## Lecture Notes: Andrew says he thinks the index creation queues up on a per-database level; with MongoDB 2.4 and later, you can create multiple background indexes in parallel even on the same database.

Beginning in MongoDB 2.6, creating an index in the background on the primary will cause the indexes to be created in the background on secondaries, as well. The secondaries will begin index creation when the primary completes building its index.

https://docs.mongodb.org/manual/core/index-creation/?&\_ga=1.141709563.897967488.1454157262#index-creation-background

## Quiz:

## Which things are true about creating an index in the background in MongoDB. Check all that apply.

* A mongod instance can only build one background index at a time per database.
* Although the database server will continue to take requests, a background index creation still blocks the mongo shell that you are using to create the index.
* Creating an index in the background takes longer than creating it in the foreground
* In Mongo 2.2 and above, indexes are created in the background by default.

## 13) Using Explain:

## Explain is used to find out what the database is doing with the query, how it is executing, what indexes it is using and how many documents in inspected.

## In all the cases we use explain, it doesn’t get the data back from the database. It may do most of the work to do the query, but that doesn’t actually bring data back all the way to the client. It just simulates the situation for the understanding.

## explain can be used from drivers and from your application if we want.

## In MongoDB 3.0, explain functionality has been changed, which is called as explain 2.0 internally.

## The first thing is that it used to be called on cursors. i.e db.foo.find().explain(). The find method would return a cursor and then we used to call explain on it.

## In 3.0, it got changed and the preferred method for using explain is to call on collections. db.foo.explain().find() .

## explain method would return a explainable object and on the explainable object we can run find, update, remove, aggregate, on top of it. Most notably we cant run an “insert” on it. So you cant find out what query optimizer would have done on insert. But, there is nothing much to learn about inserts because it simply puts the data into database and it has up there all the indexes. These is no find portal for a insert, but there is find portion for update, remove etc.

## explainable object also include “help”. On shell, if we execute db.collection.explain.help() , will helps us to figure out which functions can be used with explain object.

## Example :

## >> var exp = db.example.explain() // exp is a explainable object.

## >> exp.help() // give all the methods that can used on explainable objects.

## >>exp.find({a:17, b:13}).sort({b:-1});

## this result includes the parsed query in the json representation which is used internally. And then it shows you the winningPlan, which is the one that is chosen to execute the query, and rejected plan which is rejected. The details include the index and documents parsed.

## >>db.example.explain().find({a:17, b:13}).sort({b:-1}); // give the same result as above.

## But in the earlier version of mongoDB, the same is done as follows.

## >>db.example.find({a:17, b:13}).sort({b:-1}).explain().; // give the same result as above. But does a little different. It first get the cursor with explain set on it.

## Note :: :: So then, why it is changed?

## This is because certain thing like count does not return a cursor. So on calling explain on count, will throw an exception. The new version is expanded on different verities of fuctions.

## Quiz:

## Which of the following are valid ways to find out which index uses a particular query? Check all that apply.

## db.example.find( { a : 1, b : 2 } ).explain()

## db.example.explain().remove( { a : 1, b : 2 } )

## var exp = db.example.explain(); exp.find( { a : 1, b : 2 } )

## curs = db.example.find( { a : 1, b : 2 } ); curs.explain()

## db.example.remove( { a : 1, b : 2 } ).explain()

## db.example.explain().find( { a : 1, b : 2 } )

## 13) Explain: verbosity:

## Explain command can be ran on 3 different modes:

## Query Planner Mode: This is the default mode,

## Executionstats.:

## allPlansExecution:

## There is increasing level of verbosity.

## Query Planner Mode: This is the default mode. It only tells what database has done with indexes while fetching the results. But doesn’t tell us the results of using that indexes are.

## Executionstats: db.example.explain(“executionStats”) to run in executionstats mode. This includes the statistics for the executed query. It include the docs the query returned(nReturned), docs examined, execution time.

## allPlansExecution: the query optimizer, will runs all the possible indexes that could be used , and it runs them in parallel, and then makes a decision about which one is fast. And then it remember, which one is fast for a certain shape of a query and it always that index for the final execution. When we run the explain in this mode, we are asking it to return all the solution and stats for a particular shape of a query. allPlansExecution is an array of all the execution plans possible using different indexes.

## 14) Covered Queries:

## Covered Query is a query itself can be satisfied entirely within the index and hence zero documents are inspected to satisfy the query. Now, if we can satisfy a query entirely from index, that’s going to make the query a lot faster.

## How to find a query is covered or not?

## First create an explainable object and run the query. And if the query returns the result without examining any documents. Then the query is called a covered query.

