**5. Indexing- Commands**

Contents

[1. The Problem 1](#_Toc130325915)

[2. The Solution 2](#_Toc130325916)

[3. Sort Uses Index 3](#_Toc130325917)

[4. Indexes in Mongo 3](#_Toc130325918)

[5. Create Index 5](#_Toc130325919)

[6. system.indexes collection 5](#_Toc130325920)

[7. explain() 6](#_Toc130325921)

[8. Multi-Term Query 8](#_Toc130325922)

[9. Comparison 8](#_Toc130325923)

[10. dropIndex() 9](#_Toc130325924)

[11. Nested Fields 10](#_Toc130325925)

[12. Array Field 11](#_Toc130325926)

[13. Unique 12](#_Toc130325927)

[14. Sparse 13](#_Toc130325928)

[15. Compound 15](#_Toc130325929)

[16. Sort Direction 17](#_Toc130325930)

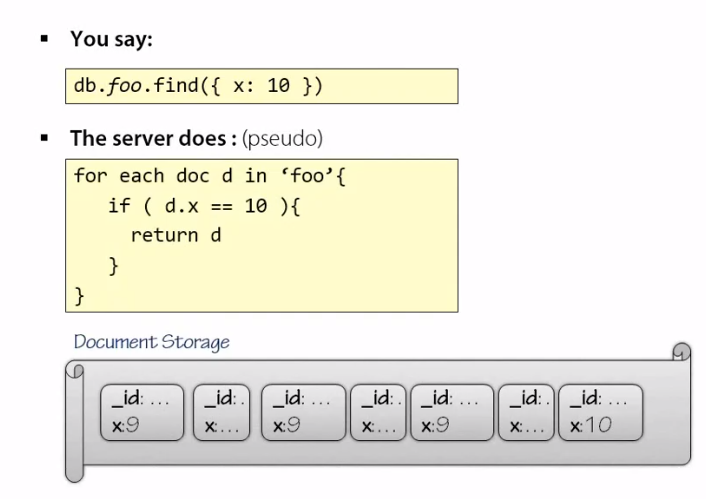
[17. Covering Index https://www.mongodb.com/docs/manual/core/query-optimization/#std-label-read-operations-covered-query 18](#_Toc130325931)

[18. Dead Weight 19](#_Toc130325932)

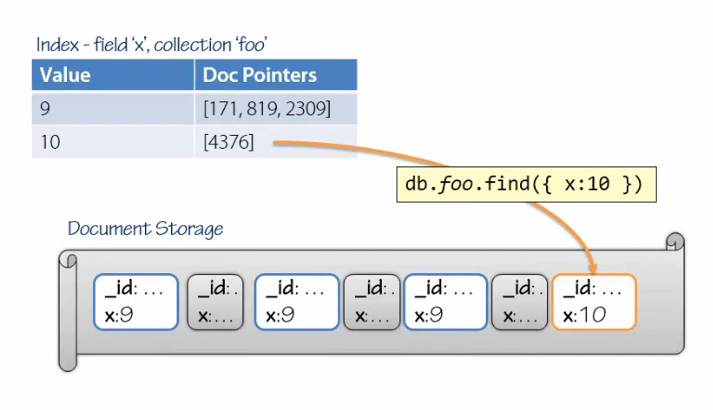
[19. Index Name 20](#_Toc130325933)

[20. References 22](#_Toc130325934)

