



Indexes and Optimization

DF200 - Optimizing Storage and Retrieval

What are Indexes for

- Speed up queries and updates
- Avoid disk I/O
- Reduce overall computation



- The aim is to reduce the resources required for an operation.
- Indexes enable us to dramatically change the resources we use, i.e., memory or disk.
- The disk should be avoided where possible as it is slower than the memory.

How Do Indexes Work

- Data in an index is ordered.
 - Ordered data can be searched efficiently.
- Values in a MongoDB index point to document **identity**.
 - If a document moves on disk, the indexes don't need to be changed.



- Data in an index is ordered.
 - When looking for specific information, you don't need to look in every document.
 - Indexes are in a binary (BTree) structure, so $O(\log N)$ to lookup.
 - Indexes are traversed by value and then point to a document.
- Values in a MongoDB index point to document **identity**.
 - Identity is a 64-bit value in a hidden field assigned when a document is inserted.
 - Documents in collections are in BTrees ordered by **identity**.
 - It's like an internal primary key.
 - It's not maintained between replicas though
 - You can see them by adding `.showRecordId()` to a cursor
 - If their physical location changes, the index stays the same

MongoDB Index misconceptions

- MongoDB is so fast it doesn't need indexes.
- Every field is automatically indexed.
- NoSQL uses hashes, not indexes

There are some misconceptions about MongoDB indexes:

- MongoDB doesn't use them
- Every field is automatically indexed
- NoSQL uses hashes instead of indexes

When do you use an index

- A good index should support every query or update.
 - Scanning records is very inefficient, even if it is not all of them.
- Who determines the best index
 - The Developer, writing the query!
 - NOT a DBA after the fact



Every query or update should be supported by an index
Developers should be responsible for creating the correct indexing

Types of Indexes

- Single-field indexes
- Compound indexes
- Multikey indexes
- Geospatial indexes
- Text indexes
- Hashed indexes
- Wildcard indexes



Types of indexes to be covered (single-field, compound, multikey, geospatial, text, wildcard)
Some will be familiar as they are the same as other DB vendors such as Oracle

How MongoDB uses and chooses Indexes

- 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
- Plan cache entries are evicted when:
 - Using that index becomes less efficient
 - A new relevant index is added
 - The server is restarted



MongoDB uses an empirical technique to look for the best index, this does not require any collection statistics and is well suited to the simple - single collection type queries MongoDB performs, normally with just one index.

If there are no appropriate indexes then it will run a collection scan.

If there is only one it will use that but test its performance first.

If there are more than one it could use it will 'Race' them

MongoDB keeps a cache of the best index for any given "Shape" of query and how well it performed in the "Race" and if future queries do badly it will have a new "Race"

If new indexes are added that could be used or the server is restarted it will invalidate the cache entry and try again.

We can run commands to see what's in the plan cache if we have a complex issue to debug but normally there will only be one useful index.

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.
- Indexing an Object type does not index all the values separately
 - It indexes the object essentially as a comparable blob.
 - You can use it for range searches against Objects or exact matches
- Indexing an Array is covered later.

- A single-field index is the most basic kind
- Builds a tree of all values pointing to the documents they are in
- It makes looking up a value $O(\log n)$ versus $O(n)$
- Allows range queries and sorting to be performant too
- Range searching against objects means if we have an objects like `{a:1, b:5}` with an index we can create a range query to find all values with `{a:1}` or even `{a : { $gte:1}}` without knowing b. `{ myObj : { $gte: { a:1 }, $lte : { a:1, b: MaxKey() } }}`
- When querying objects both exact matches and ranges rely on the field order being the same, we must check language does not rearrange the field order.

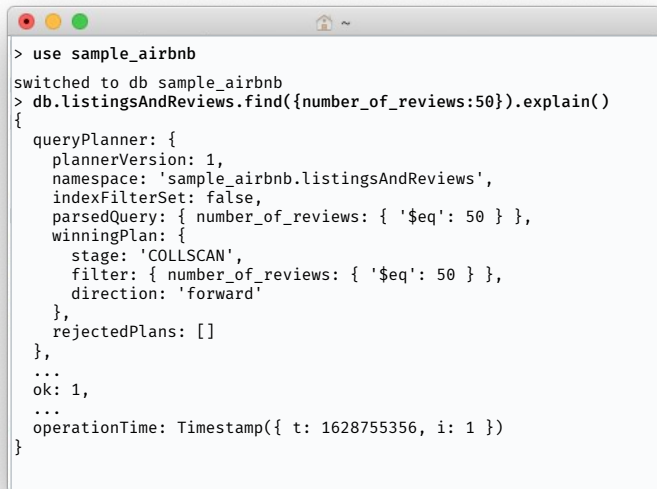
Index Demonstration

explain() on cursor shows
the query mechanics.

COLLSCAN here - short
for Collection Scan

Every document in the
collection looked at.

Very Inefficient



```
> use sample_airbnb
switched to db sample_airbnb
> db.listingsAndReviews.find({number_of_reviews:50}).explain()
{
  queryPlanner: {
    plannerVersion: 1,
    namespace: 'sample_airbnb.listingsAndReviews',
    indexFilterSet: false,
    parsedQuery: { number_of_reviews: { '$eq': 50 } },
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { number_of_reviews: { '$eq': 50 } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  ...,
  ok: 1,
  ...,
  operationTime: Timestamp({ t: 1628755356, i: 1 })
}
```

Queries tend to always be quick on small datasets, so we will use tools to see the implications rather than merely observing time.

The example query has no index, so the DB Engine must look through every document in the collection.

If the collection is too big and isn't cached in RAM, then a huge amount of Disk IO will be consumed, and it will be very slow.

Even if in RAM, it is a lot of data to look through

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"

By default, we only see what the DB Engine intends to do

"executionStats" gives us more detail, such as how long the query takes to run and how much data it has to examine

"allPlansExecution" runs all candidate plans and gathers statistics for comparison

Index Demonstration

We can see this looked at all 5,555 documents, returning 11 in 8 milliseconds

We can create an index to improve this.

Explain plans are complicated with nested stages of processing.

Key metrics are in bold here.

```
> use sample_airbnb
sample_airbnb
> db.listingsAndReviews.find({number_of_reviews:50}).explain(
  "executionStats")
...
  executionStats: {
    executionSuccess: true,
    nReturned: 11,
    executionTimeMillis: 8,
    totalKeysExamined: 0,
    totalDocsExamined: 5555,
    executionStages: {
      stage: "COLLSCAN",
      filter: {
        number_of_reviews: {
          '$eq' : 50
        }
      }
    }
  }
...
```

- **nReturned** is how many documents this stage returns - e.g., the index may narrow to 100 documents, but then an unindexed filter drops that to 10 documents
- **totalKeysExamined** - number of index entries
- **totalDocsExamined** - number of documents read

Ideally, all three of the above are the same number.

Stage shows us whether a collection scan or index was used.

Creating a simple index

In development, we can instantly create an index using `createIndex()`

In production, we need to look at the impact this will have.

There are better ways to do it in production.

A terminal window with a light gray title bar and three colored window control buttons (red, yellow, green) on the left. The terminal text shows a MongoDB command being executed:

```
> db.listingsAndReviews.createIndex({number_of_reviews:1})  
number_of_reviews_1
```

```
> db.listingsAndReviews.createIndex({number_of_reviews:1})  
number_of_reviews_1
```

Creating an index takes an object of fields with an ascending index (1) or descending index (-1) option.

