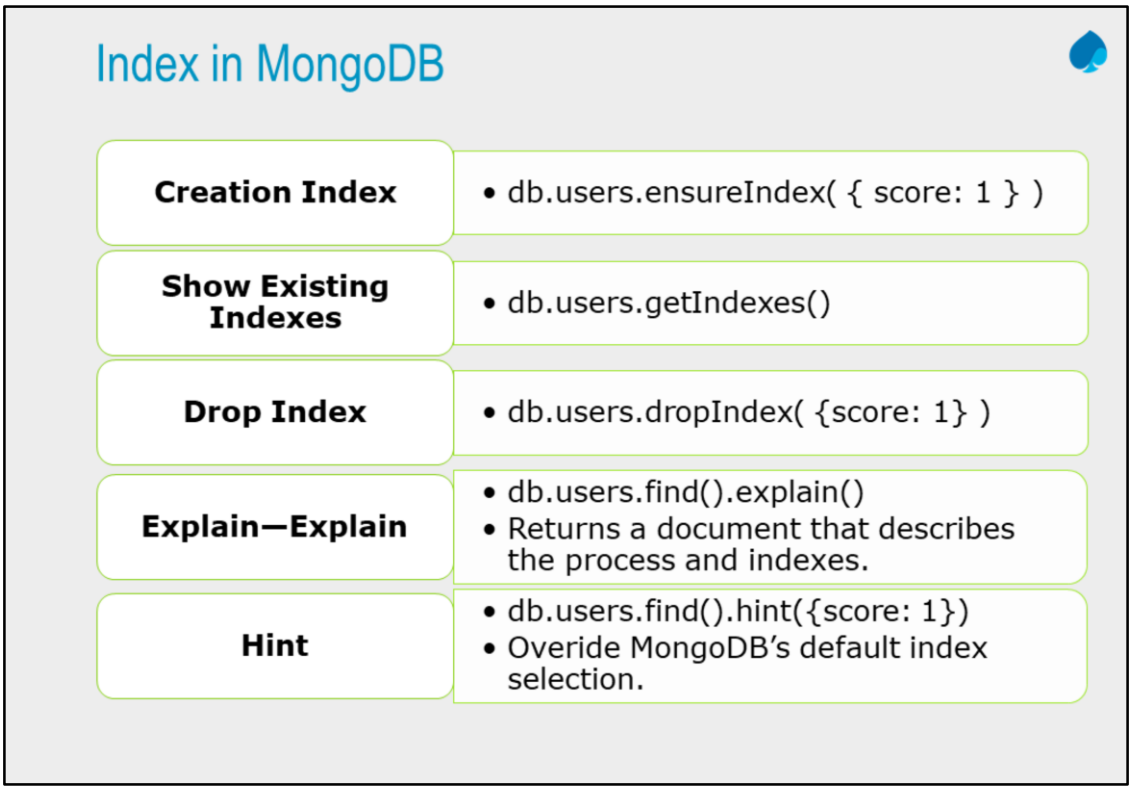
MongoDB – Indexing
Lesson-05

Capgemini



The db.collection.createIndex method only creates an index if an index of the same specification does not already exist.

Above method is Deprecated since version 3.0.0: db.collection.ensureIndex() is now an alias for db.collection.createIndex().

The db.collection.explain() method is used to returns information on the query execution of various methods like aggregate(); count(); find(); group(); remove(); and update() .

Parameter - verbosity
Description- Specifies the verbosity mode for the explain output. The mode affects the behavior of explain() and determines the amount of information to return.
The possible modes are: "queryPlanner", "executionStats", and "allPlansExecution".

Understand about Indexes- Before Index

What does database normally do when we query?

- **MongoDB must scan every document.**
- **Inefficient because process large volume of data.**

```
db.users.find( { score: { "$lt" : 30} } )
```

Understand about Indexes- What is Index?

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.

The index stores the value of a specific field or set of fields, ordered by the value of the field.

Understand about Indexes- Definition of Index

Definition

- **Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.**

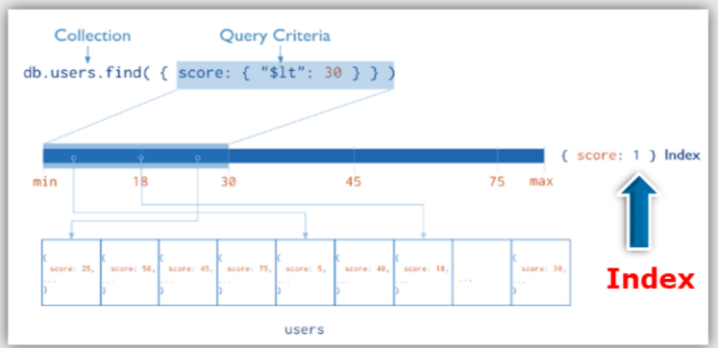


Diagram of a query that uses an index to select.

Understand about Indexes-Indexes

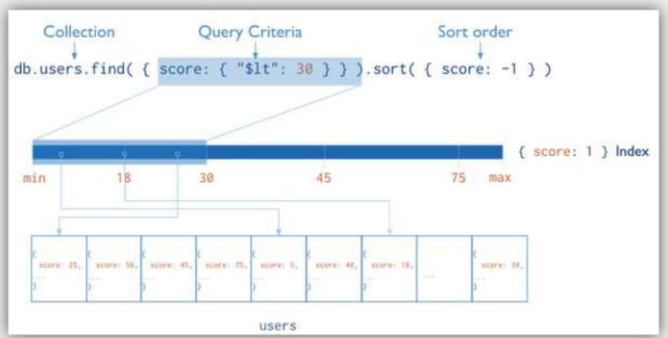
MongoDB can use indexes to return documents sorted by the index key directly from the index without requiring an additional sort phase.

Indexes help efficient execution of queries.