# The Problem



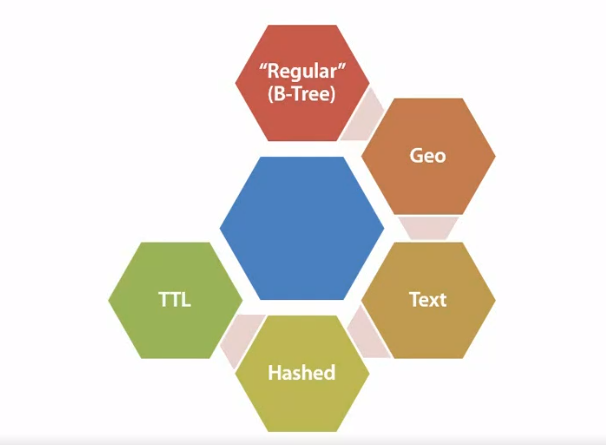
# The Solution



# Sort Uses Index

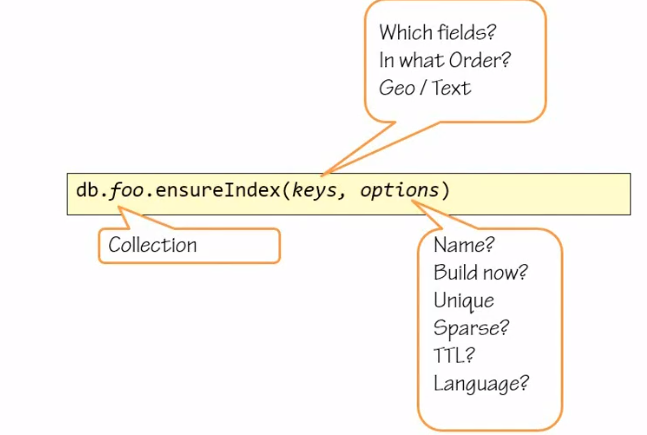
In addition, if the index keys are in a certain sort direction, documents can be sorted with the assistance of indexes. This again is a huge advantage. Documents might not be stored in the order you specify to sort on, but if you are going to sort on a certain key that is in the index, the Mongo query engine can take advantage of that sometimes and use the index for faster sorting because the index itself stores those keys in sorted order.

# Indexes in Mongo



* The regular index is an index you can use on a single field or multiple field with multiple values as well.
* There's a geo index, which is optimized for geographical queries. It doesn't have to be geography, this supports proximity of points to a center, letting you do queries like things near something, sort by nearness, sort by proximity to a certain point, which is great to say find me restaurants around this location.
* There's a fairly new text index. This text index allows you to do things like search engines do, parsing text queries and comparing them against text fields. This is great because now you can use Mongo instead of having another installation of a search engine to index all your documents as a separate operation.
* There's also a hashed index, which is mainly used in context of [sharding](https://www.mongodb.com/docs/manual/sharding/). It allows you to index on a certain field, but have the key values be more evenly distributed instead of clustered. This supports sharding and allows you to spread documents more evenly across your shards.
* And finally, there's a TTL, a time to live index. This supports expiring documents. Using a TTL index, you can designate a data time field on your document to be an expiration date, and Mongo will automatically remove this document from your collection when it expires. This again reduces your overhead and writing all kind of batch jobs to expire documents and remove them yourself.

# Create Index

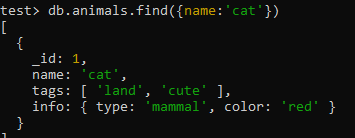


# system.indexes collection

Find a document in the animals collection that has the name of ‘cat’

========================================================

db.animals.find({name:'cat'})



Find out for the collection animals in the database test what indexes exist

===============================================================

db.system.indexes.find({ns:'test.animals'},{key:1}) // Not in use anymore

***db.collection.getIndexes()***

db.animals.getIndexes()



The result is there is one index on the field ID, so there's no index on the name of the animal, only on the ID of those documents. This is how we can tell whether an index exists, but how can we tell whether Mongo uses an index?

# explain()

db.animals.find({name:'cat'}).explain()



Also, try =>

db.animals.find({name:'cat'}).explain("executionStats")

**This will return detailed information about the query execution, including whether an index was used. Look for the executionStats section in the output, which will provide information like the totalKeysExamined and totalDocsExamined. If totalKeysExamined is greater than 0, it means an index was used.**

Add an index to animals collection

==================================

***NOTE***

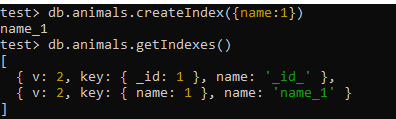
Removed in 5.0

db.collection.ensureIndex() has been replaced by db.collection.createIndex(). For an ascending index on a field, specify a value of 1; for the descending index, specify a value of -1.

db.animals.createIndex({name:1})

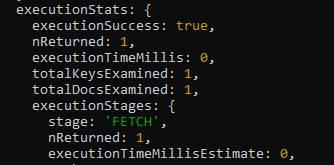
The 1 in the createIndex method signifies the sort order for the index. Specifically, using 1 means the index will sort the documents in ascending order based on the name field. Alternatively, if you use -1, the index will sort the documents in descending order.

db.animals.getIndexes()



v: 2 indicates the version of the index, showing you’re using the format and optimizations introduced in MongoDB 2.0 and later versions.

db.animals.find({name:'cat'}).explain("executionStats")



**explain.totalDocsExamined**

Number of documents examined during the query execution.

**explain.totalKeysExamined**

Number of index keys examined.

**explain.indexesUsed**

Array of strings with the names of the indexes used by the query.