In languages where members of objects aren't ordered, the syntax is a little different.

The return value is the name of the newly created index.

Making indexes in the production system needs to consider the impact on the server, cache, disks, and any locking that may occur.

Rolling index builds are generally preferred in clustered production environments.

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

```
> db.listingsAndReviews.find({number_of_reviews:50}).explain(
"executionStats")
...
executionStats: {
  nReturned: 11,
  executionTimeMillis: 1,
  totalKeysExamined: 11,
  totalDocsExamined: 11,
  executionStages: {
    stage: 'FETCH',
    nReturned: 11,
    works: 12,
    docsExamined: 11,
    ...
    inputStage: {
      stage: 'IXSCAN',
      nReturned: 11,
      works: 12,
      ...
    }
  }
}
```

The example shows improved efficiency of using an index - Note how **totalKeysExamined**, **totalDocsExamined**, and **nReturned** are all the same.
Same behavior as an RDBMS

Explainable Operations

- find()
- aggregate()
- count()
- update()
- remove()
- findAndModify()

The newer style API for example `updateOne()` or `updateMany()` does not allow explain so you need to use `update()` - same functionality

Explain can be run on the operations shown

Note that `updateOne()` or `updateMany()` does not allow explain, so `update()` with options can be used

Applying explain()

- A flag is sent to the server with the operation to say it's an explain command.
- We can set this on the cursor as we do for sort() or limit()
- What if the function we are calling does not return a cursor?
 - For example count() or update()
 - We need to set the flag on the collection object we call with.

```
peoplecoll = db.people
explainpeoplecoll = peoplecoll.explain()
explainpeoplecoll.count()
```

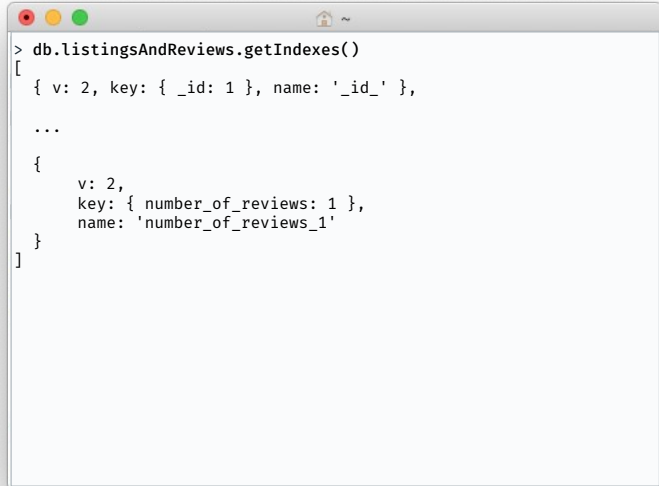


A query is sent to the server once we start requesting from the cursor - so we set a flag on the cursor to request the explain plan rather than the results.

If we don't have a cursor, calling explain() on a collection returns a collection with that flag set.

Listing Indexes

Call `getIndexes()` on a collection to see the index definitions

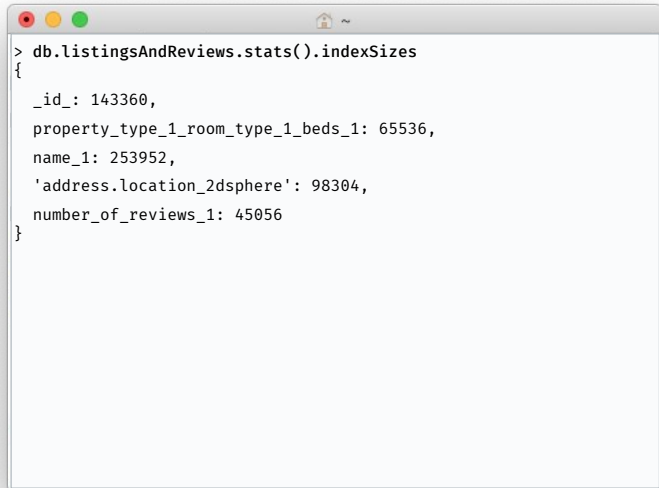
A terminal window with a light gray background and a title bar with red, yellow, and green window control buttons. The terminal shows the command `db.listingsAndReviews.getIndexes()` and its output, which is a JSON array of index definitions. The output includes an index on `_id` and an index on `number_of_reviews`.

```
> db.listingsAndReviews.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  ...
  {
    v: 2,
    key: { number_of_reviews: 1 },
    name: 'number_of_reviews_1'
  }
]
```

`getIndexes()` shows index information (the number of indexes may vary from what you see here on the slide)

Index Sizes

We can call `stats()` method and look at the **indexSizes** key to see how large each index is in **bytes**.

A terminal window with a light gray background and a dark gray title bar. The title bar has three colored window control buttons (red, yellow, green) on the left and a home icon and a tilde symbol on the right. The terminal displays a MongoDB command and its output. The command is `> db.listingsAndReviews.stats().indexSizes`. The output is a JSON object: `{ _id_: 143360, property_type_1_room_type_1_beds_1: 65536, name_1: 253952, 'address.location_2dsphere': 98304, number_of_reviews_1: 45056 }`.

```
> db.listingsAndReviews.stats().indexSizes
{
  _id_: 143360,
  property_type_1_room_type_1_beds_1: 65536,
  name_1: 253952,
  'address.location_2dsphere': 98304,
  number_of_reviews_1: 45056
}
```

Among other options, the scaling factor can be passed to see stats in desired unit.
`db.listingsAndReviews.stats({ scale: 1024 }).indexSizes` shows the index sizes in KB.

Exercise

Use the **sample_airbnb** database and the **listingsAndReviews** collection.

1. Find the name of the host with the most total listings (this is an existing field)
2. Create an index to support the query.
3. Calculate how much more efficient it is now with this index.

Note - this is a tricky question for a number of reasons!



Index Options

- Indexes can enforce a unique constraint.
`db.a.createIndex({custid: 1}, { unique: true})`
 - NULL is a value and so only one record can have a NULL in unique field.
- Sparse Indexes don't index missing fields or nulls.
`db.scores.createIndex({ score: 1 } , { sparse: true })`
 - Sparse Indexes are superseded by Partial Indexes
 - Use `{ field : { $exists : true } }` for your `partialFilterExpression`.
- Partial indexes index a subset of documents based on values.
 - Can greatly reduce index size
`db.orders.createIndex({ customer: 1, store: 1 },
 { partialFilterExpression: { archived: false } })`

Unique constraints can be enforced using indexes.

Partial indexes can be used to only index fields that meet a specified filter expression. Useful for reducing index size.

The Index is only used where the query matches the condition for document to be indexed.

Hashed Indexes

- Hashed Indexes index a 20 byte md5 of the BSON value.
 - They support exact match only.
 - They cannot be used for unique constraints.
 - They can potentially reduce index size if original values are large.
 - Downside: random values in a BTree use excessive resources.

```
db.people.createIndex({ name : "hashed"})
```



Hashed indexing creates performance challenges for range matches etc., so it should be used with caution.