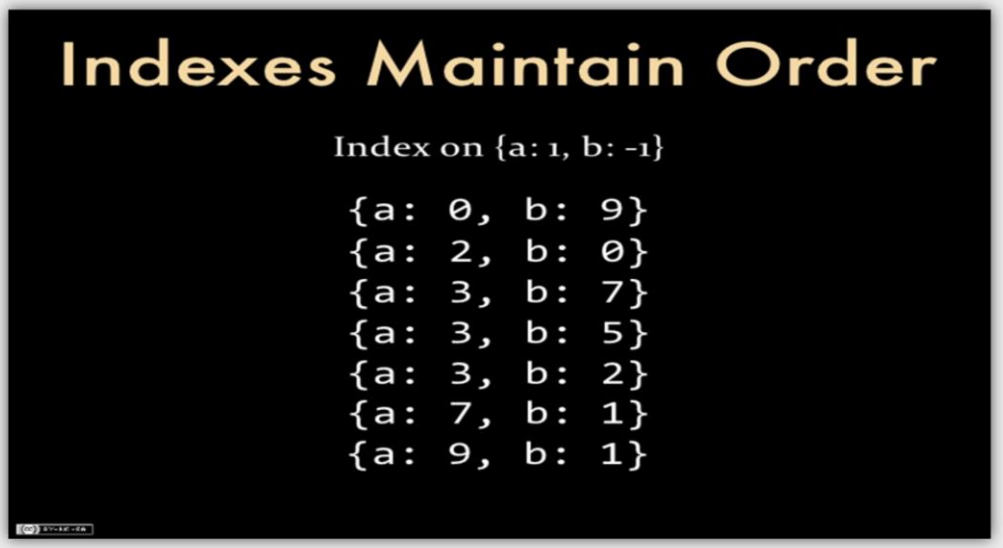
- Without indexes MongoDB must scan every document in a collection to select those documents that match the query statement.

These collection scans are inefficient because they require Mongod to process a larger volume of data than an index for each operation.

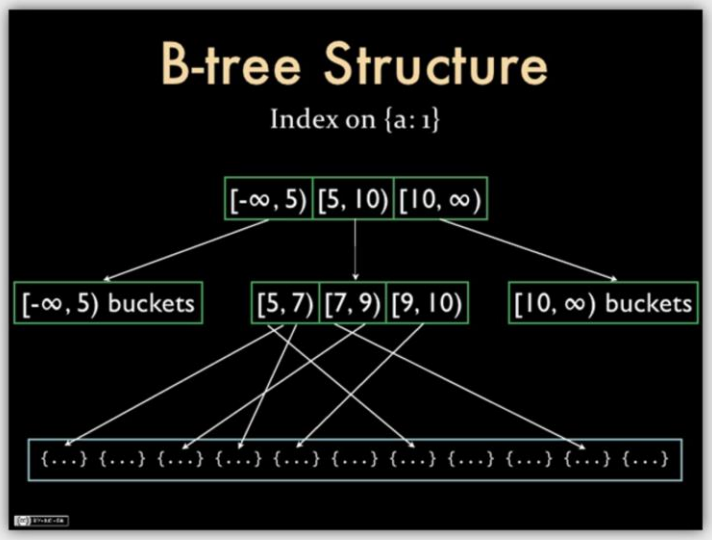
Understand about Indexes--Diagrammatic Representation



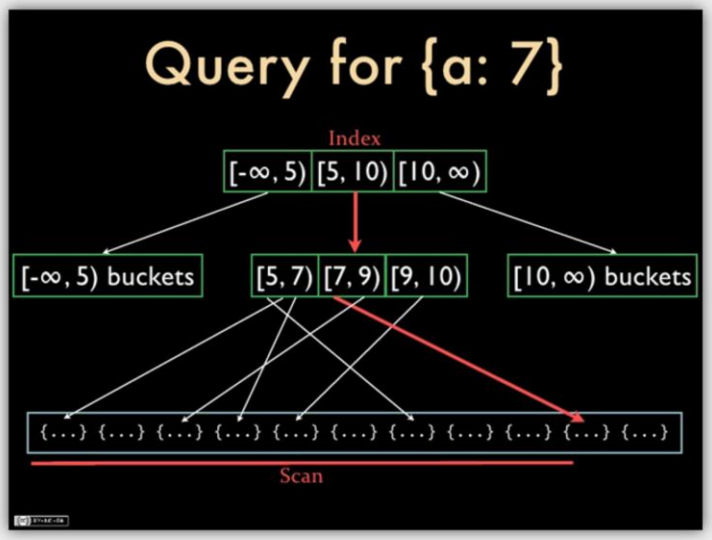
Understand about Indexes-Indexes



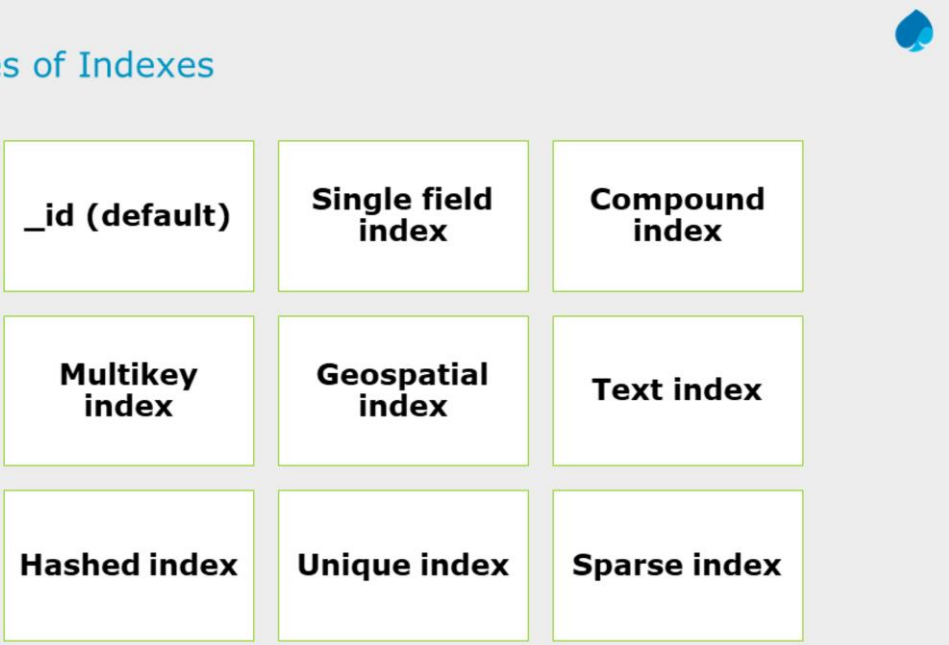
Understand about Indexes-B-tree Index Structure