## This can be achieved if the query asked the fields that are basically the subset of the index and if the index does not include \_id then \_id has to be suppressed. Else MongoDB will examine the documents to find out the other needed fields.

## Example: { \_id :12123412 ,I:1 , J: 1 ,K:1} and index is on I,J,K

## >> db.test.explain().find({I:1,J:1}) // This will return 100 docs but also examines 100 docs because even the index include k value but it need \_id value to show it. So it will show the index.

## >> db.test.explain().find({I:1,J:1},{\_id:0,I:1 , J:1 , K:1}) // This will return 100 docs without examining 100 docs because I,J,K values are asked from the projections and they are available in the index.

## >> db.test.explain().find({I:1,J:1},{\_id:0 }) // This will return 100 docs but also examines 100 docs because even the index include k value but it mongoDB has no idea on the fields available in docs. So it will examine to make sure that there is no extra field. If fields are available then it will show. However, mongo server will examine in all the cases. So when projection asked for only indexed elements then it will be covered.

## 14) When is an index used?

## Let us find out how mongoDB choose an index to satisfy a query.

## 

## Let us image, we have five indexes. When the query comes in, MongoDB looks at the query’s shape. Shape has to do with, what fields are being searched on and additional information, such as is there is a sort. Based on such information, system identify the candidate indexes that it may be able to use in satisfying the query.

## So let us assume, we have a query command and 3 out of 5 indexes are identified as candidates for the queries. MongoDb then create 3 query plans, one each for these indexes and in three parallel threads, issue the query such that each one will use a different index and see which one is able to return results the fastest.

## So visually, we can think of this as a race something like this(red lines). The idea here is that the first query plan, which reached goal state, is the winner.

## So what’s the goal state here?

## query plan returns all the results.

## Certain threshold number of results.

## And also more importantly Mongo DB will store this in the cache for the subsequent quires. If a query with similar query shapes comes to Mongo DB then the same query plan will be used. Now in course of time, the collection changes, the indexes changes, so we don’t want this to necessarily be the index we use for ever. There are several ways the query plan end up being evicted from cache.

## Threshold number of writes(1000 writes as of now)

## Rebuild the index.

## If any index is added or dropped from the collection.

## mongoD process is restarted. This will clear the cache and hence we will loose the query plan and other plans.

## So this is the basic process, which mongoDb uses to figure out which index to use for the queries we submit

## Quiz: When is an Index Used?

Given collection foo with the following index:

db.foo.createIndex( { a : 1, b : 1, c : 1 } )

Which of the following queries will use the index?

Top of Form

a) db.foo.find( { b : 3, c : 4 } )

b) db.foo.find( { c : 1 } ).sort( { a : 1, b : 1 } )

c) db.foo.find({c:1}).sort({a:-1, b:1})

d)  db.foo.find( { a : 3 } )

Bottom of Form

## Answer

## To verify the answer key, and see how each index is used, you can check explain("executionStats").

## The overriding principle, though, is that you must use a left-subset (or "prefix") of the index. For sorting, it must either match the index orientation, or match its reverse orientation, which you can get when the btree is walked backwards.

## 15) How large is your index?

## As with other database, it is very important for us to fit working set into the memory. Working set is the portion of our data that clients are frequently accessing. As you might image, the key component of this is indexes. For performance reasons, its essential that we can fit the entire working set into the memory because going to disks for data is a time consuming process and performance will degrade significantly if, for frequently accessed data, we have to go to disks regularly.

## This is specially true with indexes because if in order to search an index, we have to pull it from disks into memory we will loose a lot of performance benefits of having the indexes in the first place. So it is specially important that the indexes are fit into the memory.

## Stats():To see the size of the indexes we can use the stats method on the collection of interest. So we call the stats method and in the response we can see “totalIndexSize” fields value gives the size. This is also broken down to individual sizes in field “indexSizes” over the collection.

## db.collection.stat();

## totalIndexSize(): This will directly give the size of the indexes for the collection.

## Note: Now in Mongo 3.0, we have wired tiger storage engine, which would support few different types of compressions one of which is prefix compression, allow having smaller indexes. So start the mongo server with wiredTiger and with wiredTigerIndexPrefixCompression option turned on.

## ./mongod storageEngine wiredTiger wiredTigerIndexPrefixCompression true dbpath {filePath}

## Run the stat method on the collection. The size of the indexes will be smaller then the MMAP v1 server index size. However, the compression is done on the cost of CPU and it is really depends on ur data set as to whether or not you can take advantage of something like prefix compression.