Random values (Hashes or traditional GUIDS) in a BTree maximize the requirement for RAM and Disk/IO and so should be avoided. This is covered later in the course.

Indexes and Performance

- Indexes improve read performance when used.
- Each index adds ~10% overhead
 - Hashed Indexes can add a lot more.
- An index is modified any time a document:
 - Is inserted (applies to all indexes)
 - Is deleted (applies to all indexes)
 - Is updated in such a way that its indexed field changes



Indexes must be applied with careful consideration as they do create overhead when writing data
Unused indexes should be identified and removed

Index Limitations

- You can have up to 64 indexes per collection.
- You should NEVER be anywhere close to that upper bound.
- Write performance will degrade to unusable at somewhere between 20-30.
- 4 is a good number to aim for



The hard limit is 64 indexes per collection, but you should not have anywhere near this number

Use Indexes with Care

- Every query should use an index.
- Every index should be used by a query.
- Indexes require RAM.
- Be mindful about the choice of key.



Depending on the size and available resources, indexes will either be used from disk or cached. You should aim to fit indexes in the cache. Otherwise, performance will be seriously impacted.

Index Prefix Compression

- MongoDB Indexes use a special compressed format
 - Each entry is just 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 mean less RAM required to keep them in RAM



MongoDB uses index prefix compression to reduce the space that indexes consume. Where an entry shares a prefix with a previous entry in the block, it has a pointer to that entry and length, and then the new data. So subsequent identical keys take very little space. This helps optimize cache usage.

Introduction to Multikey Indexes

- A multikey index is an index that has indexed an array.
- An index entry is created on each **unique** value found in an array.
- Multikey indexes can index primitives, documents, or sub-arrays.
- There is nothing special that you need to do to create a multikey index.
- You create them using `createIndex()` just as you would with an ordinary single-field index.
- If any field in the index is ever found to be an array then the index is described as being multikey.



Multikey indexes are indexes on an array.

They are created using the same syntax as normal indexes

You cannot create a compound multikey index if more than one to-be-indexed field of a document is an array.

Multikey Basics

Exercise for the class

- How many records are we inserting?
- How many index entries are there in total for lap_times?
- How many records will the queries find?
- Will they use our index?

```
> use test
> db.race_results.drop()

> db.race_results.createIndex( { "lap_times" : 1 } )

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

// Answer Questions before running these two!

> db.race_results.find( { lap_times : 1 } )

> db.race_results.find( { "lap_times.2" : 3 } )
```

Array of Documents

How Many Documents in total?

For each query:

- How many results?
- Which index, if any, will it use?

```
> 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 } )

// Answer Questions before running the below queries

> db.blog.find( { "comments" : { "name" : "Bob", "rating": 1 } })
> db.blog.find( { "comments" : { "rating" : 1 } } )
> db.blog.find( { "comments.rating" : 1 } )
```

Arrays of Arrays

How Many Documents in total?

For each query:

- How many results?
- Which index, if any, will it use?

```
> db.player.drop()
> db.player.createIndex( { "last_moves" : 1 } )
> db.player.insertMany([
  { "last_moves" : [ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ] ] },
  { "last_moves" : [ [ 3, 4 ], [ 4, 5 ] ] },
  { "last_moves" : [ [ 4, 5 ], [ 5, 6 ] ] },
  { "last_moves" : [ [ 3, 4 ] ] },
  { "last_moves" : [ [ 4, 5 ] ] } ])

// Answer Questions before running below queries

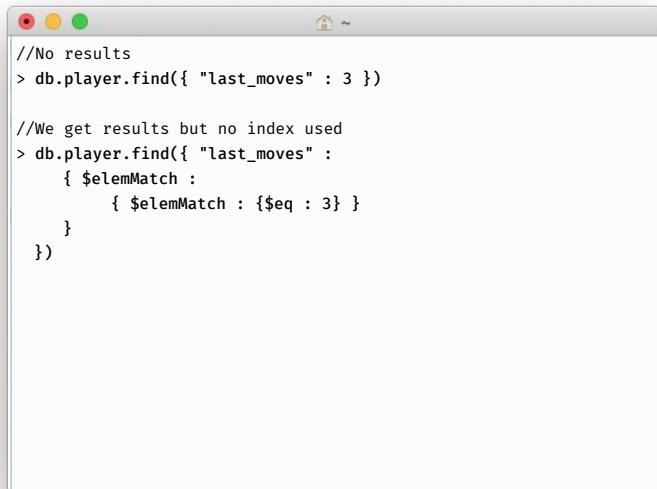
> db.player.find( { "last_moves" : [ 3, 4 ] } )
> db.player.find( { "last_moves" : 3 } )
> db.player.find( { "last_moves.1" : [ 4, 5 ] } )
```

How to do that nested Query

Remember \$elemMatch?

Find where an array member matches this query.

Sadly not indexed - how could you fix that?

A terminal window with a light gray background and a title bar with red, yellow, and green window control buttons. It shows two MongoDB queries. The first query, > db.player.find({ "last_moves" : 3 }), returns //No results. The second query, > db.player.find({ "last_moves" : { \$elemMatch : { \$elemMatch : { \$eq : 3 } } } }), returns //We get results but no index used. The code is as follows:

```
//No results
> db.player.find({ "last_moves" : 3 })

//We get results but no index used
> db.player.find({ "last_moves" :
  { $elemMatch :
    { $elemMatch : { $eq : 3 } }
  }
})
```

On the Previous page we tried to search for a value in an array of arrays - and not only could we not index it but there seemed to be no way to search for it.

If you do need to search in an array of arrays it is possible - but using the very powerful if misunderstood \$elemMatch which returns true or false based on an array member matching a query.

Compound Indexes

- Create an index based on more than one field.
 - They are called Compound Indexes
 - MongoDB normally only uses one index per query
 - Compound indexes are the most common type of indexes
 - They are the same conceptually as used in an RDBMS
- You may use up to 32 fields in a compound index.
- The field order and direction is **very** important.
- You create them like a single field index but with more fields specified

```
db.people.createIndex({lastname:1, firstname:1, score:1})
```

Compound Indexes can support queries that match multiple fields

MongoDB Will not use two indexes together in a query except to support different top level \$or clauses

They should be used instead of creating multiple single indexes

Limit of 32 fields per index

Compound Indexes

- An Index can be used as long as the **first field in index is in the query**.
- Other fields in the Index do not **need** to be in the query.

```
createIndex({country:1,state:1,city:1})  
find({country:"UK",city:"Glasgow"})
```

Uses an Index for country and city but must look at every state in the country, so looks at many index keys.

```
createIndex({country:1,city:1,state:1})
```

A Better Index for this query as can go straight to country and city.



The order of fields in a compound index is important.

In addition to supporting queries that match all the index fields, compound indexes can support queries that match on the prefix of the index fields.

The Order of Fields Matters

- Equality First.
 - In order of selectivity
 - What fields, for a typical query, will filter the most.
 - selectivity != 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 Range or Sort (Usually Sort)
 - Sorts are much more expensive than range queries when no index is used.
 - The directions matter when doing range queries.

Order of sort should usually be Equality, Sort, Range but it can be sometimes be better to have range first.

Putting the most selective fields first , can greatly reduce the quantity of index that is in the working set. If you have a field for "Archived" which is true/false having at the start keeps the archived portion of the index out of RAM.