**explain.nReturned**

Number of documents that match the query condition.

**explain.executionTimeMillisEstimate**

Estimated time in milliseconds for the query execution.

# Multi-Term Query

Find a document with a name cat and with a tag land

================================================

\*\* Still uses the index

# Comparison

Index can be used for Range Queries

=========================================

db.animals.find({name:{$lt:'dog'}},{name:1})



db.animals.find({name:{$lt:'dog'}},{name:1}).explain("executionStats")



\*\* Read this as find me a document in the bounds of the index between the empty string and dog, so something that sorts first is the empty string and up to dog.

Find animals whose name is less than donkey

===========================================================

db.animals.find({name:{$lt:'donkey'}},{name:1})



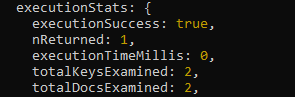
Find an animal whose name is less than donkey and lives on land

===========================================================



Check explain()

=========================



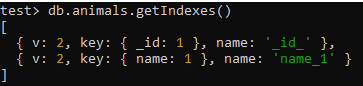
\*\* It had to scan two objects. There were two objects using the index that matched the criteria, but eventually only one document got returned because Mongo obviously applies all your terms. So it first digs up whatever it can with the index and then refines it with a balance of the query terms.

# dropIndex()

Need the index name in order to drop it

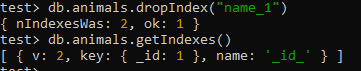
=========================================

db.animals.getIndexes()



\*\* Our collection animals has an index named ***name\_1***.

db.animals.dropIndex("name\_1")



Drop index that does not exist

=================================

db.animals.dropIndex("name\_1")



Index on Id field cannot be dropped

=================================

db.animals.getIndexes()



db.animals.dropIndex("\_id\_")



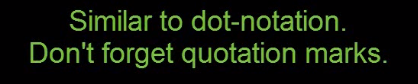
# Nested Fields

\*\* Mongo supports indexing arbitrarily nested fields. This is not true for every NoSQL database out there, and certainly not true for most relational databases where if you store a blob inside a column, a relational database typically does not let you create an index on a part of that blob.

Create index on colour of the animal

====================================

db.animals.createIndex({"info.color":1})





Query for a grey animal

===========================

db.animals.find({"info.color":"grey"},{name:1});



Check if the index was used

==================================================

db.animals.find({"info.color":"grey"},{name:1}).explain("executionStats");



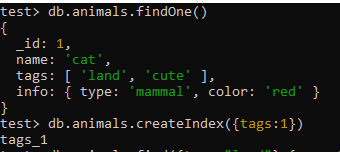
# Array Field

Index on an Array

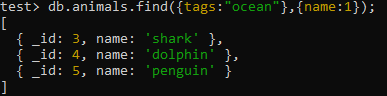
=======================

db.animals.findOne()

db.animals.createIndex({tags:1})



db.animals.find({tags:"ocean"},{name:1});



db.animals.find({tags:"ocean"},{name:1}).explain("executionStats");

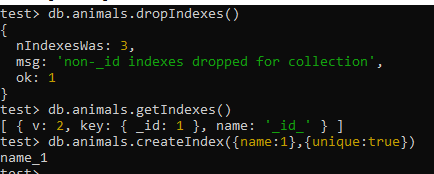


# Unique

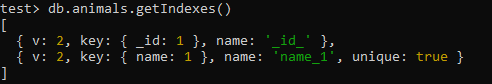
db.animals.dropIndexes()

db.animals.getIndexes()

db.animals.createIndex({name:1},{unique:true})



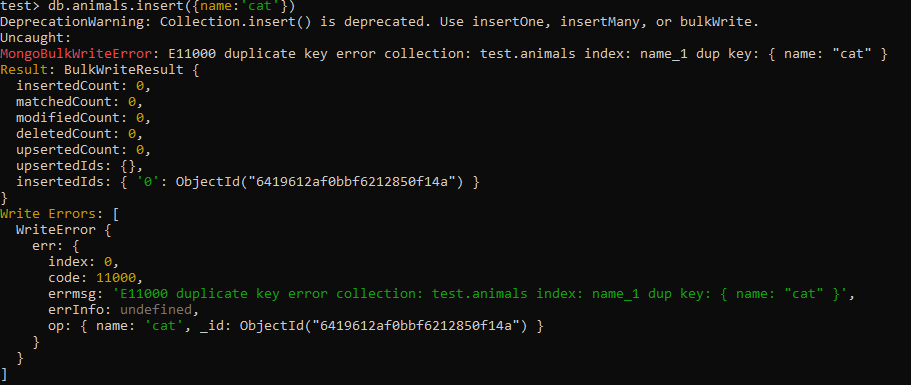
db.animals.getIndexes()