## Stats output of wired tiger looks a bit different. In particular we have a wired tiger document that explains the bunch of stats of that particular storage.

## 

## 15) Number of Index entries(Index cardinality)?

## Index cardinality is the number of points are there for each different type of index the mongoDB supports.

## In a regular index, for every single key that you put in the index, there certainly going to be a index point. And in addition, if there is no key, then there’s going to be an index point on null key. So essentially, you get about one to one relative to the number of documents in the collection in terms of index.

## The size of the index is directly proportional to the collection size in terms of its end points to documents.

## In Sparse Index: when a key is missing then we don’t index that particular document in Sparse Index. So index points that could be less or equal to number of documents in collection.

## In MultiKey Index: Multikey Indexes are on arrays. And each elements of the array is indexed. So it could be greater than the number of documents and this could be significantly greater.

## Maintenance of index:All this comes up because indexes need to be maintained. There’s a cost of maintaining them. And if anything cause the index to have to get re-written. For example, let say the document has to moved to some location because the update on document has increased the size of it. In this case, every single index that pointing to this document has to be updated. However, if the key is null then we don’t need to update the sparse index. But if it’s a regular index, then we have to update one index point for sure and if it is multikey index then we may have to update many fields has required.

## Lecture Notes:: In this lecture, we talk about the cost of moving documents, in terms of updating index entries. That cost only exists in the MMAPv1 storage engine. In the WiredTiger storage engine, index entries don't contain pointers to actual disk locations. Instead, in WiredTiger, the indexes contain \_id values. As \_id is immutable, the indexes don't change for document moves, although document moves do require updating the index that translates between \_id values an disk locations.

## 16) Geospatial Indexes

## Geospatial indexes allow us to find thing based on locations. The first model we gonna discuss a 2D model . In a 2d world we have a Cartesian plane, x and y co-ordinate. There are many products that are placed in such place.

## In order to search based on location.

## document need to have some sort of x,y location stored in it. It is basically a array of values of x and y coordinates. ‘Location’:[x,y].

## we have to ensureIndex that these are location and used then in location based searches. This can be done by creating the index as ensureIndex( { “location” : ‘2d’} ) . “2d” is reserved and it tell the mongo that it is 2 dimensional geospatial index. We can also add compound to it. That is not a problem.

## $near: find( {‘location’ : { ‘$near’ : [x,y] }} ) : this will give the result of list of docs that are near to x and y in increasing order of distance.

## Example:

## >>db.store.ensureIndex( { location:’2d’ , type : 1} ) // compound index on location and type.

## >> db.store.getIndexes();

## >>db.stote.find( { ‘location’ : { ‘$near’ : [50,50 ]}}).limt(20)

## 17) Geospatial spherical:

## 18) Text indexes (Full text search Index):

## This is a particular type of index, which will come in handy while searching a text. Let us assume that we have a very large document like some test book in a key called “mytext”.

## If we wanted to search on any word on the document then mongoDb will not return anything because mongoDB searches the entire string when we are searching on strings. So as an alternative, we can puts all the words of complete large test into array and then use $set notation operator to push things into it and search for whether or not the words are included but the that’s pretty tedious.

## Instead we can use Full Text Search index, which will index this entire document and every word much in the way an array is indexed, to allow us to do query into the text, basically applying OR operator looking for one of several words.

## Syntax for indexing:

## db.collection.ensureIndex( {‘fieldname’ : ‘text’}) ; // text is the reserved word that is used to say DB for full text indexing.

## Syntax for querying:

## db.collection.find( { ‘$text’ : { ‘$search’: ‘dog’ } } ) //$text and $search specifies that we are using text index for searching the word “dog”.

## db.collection.find( { ‘$text’ : { ‘$search’: ‘dog moss’ } } ) // result include dog or moss.

## Note: Capitalization makes no difference.

## It does not consider the stop works is, the etc.

## Ranking the Result:

## db.collection.find( { ‘$text’ : { ‘$search’: ‘dog moss’ } }, {score: {$meta : ‘textScore’}} ).sort({score:{$meta:’textScore’}})

## $meta: will get the meta data into result. textScore store the meta data for ranking of the search match criteria. So we are sorting on textscore by getting out by using metadata.

## 19) Efficiency of index use:

## In this chapter, we try to understand the type of thinking that we should have while designing the indexes for our applications.

## The Goal for the indexing is that “The Read/Write operations should be as efficient as possible”.

## But as with so many things, this requires some upfront thinking and some experimentation. To be sure you get the right indexes in place, what you’d really want to do is test your application under some real world workloads and make some adjustments from there.