Understand about Indexes-Index Querying



Types of Indexes



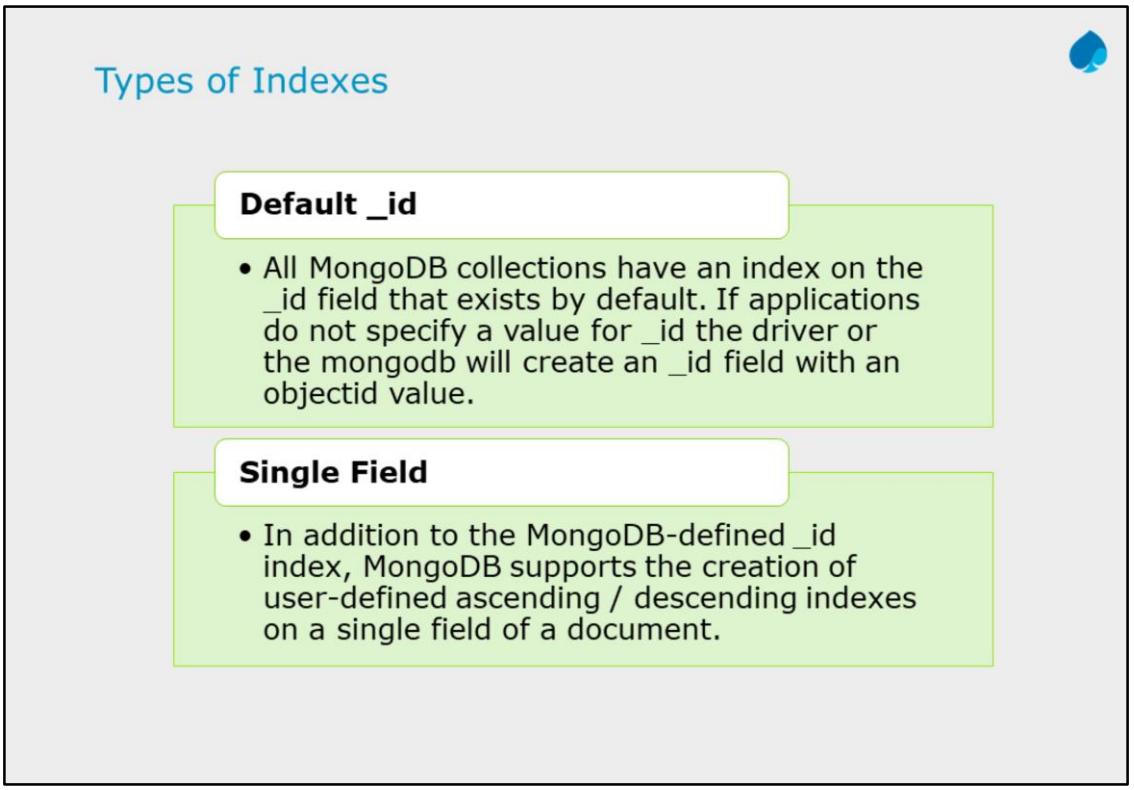
Types of Indexes-About Indexes

Without indexes, MongoDB must perform a *collection scan*, i.e. scan every document in a collection, to select those documents that match the query statement.

The ordering of the index entries supports efficient equality matches and range-based query operations.

The index stores the value of a specific field or set of fields, ordered by the value of the field.

MongoDB can return sorted results by using the ordering in the index.



Create Index On Firstname

```
> db.employees.createIndex({firstname:-1})
```

Types of Indexes-Example

{ "_id" : ObjectId(...), "name" : "Alice", "age" : 27 }

The following command creates an index on the name field:

- db.friends.createIndex({ "name" : 1 })

- **Single Field Indexes**
  - db.users.ensureIndex({ score: 1 })

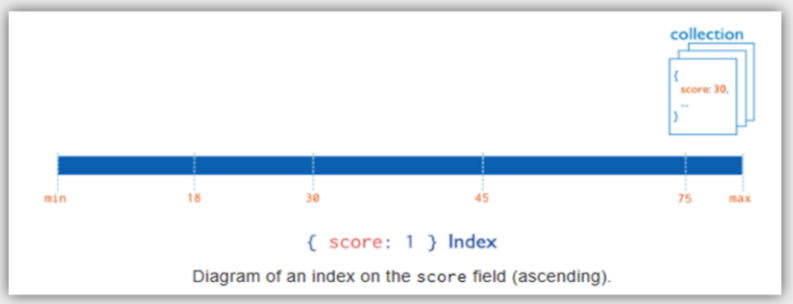


Diagram of an index on the score field (ascending).

The db.collection.createIndex method only creates an index if an index of the same specification does not already exist.

Above method is Deprecated since version 3.0.0: db.collection.ensureIndex() is now an alias for db.collection.createIndex().

Types of Indexes-Compound Indexes

MongoDB also supports user-defined indexes on multiple fields, i.e. Compound indexes.

The order of fields listed in a compound index has significance. For instance, if a compound index consists of { userid: 1, score: -1 }, the index sorts first by userid and then, within each user id value, sorts by score.

Types of Indexes-Compound Indexes (contd.)

Compound Field Indexes

- db.users.ensureIndex({ userid:1, score: -1 })

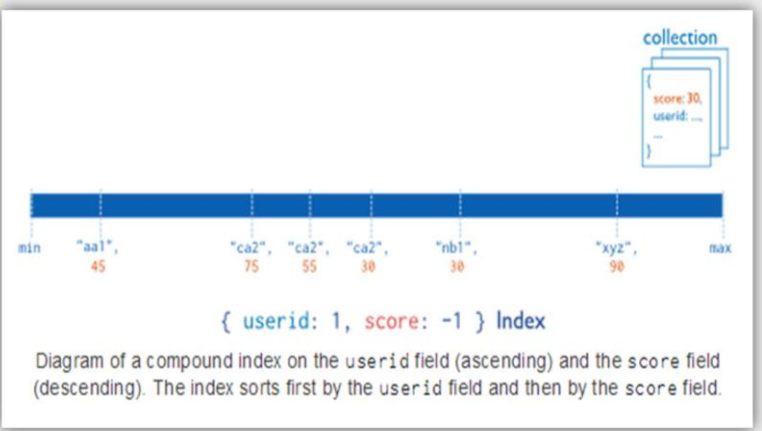


Diagram of a compound index on the `userid` field (ascending) and the `score` field (descending). The index sorts first by the `userid` field and then by the `score` field.