But be aware that Selectivity and Cardinality are different concepts.

Example: A Simple Message Board

We will look at the indexes needed for a simple Message Board App.

We want to automatically clean up our board and remove some older, low-rated anonymous messages on a regular basis.

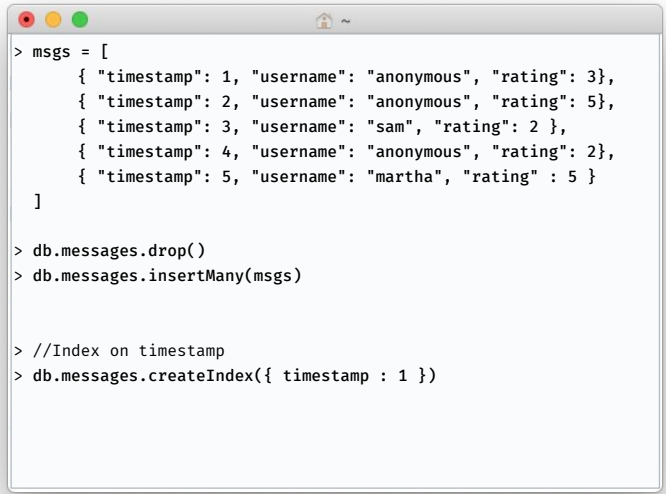
Here is our query requirement:

1. Find all messages in a specified timestamp range.
2. Select for whether the messages are anonymous or not.
3. Sort by rating from lowest to highest.

Message Board Index

Here we create five messages, just the relevant fields.

And an index on timestamp.

A terminal window with a light gray background and a title bar with red, yellow, and green window control buttons. It displays MongoDB commands and their output. The commands include creating an array of five message objects, dropping the 'messages' collection, inserting the array, and creating an index on the 'timestamp' field. The output shows the array of messages and the index creation status.

```
> msgs = [
  { "timestamp": 1, "username": "anonymous", "rating": 3},
  { "timestamp": 2, "username": "anonymous", "rating": 5},
  { "timestamp": 3, "username": "sam", "rating": 2 },
  { "timestamp": 4, "username": "anonymous", "rating": 2},
  { "timestamp": 5, "username": "martha", "rating" : 5 }
]

> db.messages.drop()
> db.messages.insertMany(msgs)

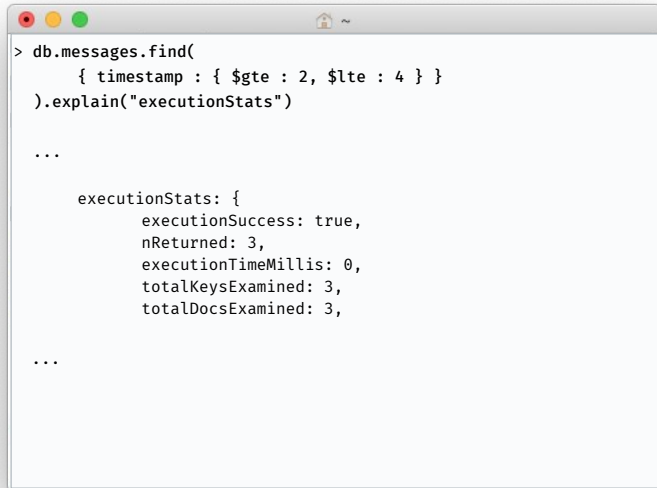
> //Index on timestamp
> db.messages.createIndex({ timestamp : 1 })
```

Example - A Simple Message Board

Message Board Index

Explain shows good performance, but this is not our whole query.

Need to filter for anonymous users

A terminal window with a light gray background and a title bar with red, yellow, and green window control buttons. The terminal displays a MongoDB command and its output. The command is `> db.messages.find({ timestamp : { $gte : 2, $lte : 4 } }).explain("executionStats")`. The output shows an ellipsis followed by an `executionStats` object containing `executionSuccess: true`, `nReturned: 3`, `executionTimeMillis: 0`, `totalKeysExamined: 3`, and `totalDocsExamined: 3`, followed by another ellipsis.

```
> db.messages.find(
  { timestamp : { $gte : 2, $lte : 4 } }
).explain("executionStats")

...

  executionStats: {
    executionSuccess: true,
    nReturned: 3,
    executionTimeMillis: 0,
    totalKeysExamined: 3,
    totalDocsExamined: 3,
  }

...
```

Example - A Simple Message Board

Message Board Index

Not as efficient as it could be:

`totalKeysExamined > nReturned`

What if we add username to the index?

No improvement?



```
> db.messages.find(
  { timestamp: { $gte : 2, $lte : 4 }, username: "anonymous" }
).explain("executionStats")
...
  nReturned: 2,
  executionTimeMillis: 0,
  totalKeysExamined: 3,
  totalDocsExamined: 3,
> db.messages.dropIndex("timestamp_1")
> db.messages.createIndex( { timestamp: 1, username: 1 })
> db.messages.find(
  { timestamp: { $gte : 2, $lte : 4 }, username: "anonymous" }
).explain("executionStats")
...
  nReturned: 2,
  executionTimeMillis: 0,
  totalKeysExamined: 4,
  totalDocsExamined: 2,
```

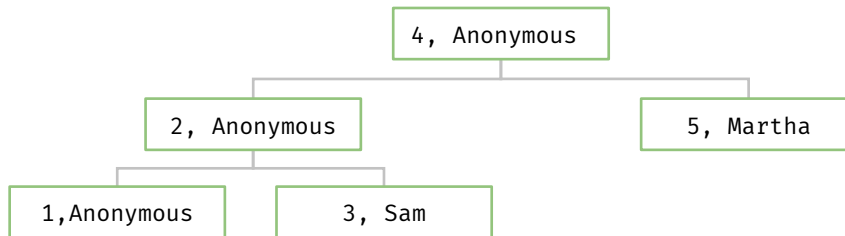
Example - A Simple Message Board

We are dropping the previously created index "timestamp_1" because if we don't do so, the priority would still be given to that index despite of having the newly created index in place.

Index in wrong order

Query: {timestamp:{\$gte:2, \$lte:4}, username:"anonymous"}

Index: { timestamp: 1, username: 1 }



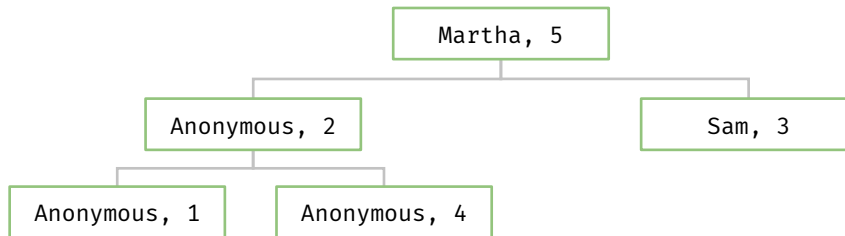
1. Must do range part first as the timestamp is first in the index
2. Start at the first timestamp and check ≥ 2
3. Walk tree Left to Right until timestamp, not ≤ 4 (3 nodes) check each if 'anonymous.'
4. Return only 2 of the three nodes visited (2 and 4)

Index order matters as demonstrated in the example.