## But however that is out of this course, In this course we will discuss about the initial thinking that we have to do to get ready for some of the real world testing.

## Selectivity of Index: to what degree for a given query pattern the index is going to minimize the number of records scanned. And we have to consider this in scope of all operations to satisfy a query and some times make some trade offs. So we have to consider how sorts are handled.

## Example: To understand execute of queries, first run the explain command on executionStats. In the result, we have to consider the totalKeysExamined and nReturned. If the difference between the two is very much then we have understand that the index is not so selective and remember selectivity is one of our goal.

## Then analyze the query planner and understand the root cause of the execution. Understand which field indexing would help us to optimize the query and then create index on top it.

## However, we don’t need to create the index but mongo Db provided a way of forcing database to use a particular index using hint(). We don’t need to create index in runtime environment but this is the way of overriding what would be output of the query planner.

## Hint(): we can specify the particular index that we want to use for a query, by specifying it name or its actual shape.

## Find().hind({class\_id : 1 }).explain(“executionStats”); // we have specified the shape of the index for the query.

## Execute the query and check the totalKeysExamined and nReturned , if the count is less then creat e the index. Else try other one similarly.

## Quiz: Efficiency of Index Use

In general, based on the preceding lecture, what is the primary factor that determines how efficiently an index can be used?

1) The number of fields in a compound index

2) Whether we are able to sort using the index

3) The selectivity of the index

Answer :: Selectivity is the primary factor that determines how efficiently an index can be used. Ideally, the index enables us to select only those records required to complete the result set, without the need to scan a substantially larger number of index keys (or documents) in order to complete the query. Selectivity determines how many records any subsequent operations must work with. Fewer records means less execution time.

Bottom of Form

## 20) Efficiency of index use:

## while creating a compound index then we have to consider the qualitative selection first then the range queries. This is not true all the time, but works for most.

## For example: in a student database we have get all student whose grade greater than 75 and class is 5th.

## If index is (grade,class) will take long time but index (class, grade) will be very less.

## If we have a sorting in the query, it is better we keep the sort field next to the quantity select. That will allow sorting in db instead of memory.

## Quiz: Efficiency of Index Use

## In general, which of the following rules of thumb should you keep in mind when building compound indexes? Check all that apply. For this question, use the following definitions:

## equality field: field on which queries will perform an equality test

## sort field: field on which queries will specify a sort

## range field: field on which queries perform a range test

## Options:

## a) Sort fields before equality fields

## b) Equality fields before range fields

## c) Sort fields before range fields

## d) Equality fields before sort fields

## e) Range fields before equality fields

## 21) Logging slow queries.

## We have seen the explain command, and how to use the indexes to optimize the query. But ultimately to debug the performance of your program we have to do some profiling to figure out what is slow inside your programs. There is a profiler built in the mongo-db .

## Before that there is a default facility that will help us and that is mongo automatically logs the slows queries above 100 millisecs to the logs that to mongoD started.

## 22) Logging slow queries.

## The profiler is a most sophisticated facility , it will write entries, documents to system.profile, for any query that take longer than a specified amount of time.

## There are 3 level of profilers:

## level 0 : it means it is off

## level 1: I want to log my slow queries.

## Level 2: I want to log all my queries.

## So why we want to log all out queries?

## This is not for the concern of optimization but while writing a query and executing it from the program, it would be easy to understand the database traffic and what it is doing, if the logs are running. So this is more of a general debugging feature then a performance-debugging feature and we can use that while debugging our own programs.

## But right now we will focus on this level 1, i.e logging slow queries. To start in this mode:

## ./mongod –dpath /usr/., --profile 1 –slowms 2 // this will start the profiler in level 1 and logs the queries which executes more then 2ms.

## To find the stats of the query executed slowly, we have to check in db.system.profile.

## db.system.profile.find()

## In the return list, include the query we also xecuted.

## This is a cap Collection, which means that there’s a fix sized collection and it will recycle the space in the collection after it used it up.

## We can write the quires on profile collection as same as other collection to understand the statistics.

## getProfilingLevel(): to get the profilelevel in which mongoD is running. This is executed on database.

## db.getProfilingLevel()

## getProfilingStatus():(): to get the status of the profile in which mongoD is running. This include the level and slowms options . This is executed on database.

## setProfilingLevel() :to start the profiler, we also used this. It takes two argument, 1st is level and 2nd is slowms option.

## db.setProfilingLevel (1,4);

## set the profile to 0 to stop and 1 or 2 to start and running.