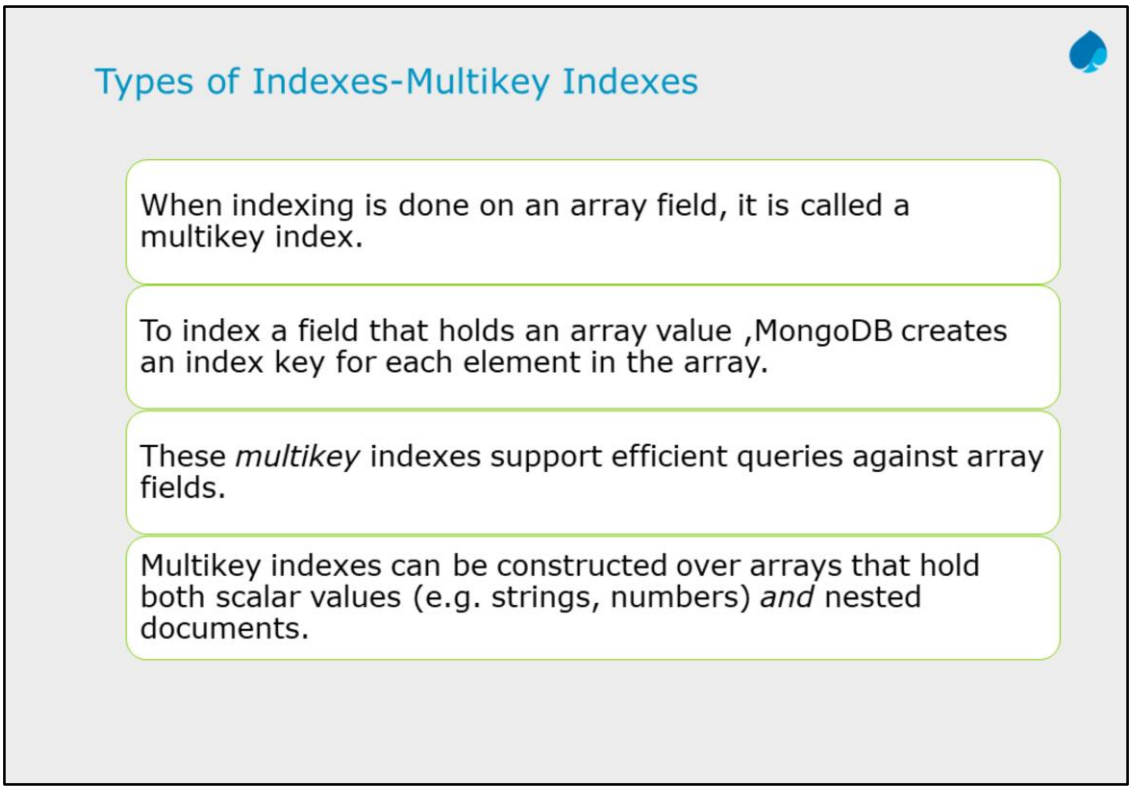
Types of Indexes-Example

Consider a collection named products that holds documents that resemble the following document:

- { "_id": ObjectId(...), "item": "Banana", "category": ["food", "produce", "grocery"], "location": "4th Street Store", "stock": 4, "type": "cases", "arrival": Date(...) }

If applications query on the item field as well as query on both the item field and the stock field, you can specify a single compound index to support both of these queries:

- db.products.createIndex({ "item": 1, "stock": 1 })

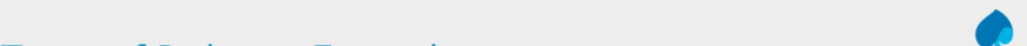


Multi Key Index-

db.employees.createIndex({"annlGrossSal.year": 1 })

Note :annual Gross Salary in employee is array of objects. so the above index created is called as multi key index.

Types of Indexes-Example



Lets consider a document:

```
{
"title": "Superman",
"tags": ["comic", "action", "xray"],
"issues": [ { "number": 1, "published_on": "June 1938"
 }
 ]
}
```

Multikey indexes lets us search on the values in the tags array as well as in the issues array. Let's create two indexes to cover both.

Types of Indexes-Example (contd.)

```
db.comics.ensureIndex({tags: 1});
Db.comics.ensureIndex({issues: 1});
```

If the document changes structure it would be better to create a specific compound index on the fields needed in sub element document.

```
Db.comics.ensureIndex({"issues.number":1,
"issues.published_on":1});
```

Types of Indexes-Limitations

Consider a collection that contains the following document:

- { _id: 1, a: [1, 2], b: [1, 2], category: "AB - both arrays" }

You cannot create a compound multikey index { a: 1, b: 1 } on the collection since both the a and b fields are arrays.