Index in correct order

Query: {timestamp:{\$gte:2, \$lte:4}, username:"anonymous"}

Index: { username: 1, timestamp: 1 }

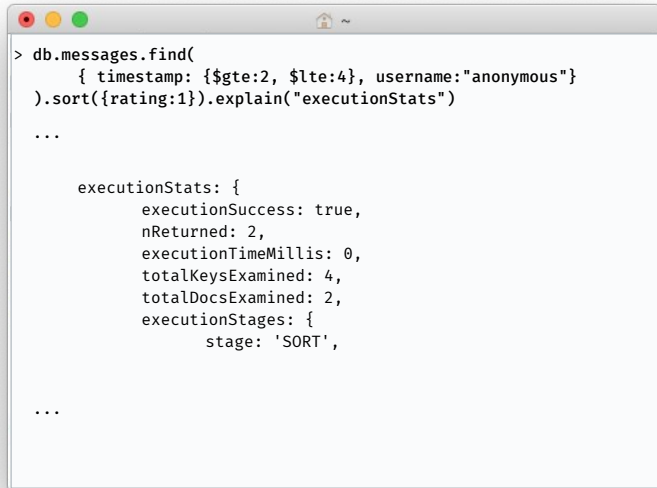


1. Exact Match at start filters down the tree to walk (Just Anonymous).
2. Find first Anonymous where timestamp ≥ 2
3. Walk tree whilst Anonymous & timestamp ≤ 4
4. Visits only two index nodes in total (2 and 4)

Indexing in the correct order will achieve improved results.

What about the sort?

- Sort by rating from lowest to highest.
 - Order in the index must match sort order
 - Otherwise, we need to reorder
 - Here we have an explicit **`SORT`** stage

A terminal window with a light gray background and a title bar with red, yellow, and green window control buttons. The terminal shows a MongoDB command and its execution statistics.

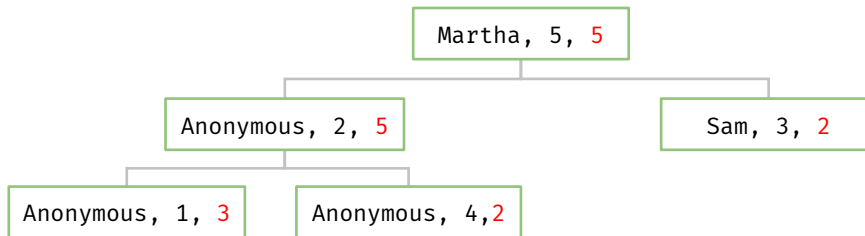
```
> db.messages.find(
  { timestamp: { $gte: 2, $lte: 4 }, username: "anonymous" }
).sort({ rating: 1 }).explain("executionStats")
...

  executionStats: {
    executionSuccess: true,
    nReturned: 2,
    executionTimeMillis: 0,
    totalKeysExamined: 4,
    totalDocsExamined: 2,
    executionStages: {
      stage: 'SORT',
    },
  },
}
```

The index should also cover sorting where possible to prevent sorting in memory.

Index in correct order ?

Query: {timestamp:{\$gte:2, \$lte:4}, username:"anonymous"} Sort: { rating: 1 }
With Index: { username: 1, timestamp: 1 , **rating:1** }

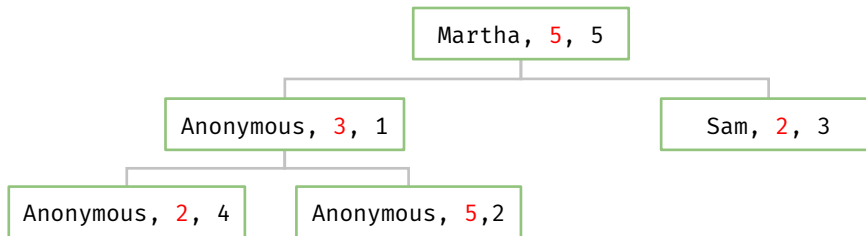


1. Exact Match at start filters down the tree to walk (Just Anonymous).
2. Find first Anonymous where timestamp ≥ 2
3. Walk tree whilst Anonymous & timestamp ≤ 4
4. Visits only two index nodes in total (2 and 4)
5. But results in order 5,2 - so need sorted

Adding a sort field to the index after a range can produce undesired results.

Index in better order ?

Query: {timestamp:{\$gte:2, \$lte:4}, username:"anonymous"} Sort: { rating: 1 }
With Index: { username: 1, **rating: 1**, timestamp: 1 }



1. Exact Match at start filters down the tree to walk (Just Anonymous)
2. Find first Anonymous where timestamp ≥ 2
3. Walk tree L to R until timestamp, not ≤ 4 (3 nodes) check each if 'anonymous.'
4. Return only 2 of the three nodes visited (2 and 4)
5. But results in the correct order

Amending the index field order with sort before range can improve performance significantly. When checking the tree in step 3, one extra node is checked as you need to check one that is 'wrong' to know where the 'right' ones end.

Rules of Compound Indexing

- Equality before range
- Equality before sorting
- Sorting before range



The general rule is Equality, Sort, Range.

But there are cases where Range may be better placed before Sort.

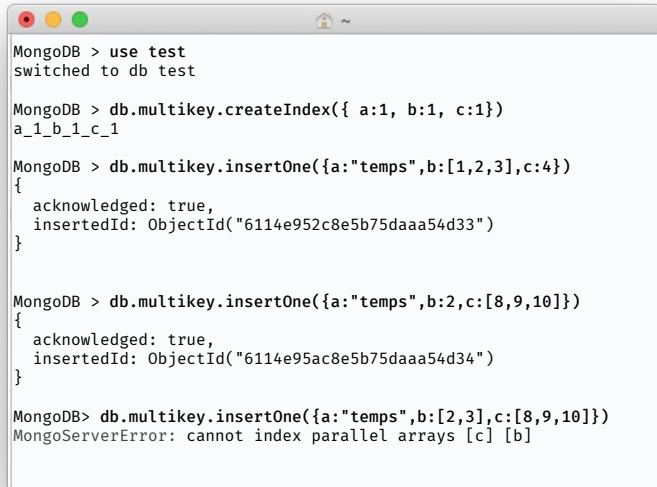
There may even be unusual cases where a Range or Sort should come before equality .