Try to insert an animal that already exists

========================================

db.animals.insert({name:'cat'})



# Sparse

<https://www.mongodb.com/docs/v3.0/core/index-sparse/>

Check number of animals in database

======================================

db.animals.count()



Check how many animals have a color field specified

==============================================

db.animals.count({"info.color" : {$exists:true}})  
  


Create a Sparse index

==============================

\*\* For very large document repositories, the index will attempt to create an entry for every document, and it will save null if the key doesn't exist, if the field doesn't exist in the document. A sparse index is an index that only stores entries for documents that have that field.

db.animals.createIndex({"info.color":1},{sparse:true})



Query on that color

==============================

db.animals.find({"info.color":"red"},{name:1});

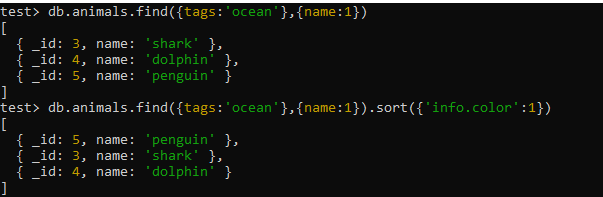


Check ocean animals then sort them

============================

db.animals.find({tags:'ocean'},{name:1})

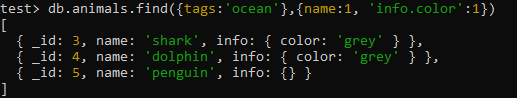
db.animals.find({tags:'ocean'},{name:1}).sort({'info.color':1})



Is the Sparse index working properly?

=========================================

db.animals.find({tags:'ocean'},{name:1, 'info.color':1})



The penguin doesn't have the color field, but it was still included in the index,

\*\* Even though the sort is by the indexed field, MongoDB will **not** select the sparse index to fulfill the query in order to return complete results:

To use the sparse index, explicitly specify the index with hint()

===============================================================

db.animals.find({tags:'ocean'},{name:1}).sort({'info.color':1}).hint( {'info.color':1 } )



# Compound

Create an index on the tags field, which is an array, and the name field, which is a string

===========================================================================

\*\* Run a query with two terms and it uses the index.

db.animals.createIndex({tags:1,name:1})



[Mongo refuses to utilize all fields from compound index](https://stackoverflow.com/questions/49185669/mongo-refuses-to-utilize-all-fields-from-compound-index) or uses another more generic index

==========================================================================

db.animals.find({name:'shark', tags:'ocean'}).explain()



Ordering makes no difference

=============================

db.animals.find({ tags:'ocean', name:'shark'}).explain()

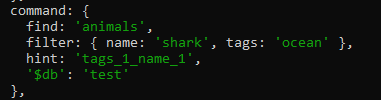


Solution is to use hint()

==========================

db.animals.find({name:'shark', tags:'ocean'}).hint('tags\_1\_name\_1').explain()





Query on only one term in the index

=====================================

db.animals.find({name:'shark'}).explain()





\*\* In this case, MongoDB uses a previously defined index. The index on the tags, then the name. The index is deemed useful if the terms in the index match the index definition from left to right.

# Sort Direction

MongoDB uses the index to sort in the direction supported by the index

=================================================================

db.animals.find().sort({tags:1,name:1}).explain()





Also in the exact opposite direction

===============================

db.animals.find().sort({tags:-1,name:-1}).explain()

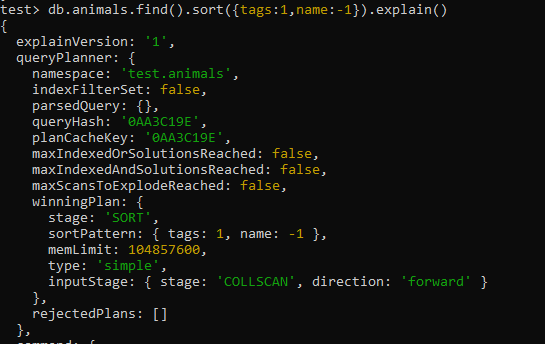




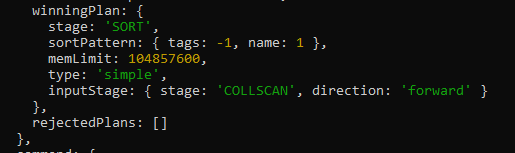
If terms are in the exact opposite sort direction, MongoDB will not use an index