Types of Indexes-Demo of Indexes in MongoDB

Import Data

Create Index

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

Show Existing Index

Hint

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

Explain

Compare with data without indexes

```
> db.zips.find().limit(20)
{ "city" : "ACMAR", "loc" : [ -86.51557, 33.584132 ], "pop" : 6055, "state" : "AL", "_id" : "35004" }
{ "city" : "ADAMSVILLE", "loc" : [ -86.959727, 33.588437 ], "pop" : 10616, "state" : "AL", "_id" : "35005" }
{ "city" : "ADGER", "loc" : [ -87.167455, 33.434277 ], "pop" : 3205, "state" : "AL", "_id" : "35006" }
{ "city" : "KEYSTONE", "loc" : [ -86.812861, 33.236868 ], "pop" : 14218, "state" : "AL", "_id" : "35007" }
{ "city" : "NEW SITE", "loc" : [ -85.951086, 32.941445 ], "pop" : 19942, "state" : "AL", "_id" : "35010" }
{ "city" : "ALPINE", "loc" : [ -86.208934, 33.331165 ], "pop" : 3062, "state" : "AL", "_id" : "35014" }
{ "city" : "ARAB", "loc" : [ -86.489638, 34.328339 ], "pop" : 13650, "state" : "AL", "_id" : "35016" }
{ "city" : "BAILEYTON", "loc" : [ -86.621299, 34.268298 ], "pop" : 1781, "state" : "AL", "_id" : "35019" }
{ "city" : "BESSEMER", "loc" : [ -86.947547, 33.409002 ], "pop" : 40549, "state" : "AL", "_id" : "35020" }
{ "city" : "HUEYTOWN", "loc" : [ -86.999607, 33.414625 ], "pop" : 39677, "state" : "AL", "_id" : "35023" }
{ "city" : "BLOUNTSVILLE", "loc" : [ -86.568628, 34.092937 ], "pop" : 9058, "state" : "AL", "_id" : "35031" }
{ "city" : "BREMEN", "loc" : [ -87.004281, 33.973664 ], "pop" : 3448, "state" : "AL", "_id" : "35033" }
{ "city" : "BRENT", "loc" : [ -87.211387, 32.93567 ], "pop" : 3791, "state" : "AL", "_id" : "35034" }
{ "city" : "BRIERFIELD", "loc" : [ -86.951672, 33.042747 ], "pop" : 1282, "state" : "AL", "_id" : "35035" }
{ "city" : "CALERA", "loc" : [ -86.755987, 33.1098 ], "pop" : 4675, "state" : "AL", "_id" : "35040" }
{ "city" : "CENTREVILLE", "loc" : [ -87.11924, 32.958324 ], "pop" : 4902, "state" : "AL", "_id" : "35042" }
{ "city" : "CHELSEA", "loc" : [ -86.614132, 33.371582 ], "pop" : 4781, "state" : "AL", "_id" : "35043" }
{ "city" : "COOSA PINES", "loc" : [ -86.337622, 33.266928 ], "pop" : 7985, "state" : "AL", "_id" : "35044" }
{ "city" : "CLANTON", "loc" : [ -86.642472, 32.835532 ], "pop" : 13990, "state" : "AL", "_id" : "35045" }
{ "city" : "CLEVELAND", "loc" : [ -86.559355, 33.992186 ], "pop" : 2369, "state" : "AL", "_id" : "35049" }
> db.zips.find().count()
29467
```

Types of Indexes-Demo of Indexes in MongoDB (contd.)

Import Data

Create Index

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

Show Existing Index

Hint

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

Explain

Compare with data without indexes

```
db.zips.ensureIndex({pop: -1})
db.zips.ensureIndex({state: 1, city: 1})
db.zips.ensureIndex({loc: -1})
```

Types of Indexes-Demo of Indexes in MongoDB (contd.)

Import Data

Create Index

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

Show Existing Index

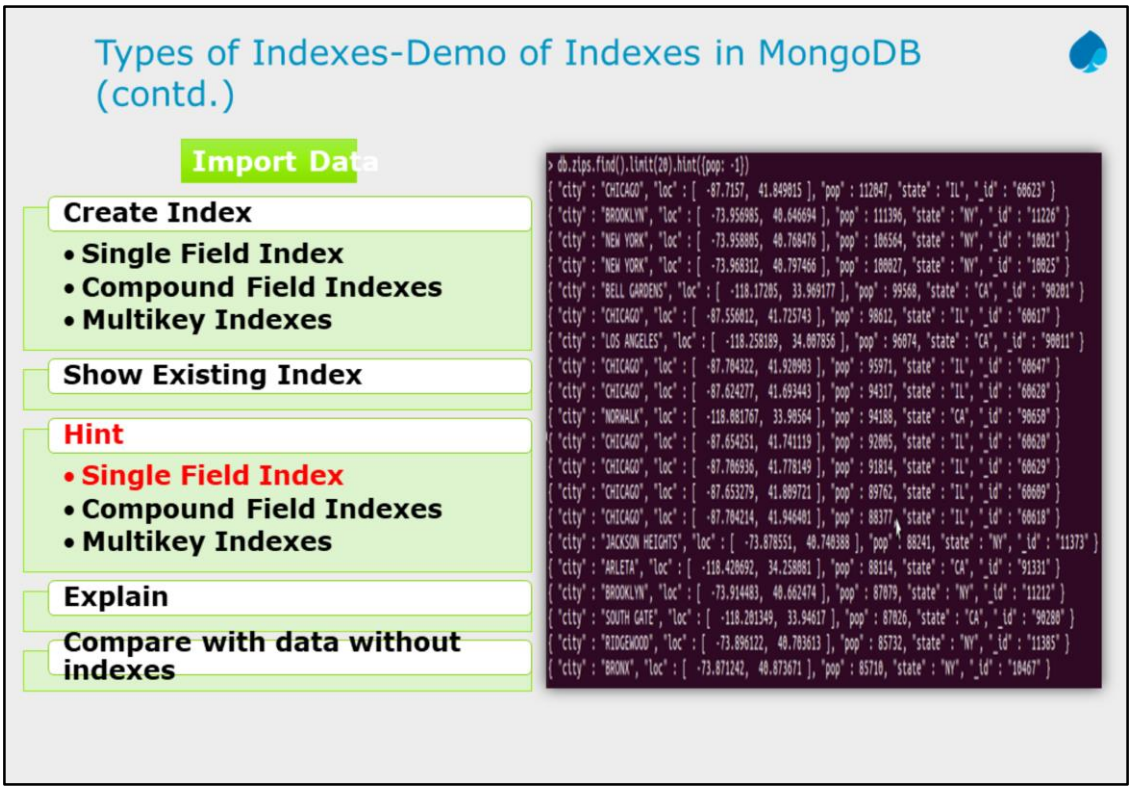
Hint

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

Explain

Compare with data without indexes

```
> db.zips.getIndexes()
[
        {
                "v" : 1,
                "key" : {
                        "_id" : 1
                },
                "ns" : "blog.zips",
                "name" : "_id_"
        },
        {
                "v" : 1,
                "key" : {
                        "pop" : 1
                },
                "ns" : "blog.zips",
                "name" : "pop_1"
        },
        {
                "v" : 1,
                "key" : {
                        "state" : 1,
                        "city" : 1
                },
                "ns" : "blog.zips",
                "name" : "state_1_city_1"
        },
        {
                "v" : 1,
                "key" : {
                        "loc" : 1
                },
                "ns" : "blog.zips",
                "name" : "loc_1"
        }
]
```



Hint()-
Call this method on a query to override MongoDB's default index selection and query optimization process. U
se db.collection.getIndexes() to return the list of current indexes on a collection.
The cursor.hint() method has the following parameter:

- Parameter – index
- Type - string or document
- Description - The index to "hint" or force MongoDB to use when performing the query. Specify the index either by the index name or by the index specification document.