MultiKey Compound Indexes

An Index can be Compound (multiple fields) and MultiKey (Multiple values for a field)

In a Document any non `_id` field can be an array

Indexing two arrays in one document is an error as we would need to store all possible combinations.



```
MongoDB > use test
switched to db test

MongoDB > db.multikey.createIndex({ a:1, b:1, c:1})
a_1_b_1_c_1

MongoDB > db.multikey.insertOne({a:"temps",b:[1,2,3],c:4})
{
  acknowledged: true,
  insertedId: ObjectId("6114e952c8e5b75daaa54d33")
}

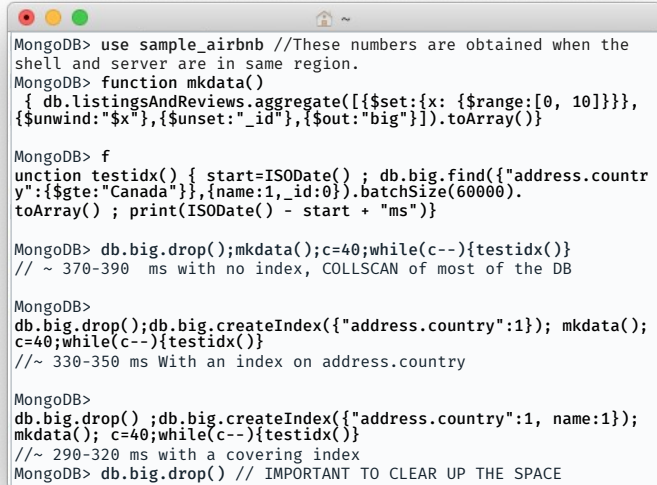
MongoDB > db.multikey.insertOne({a:"temps",b:2,c:[8,9,10]})
{
  acknowledged: true,
  insertedId: ObjectId("6114e95ac8e5b75daaa54d34")
}

MongoDB> db.multikey.insertOne({a:"temps",b:[2,3],c:[8,9,10]})
MongoServerError: cannot index parallel arrays [c] [b]
```

You can have a compound Multikey index, all the fields in the compound index are stored in an index entry for each unique value in the array field.
For any given document - which field is an array in the compound index does not matter.
But only one of the fields in any give compound index can be an array in a single document.
If this was not the case we would need an index entry for every possible combination of values - which might be huge.

Index Covered Queries

- Fetch all data from Index not the Document
- Can be much faster in some cases
- Does have some limitations to be aware of
 - { _id : 0 }
 - no multikey indexes



```
MongoDB> use sample_airbnb //These numbers are obtained when the
shell and server are in same region.
MongoDB> function mkdata()
{ db.listingsAndReviews.aggregate([{$set:{x: {$range:[0, 10]}}},
{$unwind:"$x"},{$unset:"_id"},{$out:"big"}]).toArray()}

MongoDB> f
unction testidx() { start=ISODate() ; db.big.find({"address.countr
y":{"$gte":"Canada"}},{name:1,_id:0}).batchSize(60000).
toArray() ; print(ISODate() - start + "ms")}

MongoDB> db.big.drop();mkdata();c=40;while(c--){testidx()}
// ~ 370-390 ms with no index, COLLSCAN of most of the DB

MongoDB>
db.big.drop();db.big.createIndex({"address.country":1}); mkdata();
c=40;while(c--){testidx()}
//~ 330-350 ms With an index on address.country

MongoDB>
db.big.drop();db.big.createIndex({"address.country":1, name:1});
mkdata(); c=40;while(c--){testidx()}
//~ 290-320 ms with a covering index
MongoDB> db.big.drop() // IMPORTANT TO CLEAR UP THE SPACE
```

If all the fields we need can be found in the index then we don't need to actually fetch the document.

We need to remove `_id` from the projection if it's not in the index we use to query.

We cannot use a multikey index for projection as we don't know from the index entry about position or quantity of values, or even if it's an array versus a scalar value in any given document. Indexes store data in a slightly different format to BSON as all numbers have a common format in the index (and a type) and so need to be converted back to BSON, this takes more processing time.

If we are fetching one or two values from a larger document - index covering is good - if it's most of the fields in a document it's probably better to fetch from the document.

If we need to add extra fields to the index in order to facilitate covering, add them at the end and be aware of the extra storage.

Index-covered queries are the ultimate goal but not always achievable.

Once we reach our data storage limit we cannot create anything else, including an index - so we are making the index first then making the big collection, this collection technically is larger than our storage limit in the free tier but as it is created in an aggregation we are able to generate it.

`toArray()` method on a cursor fetches all the contents - we do have some network overhead here though which is a constant in the total time - without that (for example in an aggregation) the difference is much larger proportionally.

Do remember to drop the big collection after the above commands are executed or you will not be able to perform any more write operations.

Exercise – Compound indexes

Create the best index you can for this query - how efficient can you get it?

```
> query = { amenities: "Waterfront",  
  "bed_type" : { $in : [ "Futon", "Real Bed" ] },  
  first_review : { $lt: ISODate("2018-12-31") },  
  last_review : { $gt : ISODate("2019-02-28") }  
}  
  
> project = { bedrooms:1 , price: 1, _id:0, "address.country":1}  
> order = {bedrooms:-1,price:1}  
> use sample_airbnb  
> db.listingsAndReviews.find(query,project).sort(order)
```



Geospatial Indexing

- Most indexed fields are Ordinal
 - $A < B < C$
 - And so can be sorted.
 - Sorted data can be efficiently searched with a binary chop $O(\log n)$
- Locations - coordinates are not
 - You can sort East \leftrightarrow West OR North \leftrightarrow South
 - But not both
- Therefore they require a totally different indexing concept
 - Geohashes or ZTrees
 - MongoDB uses Geohashes
- MongoDB has a very comprehensive geospatial capability

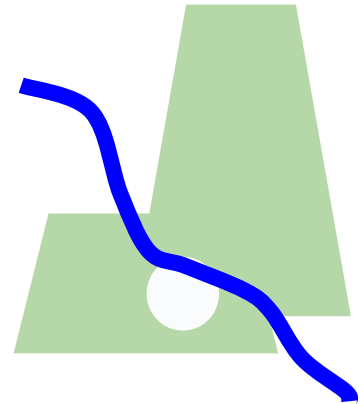


Geospatial Indexing requires a totally different indexing concept.
Geohash transforms 2d coordinate values into 1d key space values, which are then stored in normal MongoDB B-trees.

Geospatial types

- Geospatial data has its own types
 - Points
 - Lines
 - Circles
 - Polygons
 - Groups of the above additive and subtractive
- Written as GeoJSON or as [longitude,latitude]

```
area = {  
  type : "Polygon",  
  coordinates : [  
    [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ],  
    [ [ 2 , 2 ] , [ 3 , 3 ] , [ 4 , 2 ] , [ 2 , 2 ] ]  
  ]  
}
```



Geospatial data has its own types

Geospatial Capabilities

- Geo Indexes quickly determine geometric relationships:
 - All shapes within a certain distance of another shape
 - Whether or not shapes fall within another shape
 - Whether or not two shapes intersect
- Many Use cases
 - Find all bars near here
 - Find all motels along my route
 - Find all counties that are impacted by this hurricane
- Queries can be combined with other predicates
 - Find all motels on my route with a pool.
 - Geo Indexes can be compound with other fields



Geospatial has many use cases in modern applications.