=====================================================================

db.animals.find().sort({tags:1,name:-1}).explain()



db.animals.find().sort({tags:-1,name:1}).explain()



# Covering Index [https://www.mongodb.com/docs/manual/core/query-optimization/#std-label-read-operations-covered-query](https://www.mongodb.com/docs/manual/core/query-optimization/%23std-label-read-operations-covered-query)

A covered query is a query that can be satisfied entirely using an index and does not have to examine any documents. An index [covers](https://www.mongodb.com/docs/manual/core/query-optimization/#std-label-indexes-covered-queries) a query when all of the following apply:

* all the fields in the [query](https://www.mongodb.com/docs/manual/tutorial/query-documents/#std-label-read-operations-query-document) are part of an index, **and**
* all the fields returned in the results are in the same index.
* no fields in the query are equal to null (i.e. {"field" : null} or {"field" : {$eq : null}} ).

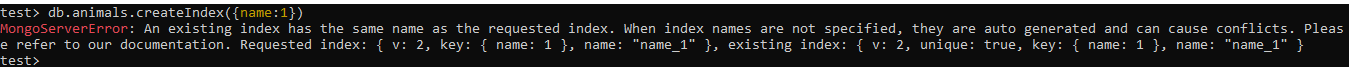
For example, a collection inventory has the following index on the type and item fields:

|  |
| --- |
| db.inventory.createIndex( { type: 1, item: 1 } ) |

Ensure there is an index on the name field

========================================

db.animals.createIndex({name:1})



Find a document with the name of cat

========================================

db.animals.find({name:'cat'},{name:1})



Does Mongo use the index only to return the results of the query?

========================================================

\*\* For the specified index to cover the query, the projection document must explicitly specify \_id: 0 to exclude the \_id field from the result since the index does not include the \_id field. For the cases where you need just a few fields and your fields are already covered by the index that's going to be used by the query, Mongo can make efficient use of the index and not visit the documents at all.

Exclude id field to ensure MongoDB uses the index

=============================================

db.animals.find({name:'cat'},{\_id:0,name:1})



# Dead Weight

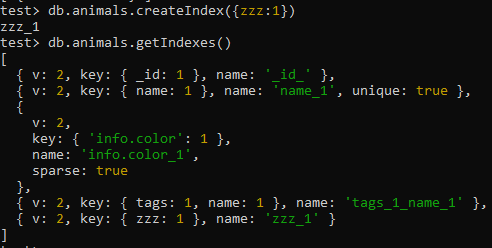
\*\* MongoDB is schema-less, so Mongo doesn't know what fields you might have in the future in the document. It also doesn't keep a central account of all fields present in all documents. So it's quite possible, and allowed, to create an index on a field name that doesn't match any field in any of your documents.

Create an index on a field that does not exist

=========================================

db.animals.createIndex({zzz:1})

db.animals.getIndexes()



\*\* None of the animals have a field named zzz, but Mongo will happily ensure there's an index on that field should one appear one day, and it keeps track of it. If you misspell a field name, Mongo is not going to prevent you from creating it, it's not going to notify you that your documents actually have the correctly spelled name and your index does not. So it can create an extra index that will never be hit and you wouldn't know it. The best way to ensure that Mongo is using indexes to satisfy your queries is to use explain, inspect the results, and make sure your queries are well optimized.

# Index Name

Mongo lets you create compound indexes with up to 31 fields in it. The name of the index is a concatenation of the field names and their sorting direction.

Create index on animals using many fields

=====================================

var keys = {

"\_id":1,

"name":1,

"tags":1,

"info.color":1,

"info.canFly":1,

"anotherVeryLongFieldName":1

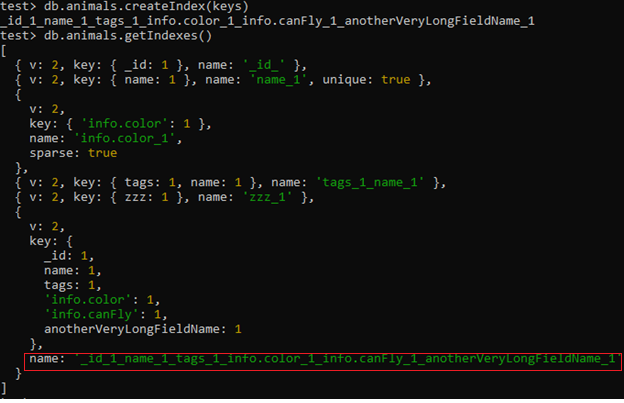
};

keys

db.animals.createIndex(keys)

db.animals.getIndexes()