  You can also specify { $natural : 1 } to force the query to perform a forwards collection scan, or { $natural : -1 } for a reverse collection scan.

  Ex

  >db.employees.find().pretty()
  >db.employees.ensureIndex({firstname: 1 })
  >db.employees.find().hint({ firstname: 1 }).pretty()

  Note-after the hint function used on find default _id index is ignored and entry are found using firstname.

Types of Indexes-Demo of Indexes in MongoDB (contd.)

Import Data

Create Index

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

Show Existing Index

Hint

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

Explain

Compare with data without indexes

```
> db.zips.find().limit(20).hint({state: 1, city: 1})
{ "city" : "98791", "loc" : [ -176.310048, 51.938901 ], "pop" : 5345, "state" : "AK", "_id" : "98791" }
{ "city" : "AKHIOK", "loc" : [ -152.580169, 57.781967 ], "pop" : 13309, "state" : "AK", "_id" : "99615" }
{ "city" : "AKIACHAK", "loc" : [ -161.39233, 60.891854 ], "pop" : 481, "state" : "AK", "_id" : "99551" }
{ "city" : "AKIAK", "loc" : [ -161.199325, 60.890632 ], "pop" : 285, "state" : "AK", "_id" : "99552" }
{ "city" : "AKUTAN", "loc" : [ -165.785368, 54.143012 ], "pop" : 589, "state" : "AK", "_id" : "99553" }
{ "city" : "ALAKANUK", "loc" : [ -164.68228, 62.746967 ], "pop" : 1186, "state" : "AK", "_id" : "99554" }
{ "city" : "ALEKNAGIK", "loc" : [ -158.619882, 59.269688 ], "pop" : 185, "state" : "AK", "_id" : "99555" }
{ "city" : "ALLAKAKET", "loc" : [ -152.712155, 66.543197 ], "pop" : 170, "state" : "AK", "_id" : "99720" }
{ "city" : "AMBLER", "loc" : [ -156.455652, 67.46951 ], "pop" : 8, "state" : "AK", "_id" : "99786" }
{ "city" : "ANAKTUVUK PASS", "loc" : [ -151.679085, 68.11878 ], "pop" : 260, "state" : "AK", "_id" : "99721" }
{ "city" : "ANCHORAGE", "loc" : [ -149.876077, 61.211571 ], "pop" : 14436, "state" : "AK", "_id" : "99501" }
{ "city" : "ANCHORAGE", "loc" : [ -150.093943, 61.096163 ], "pop" : 15891, "state" : "AK", "_id" : "99502" }
{ "city" : "ANCHORAGE", "loc" : [ -149.893844, 61.189953 ], "pop" : 12534, "state" : "AK", "_id" : "99503" }
{ "city" : "ANCHORAGE", "loc" : [ -149.74467, 61.203696 ], "pop" : 32383, "state" : "AK", "_id" : "99504" }
{ "city" : "ANCHORAGE", "loc" : [ -149.828912, 61.153543 ], "pop" : 28128, "state" : "AK", "_id" : "99507" }
{ "city" : "ANCHORAGE", "loc" : [ -149.810085, 61.205959 ], "pop" : 29857, "state" : "AK", "_id" : "99508" }
{ "city" : "ANCHORAGE", "loc" : [ -149.897401, 61.119381 ], "pop" : 17094, "state" : "AK", "_id" : "99515" }
{ "city" : "ANCHORAGE", "loc" : [ -149.779998, 61.10541 ], "pop" : 18356, "state" : "AK", "_id" : "99516" }
{ "city" : "ANCHORAGE", "loc" : [ -149.936111, 61.190136 ], "pop" : 15192, "state" : "AK", "_id" : "99517" }
{ "city" : "ANCHORAGE", "loc" : [ -149.886571, 61.154862 ], "pop" : 8116, "state" : "AK", "_id" : "99518" }
```

Types of Indexes-Demo of Indexes in MongoDB (contd.)