Coordinate spaces

- MongoDB can use Cartesian (2d) or Spherical Geometry (2dsphere)
- Cartesian
 - Simple Lat/Long -90 to 90 and -180 to 180 degrees
 - Only supports coordinate pairs (Array)
 - Does not take into account the curvature of the earth
 - 1 degree of longitude is 66 miles (111km) at the equator
 - 1 degree of longitude is 33 miles (55km) in the UK.
 - Good for small distances and simple
- Spherical 2d
 - Full set of GeoJSON objects
 - Required for larger areas



There are different methods of utilizing geospatial
Cartesian and Spherical

Geo Indexing Example

\$geoNear
\$geoWithin
\$geoIntersects

Sortable by Distance

```
> use sample_weatherdata
> db.data.createIndex({"position":"2dsphere"})
> joburg = {
  "type" : "Point",
  "coordinates" : [ -26.2, 28.0 ]
}

> projection = {position:1,"airTemperature.value":1,_id:0}

> //Distance in metres
> db.data.find({ position: { $geoNear: { $geometry: joburg,
  $maxDistance: 100000}}}, projection)

{ "position" : { "type" : "Point", "coordinates" : [ -26.1, 28.5 ]
}, "airTemperature" : { "value" : 20.5 } }
{ "position" : { "type" : "Point", "coordinates" : [ -26.8, 27.9 ]
}, "airTemperature" : { "value" : 19 } }
{ "position" : { "type" : "Point", "coordinates" : [ -26.9, 27.6 ]
}, "airTemperature" : { "value" : 20 } }
{ "position" : { "type" : "Point", "coordinates" : [ -27,
27.5 ] }, "airTemperature" : { "value" : 20.3 } }
```

\$geoNear, \$geoWithin, and \$geoIntersects are operators for using geospatial functionality within MongoDB

Geo Indexing Exercise

I like the idea of staying at "Ribeira Charming Duplex" in Porto.

It has no pool - find the five properties within 5KM that do

You will need to do more than one query; however, you should write a program (that runs in the shell) that takes the name of a property and finds somewhere nearby with a pool.

```
> use sample_airbnb
> var villaname = "Ribeira Charming Duplex"
> var nearto = db.listingsAndReviews.findOne({name:villaname})
> var position = nearto.address.location
> query = <write your query here>
> db.listingsAndReviews.find(query,{name:1})
```

Make sure you have the right indexes



Time to Live (TTL) Indexes

- Not a special Index, just a flag on an index on a Date
- MongoDB automatically deletes documents using the index based on time
- Background thread in server runs regularly to delete expired documents.

```
> db.t.drop()
> db.t.insertOne({ create_date : new Date(),
user: "bobbyg",session_key: "a95364727282",
cart : [{ sku: "borksocks", quant: 2}]})
> db.t.find()
> //TTL set to auto delete where create_date > 1 minute old.
> db.t.createIndex({"create_date": 1},{expireAfterSeconds: 60 })

> for(x=0;x<10;x++) { print(db.t.count()) ; sleep(10000) }
```

TTL indexes allow data to be deleted when it expires
The expiration period is set when creating the index

Time to Live (TTL) Indexes

- Alternative way to use TTL Indexes
 - Put the date you want it deleted in a field (expire-on)
 - Add a TTL with expireAfterSeconds set to 0
- Watch out of unplanned write load
 - Better to write your own programmatic data cleaner.
 - Schedule using a framework like cron/scheduler/Atlas



TTL indexes should be used with caution as restoring deleted data can be a huge pain or even not possible if no backups exist.

These should be widely communicated to ensure it comes as no surprise when this happens.

Be careful not to delete huge amounts of data in production.

Native text Indexes

- Superseded in Atlas by Lucene - but relevant to on-premise
- Indexes tokens (words, etc.) used in string fields.
 - It allows you to search for 'contains.'
- Algorithm
 - Split text fields into a list of words.
 - Drop language-specific stop words ("the", "an", "a", "and").
 - Apply language-specific suffix stemming
 - "running", "runs", "runner" all become "run".
 - Take the set of stemmed words and make a multikey index.
- MongoDB supports text search for several western languages.
- Queries are OR by default.
- Can be compound indexes



Text indexes use an algorithm to split text fields into words and make them available for search using contains.

Native text Indexes - limits

- Logical AND queries can be performed by putting required terms in quotes
 - This can also be used for required phrases "ocean drive"
 - This applied as secondary filter to an OR query for all terms
 - This makes AND and Phrase queries very inefficient.
- No fuzzy matching
- Many index entries per document (slow to update)
- No wildcard searching
- Indexes are smaller than Lucene though

"ocean drive" - Ocean OR Drive

"\"ocean\" \"drive\"" - Ocean AND Drive

"\"ocean drive\"" - Ocean and Drive as two consecutive words with one space



Text indexes are limited, and Lucene should be used instead on Atlas

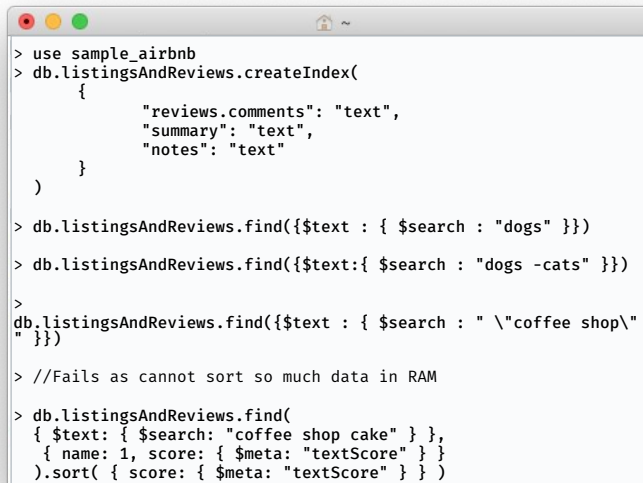
Text Index Example

Only one text index per collection, so create on multiple fields.

Index all fields with

```
db.collection.createIndex(  
  { "$**": "text" }  
)
```

Use \$meta and \$sort to order if results are small.



```
> use sample_airbnb  
> db.listingsAndReviews.createIndex(  
  {  
    "reviews.comments": "text",  
    "summary": "text",  
    "notes": "text"  
  }  
)  
  
> db.listingsAndReviews.find({$text : { $search : "dogs" }})  
> db.listingsAndReviews.find({$text:{ $search : "dogs -cats" }})  
  
>  
db.listingsAndReviews.find({$text : { $search : " \"coffee shop\" "  
" }})  
  
> //Fails as cannot sort so much data in RAM  
  
> db.listingsAndReviews.find(  
  { $text: { $search: "coffee shop cake" } },  
  { name: 1, score: { $meta: "textScore" } }  
) .sort( { score: { $meta: "textScore" } } )
```

You can use -term as in "dogs -cats" to say do NOT return results with this term.

Wildcard Indexes

- Dynamic schema means it's hard to index **all** fields
 - There are alternative schemas that we will see later
- Wildcard indexes index all fields, or a subtree
- Index Entries treat the fieldpath as the first value
 - Normal index on "user.firstname" Index contains "Job","Joe","John" as keys.
 - Wildcard Index on "user" contains the fieldnames in the keys:
"firstname.Job", "firstname.Joe", "firstname.John",
and any other other fields in user e.g.
"lastname.Adams", "lastname.Jones", "lastname.Melville"
 - What does that do to the index size?
 - What are the performance and hardware implications?
 - This feature is easy to overuse/misuse.
 - Indexing correctly matters - not just "index everything"

Wildcard indexes are a new feature intended for dynamic schemas

They should not be used as a shortcut to index static schemas.

Even if a single wildcard index could support multiple query fields, MongoDB can use the wildcard index to support only one of the query fields. All remaining fields are resolved without an index.

More Index usage tips (or hints)

- We can use 'hint' to tell MongoDB what Index to use in find.
- We can use an index for a Regular Expression match,
 - If anchored at the start { name: /^Joh/ }
 - But beware of the case, and also be aware that's a range query >= "Joh" and < "Joi"
- We can set a collation order for Indexes (No diacritics, case insensitive, etc.)
 - Preferred collation can be specified for collection and views.
 - Indexes and Queries can specify what collation use if not the default.