  
So if my compound index has long names and a lot of them, and then I created an index using those keys, then the generated name will be quite long. It might hit against the 128 character limit that Mongo imposes on indexes name. That 128 includes the name of the collection. So, not a real problem with small indexes, not a real problem when you have just a few fields and their names aren't long. A big problem if you have many fields and their names are long. This can occur not because you choose huge names, but also if you index very deep properties nested inside fields and subdocuments. To resolve that, you can choose what name to give your index when you create the index.

Drop the previous index

==================================

db.animals.dropIndex("\_id\_1\_name\_1\_tags\_1\_info.color\_1\_info.canFly\_1\_anotherVeryLongFieldName\_1")



Create an index with a same definition and specify a name

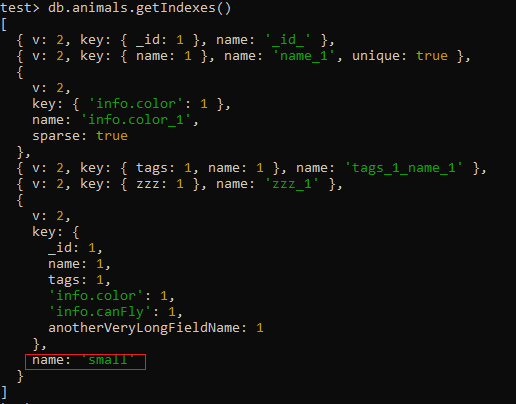
db.animals.createIndex(keys,{name:'small'})



Check if the index has been created with the specified name

====================================================

db.animals.getIndexes()



|  |
| --- |
| Geospatial Indexes: |
| 2dsphere |
| 2d |

db.places.insertMany([

{

name: "Cubbon Park",

location: { type: "Point", coordinates: [77.592, 12.976] },

description: "A large green space in the heart of Bengaluru with lush gardens."

},

{

name: "Lalbagh Botanical Garden",

location: { type: "Point", coordinates: [77.584, 12.950] },

description: "Historic garden with diverse plant species and a famous glass house."

},

{

name: "Bangalore Palace",

location: { type: "Point", coordinates: [77.591, 12.998] },

description: "Grand palace with Tudor-style architecture and sprawling grounds."

},

{

name: "MG Road",

location: { type: "Point", coordinates: [77.609, 12.975] },

description: "Bustling street known for shopping, dining, and entertainment."

}

])

Geospatial indexes in MongoDB allow you to efficiently perform queries that involve geospatial data, such as finding documents within a certain distance from a point or within a specific area. There are two primary types of geospatial indexes:

**1. 2dsphere Index**

* **Use Case:** This is used for storing and querying GeoJSON data and legacy coordinate pairs. It supports complex geometries and can handle spherical geometry calculations, making it suitable for earth-like spherical representations.
* **Creation:**

db.places.createIndex({ location: "2dsphere" })

db.places.find({

location: {

$near: {

$geometry: {

type: "Point",

coordinates: [77.5946, 12.9716] // Example coordinates (Bengaluru)

},

$maxDistance: 5000 // Distance in meters

}

}

})

**2d Index**

* **Use Case:** This index type is used for querying legacy coordinate pairs (flat, non-spherical geometries). It’s best for simpler use cases and does not support GeoJSON objects or spherical calculations.
* **Creation:**

db.restaurants.insertMany([

{

name: "Pizza Place",

location: [77.610, 12.930], // Correct format for 2d index

cuisine: "Italian"

},

{

name: "Sushi Spot",

location: [77.615, 12.940], // Correct format for 2d index

cuisine: "Japanese"

},

{

name: "Taco Town",

location: [77.620, 12.950], // Correct format for 2d index

cuisine: "Mexican"

},

{

name: "Burger Barn",

location: [77.625, 12.960], // Correct format for 2d index

cuisine: "American"

}

])

db.restaurants.createIndex({ location: "2d" })

db.restaurants.find({

location: {

$near: [77.610, 12.930], // Example coordinates

$maxDistance: 0.01 // Distance in degrees

}

})

This query will return restaurants near the provided coordinates within the specified distance. The 2d index will help optimize the performance of this query.

# References

<https://www.mongodb.com/docs/manual/indexes/>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*