Import Data

Create Index

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

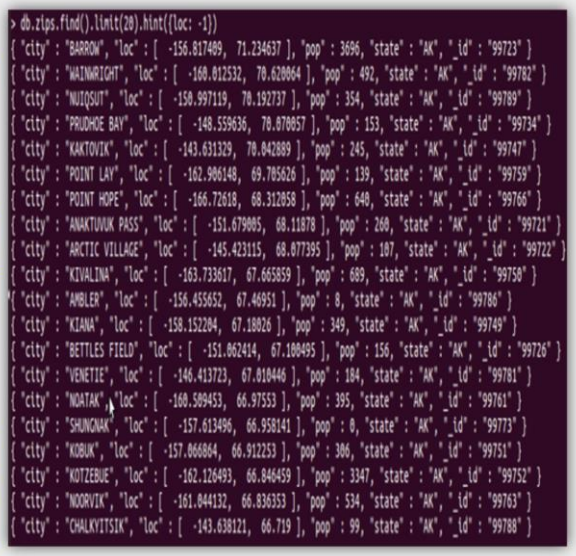
Show Existing Index

Hint

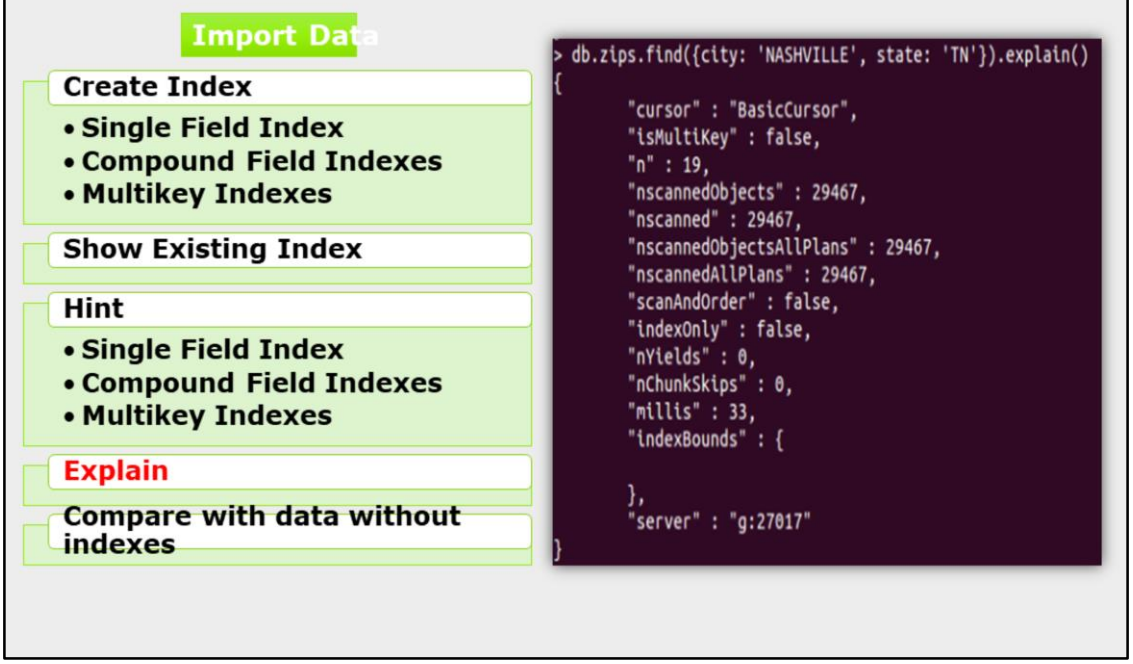
- Single Field Index
- Compound Field Indexes
- Multikey Indexes

Explain

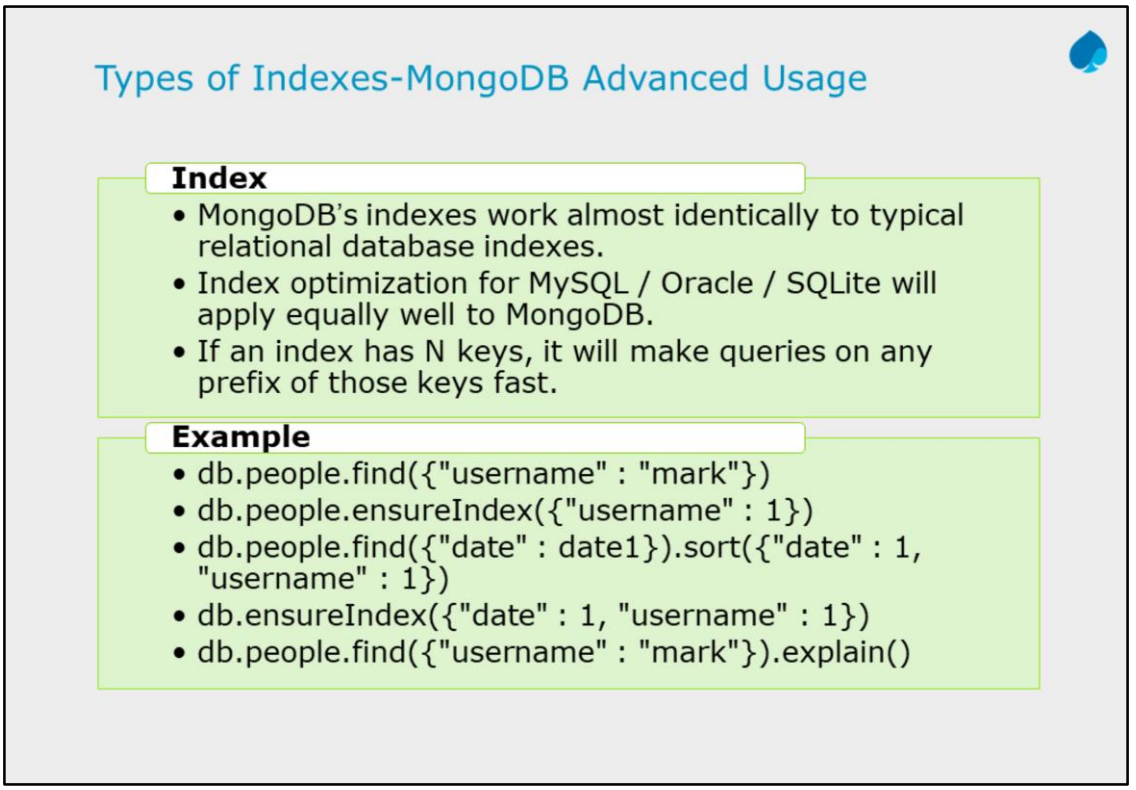
Compare with data without indexes



Types of Indexes-Demo of Indexes in MongoDB (contd.)




```
> db.employees.find({firstname:"Mahima"}).explain()
{
      "queryPlanner" : {
            "plannerVersion" : 1,
            "namespace" : "cgdb.employees",
            "indexFilterSet" : false,
            "parsedQuery" : {
                  "firstname" : {
                        "$eq" : "Mahima"
                  }
            },
            "winningPlan" : {
                  "stage" : "FETCH",
                  "inputStage" : {
                        "stage" : "IXSCAN",
                        "keyPattern" : {
                              "firstname" : 1
                        },
                        "indexName" : "firstname_1",
                        "isMultiKey" : false,
                        "direction" : "forward",
                        "indexBounds" : {
                              "firstname" : [
                                    "[\"Mahima\", \"Mahima\"]"
                              ]
                        }
                  }
            },
            "rejectedPlans" : [ ]
      },
      "serverInfo" : {
            "host" : "PUNL65530",
            "port" : 27017,
            "version" : "3.0.15",
            "gitVersion" : "b8ff507269c382bc100fc52f75f48d54cd42ec3b modul
: enterprise"
```



Do not index every key. This will make inserts slow, take up lots of space, and probably not speed up your queries very much. Figure out what queries you are running, what the best indexes are for these queries, and make sure that the server is using the indexes you've created using the explain and hint tools described in the next section.