Note: A \$regex implementation is not collation-aware.

Even more Index usage tips

- Indexes have ~10% overhead on writing per index entry (beware of multikey)
- Accessing Index should ideally be from RAM based cache, if we don't have it cache, it will fetch data from disk.
- We can use `$indexStats()` aggregation to see how much each index is used.
- More Indexes generally means more RAM required.
- Remove indexes that aren't used by any queries to save CPU and RAM.
- If one index is a prefix of another it is redundant { a: 1} and {a:1,b:1}

- Indexes are mostly just like RDBMS indexes - but queries are simpler.

Indexing should be used effectively to improve query performance, but over-indexing or indexing incorrectly can have an adverse effect.

Indexes do NOT need to be held entirely in RAM - like collections the database will cache in RAM parts it accesses often and evict those it doesn't.

The index exists on DISK with a cache in RAM of parts accesses recently - we want to access something from RAM so we need to design the indexes so infrequently accessed data does not get cached (or add more RAM). Often this is based on a date value in the index.

Indexes in Production

To build an index, one has to read whole data, which may be much bigger than the RAM size in the production. So, there were two ways of creating an index in the previous versions: Foreground and Background.

Foreground Index (MongoDB Pre 4.2)

Foreground index builds were fast but required blocking all read-write access to the parent database of the collection being indexed for the duration of the build.

Background Index (MongoDB Pre 4.2)

Background index builds were slower and had less efficient results but allowed read-write access to the database and its collections during the build process.

Hybrid Index (MongoDB 4.2+)

Does not lock the server and builds quickly. It is in the newer version of MongoDB.



These were two way of creating index in previous version of MongoDB

Indexes in Production: Rolling Build

- In production, many maintenance tasks are done in a Rolling Manner
 - Rolling updates are fast with minimal impact on production.
 - This 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.

- It starts building an index on the secondary members one at a time, considering it as a standalone member
- It requires at least one replica set election
- Steps to create Rolling Index
 - Remove any one secondary server from the cluster
 - Start secondary as standalone
 - Build the index foreground/hybrid
 - Add secondary back to the cluster with index already created
 - Repeat for all other secondary servers
 - Step down the primary and one of secondary become primary
- Atlas/cloud manager/ops manager do it for you automatically

For workloads that cannot tolerate performance decrease due to index builds, consider using the procedure provided above, to build indexes in a rolling fashion.

Indexes in Production: Hidden Indexes

- Hidden indexes allow you to prevent the database using an index without dropping it.
- Creating an Index is expensive and takes time.
- Dropping an Index you no longer think you need could be a risk
 - Hiding it lets you test if it is OK to drop it.
- If you Hide an index.
 - It is no longer used in any operations like `find()` or `update()`
 - It is still updated and can be re-enabled at any time.
 - It still applies unique constraints
 - If it is a TTL index then documents will still be removed.



Hidden indexes are not available before version 4.4

Recap

- Indexes improve efficiency and speed of reads
- Every query should use an index
- Compound indexes are the most used ones in MongoDB
- The order in a compound index is very important.
- MongoDB can `explain()` how an operation is being indexed.





Answers

Exercise Answers

Exercise – Indexing

Find the name of the host with the most total listings

```
db.listingsAndReviews.find({}, {"host.host_total_listings_count":1,  
"host.host_name":1}).sort({"host.host_total_listings_count":-1}).limit(1)  
  
{ "_id" : "12902610", "host" : { "host_name" : "Sonder", "host_total_listings_count" : 1198 } }
```

Now create an index to support this and show somehow how much more efficient it is.

```
"executionTimeMillis" : 11,  
"totalDocsExamined" : 5555,  
"works" : 5559,
```

```
db.listingsAndReviews.createIndex({"host.host_total_listings_count": 1})
```

```
"executionTimeMillis" : 1,  
"totalKeysExamined" : 1,  
"totalDocsExamined" : 1,  
"works" : 2,
```



Note: In the first query where we apply a sort and limit, MongoDB doesn't guarantee consistent return of the same document especially when there are simultaneous writes happening. It is advisable to include the `_id` field in sort if consistency is required.

Exercise – Compound Indexes

Create the best index you can for this query - how efficient can you get it?

```
query = { amenities: "Waterfront",
  "bed_type" : { $in : [ "Futon", "Real Bed" ] },
  first_review : { $lt: ISODate("2018-12-31") },
  last_review : { $gt : ISODate("2019-02-28") }
}
project = { bedrooms:1 , price: 1, _id:0, "address.country":1}
order = {bedrooms:-1,price:1}
db.listingsAndReviews.find(query,project).sort(order)

//One Answer - you may do better!
db.listingsAndReviews.createIndex({amenities:1,bedrooms:-1,price:1,
bed_type:1,first_review:1,last_review:1})
```

DOCS_EXAMINED:	13
KEYS_EXAMINED:	117
TIME:	0MS



Exercise Answers

Multikey Basics Answers:

Slide 1 (Simple Arrays): 3 Record

Only 1 Index Entries as only one entry needed for duplicate '3' in third record.

`db.race_results.find({ lap_times : 1 })` - 1 record, will use index

`db.race_results.find({ "lap_times.2" : 3 })` - 1 record, will not use index but if you add `, "lap_time":3` to the query it will at least partially use the index to narrow down records.

Slide 2 (Arrays of Documents): 3 Records,

`db.blog.find({ "comments" : { "name" : "Bob", "rating": 1 } })` - 1, using comments index

`db.blog.find({ "comments" : { "rating" : 1 } })` - 0 - there is no object of exactly that shape, uses comments index

`db.blog.find({ "comments.rating" : 1 })` - 2, using comments.rating index

Slide 3:



Multikey Basics Answers II:

Slide 3 (Arrays of Arrays): 5 Documents total

```
db.player.find( { "last_moves" : [ 3, 4 ] } ) - finds 3, using index
db.player.find( { "last_moves" : 3 } ) - finds 0, using index
db.player.find( { "last_moves.1" : [ 4, 5 ] } ) - finds 1, does not use index
```

Slide 4 (using \$elemMatch to query):

You cannot index an anonymous value in a multi-dimensional array like this, however, if you change the schema to

```
{ "last_moves" : [ {p:[ 1, 2 ]}, {p:[ 2, 3 ]}, {p:[ 3, 4]} ] }
```

Then you could index on {"lastmoves.p":1} as a multikey index nested arrays are indexable as long as they are not anonymous.



Exercise – Geo Indexing

I like the idea of staying at "Ribeira Charming Duplex" in Porto

It has no pool though. Find the five properties within 5KM of it that do

This should be a single bit of code that may do multiple queries

```
db.listingsAndReviews.createIndex({amenities:1, "address.location":"2dsphere"})
db.listingsAndReviews.createIndex({"name":1})

var villaname = "Ribeira Charming Duplex"
var nearto = db.listingsAndReviews.findOne({name:villaname})
var position = nearto.address.location
query = { "amenities" : "Pool" ,
          "address.location" : { $geoNear : { $geometry : position, $maxDistance: 5000 }}}

db.listingsAndReviews.find(query,{name:1})
```