Types of Indexes-MongoDB Advanced Usage (contd.)

Indexes can be created on keys in embedded documents in the same way that they are created on normal keys.

Indexing for Sorts: Indexing the sort allows MongoDB to pull the sorted data in order, allowing you to sort any amount of data without running out of memory.

Index Nameing rule:
keyname1_dir1_keyname2_dir2_..._keynameN_dirN, where keynameX is the index's key and dirX is the index's direction (1 or -1).

```
– db.blog.ensureIndex({"comments.date" : 1})
– db.people.ensureIndex({"username" : 1}, {"unique" : true})
– db.people.ensureIndex({"username" : 1}, {"unique" : true})
– autocomplete:PRIMARY> db.system.indexes.find()
        { "name" : "_id_", "ns" : "test.fs.files", "key" : { "_id" : 1 }, "v" : 0 }
        { "ns" : "test.fs.files", "key" : { "filename" : 1 }, "name" :
          "filename_1", "v" : 0 }
        { "name" : "_id_", "ns" : "test.fs.chunks", "key" : { "_id" : 1 }, "v" :
          0 }
        { "ns" : "test.fs.chunks", "key" : { "files_id" : 1, "n" : 1 }, "name" :
          "files_id_1_n_1", "v" : 0 }
```

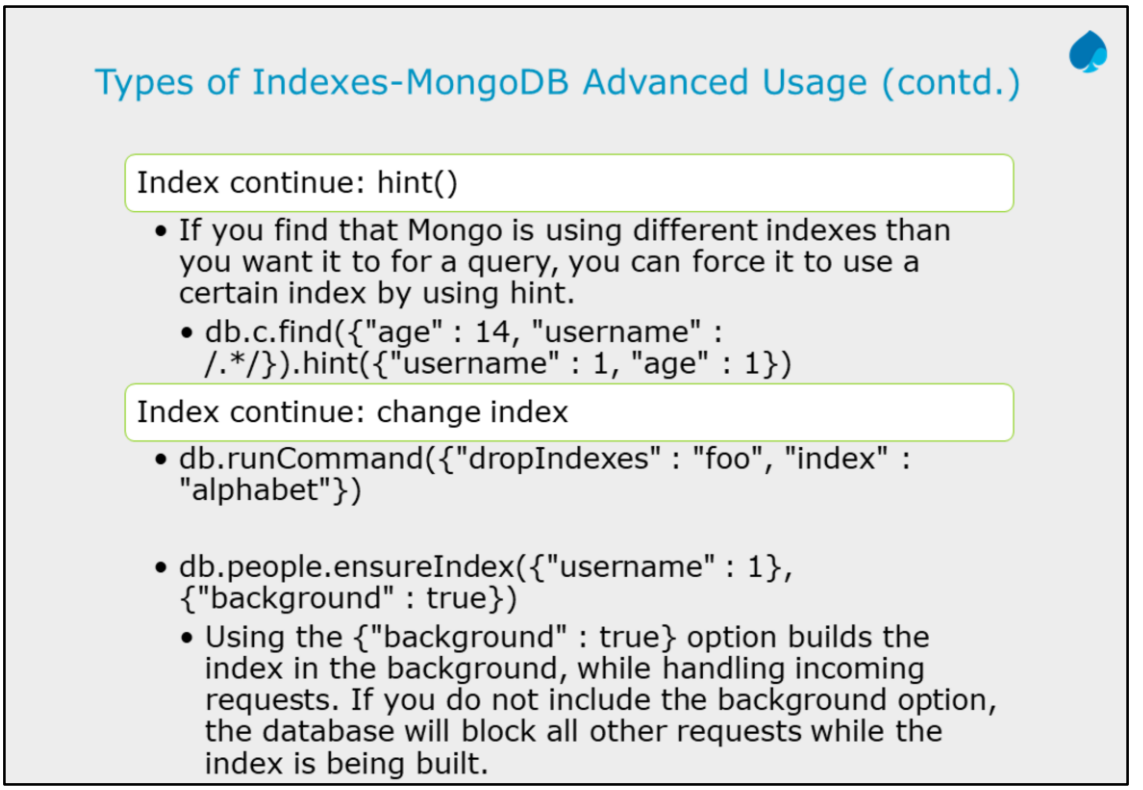
- A unique index ensures that the indexed fields do not store duplicate values; i.e. enforces uniqueness for the indexed fields. By default, MongoDB creates a unique index on the _id field during the creation of a collection.
- The drop duplicates functionality on index creation is no longer supported since Mongo version 3.0
- Do not index every key. This will make inserts slow, take up lots of
- space, and probably not speed up your queries very much. Figure out
- what queries you are running, what the best indexes are for these queries,
- and make sure that the server is using the indexes you've created
- using the explain and hint tools described in the next section.

Types of Indexes-MongoDB Advanced Usage (contd.)

Index continue: explain()

- Explain will return information about the indexes used for the query (if any) and stats about timing and the number of documents scanned.

The important parts of this result are as follows:
"cursor" : "BasicCursor"
This means that the query did not use an index (unsurprisingly, because there was no query criteria). We'll see what this value looks like for an indexed query in a moment.
"nscanned" :
This is the number of documents that the database looked through. You want to make sure this is as close to the number returned as possible.
"n" :
This is the number of documents returned. We're doing pretty well here, because the number of documents scanned exactly matches the number returned. Of course, given that we're returning the entire collection, it would be difficult to do otherwise.
"millis" :
The number of milliseconds it took the database to execute the query. 0 is a good time to shoot for.



Hinting is usually unnecessary. MongoDB has a query optimizer and is very clever about choosing which index to use. When you first do a query, the query optimizer tries out a number of query plans concurrently. The first one to finish will be used, and the rest of the query executions are terminated. That query plan will be remembered for future queries on the same keys. The query optimizer periodically retries other plans, in case you've added new data and the previously chosen plan is no longer best. The only part you should need to worry about is giving the query optimizer useful indexes to choose from.

---

db.runCommand(*command*)¶-

Provides a helper to run specified database commands. This is the preferred method to issue database commands, as it provides a consistent interface between the shell and drivers.

Parameter-command

Type-document or string

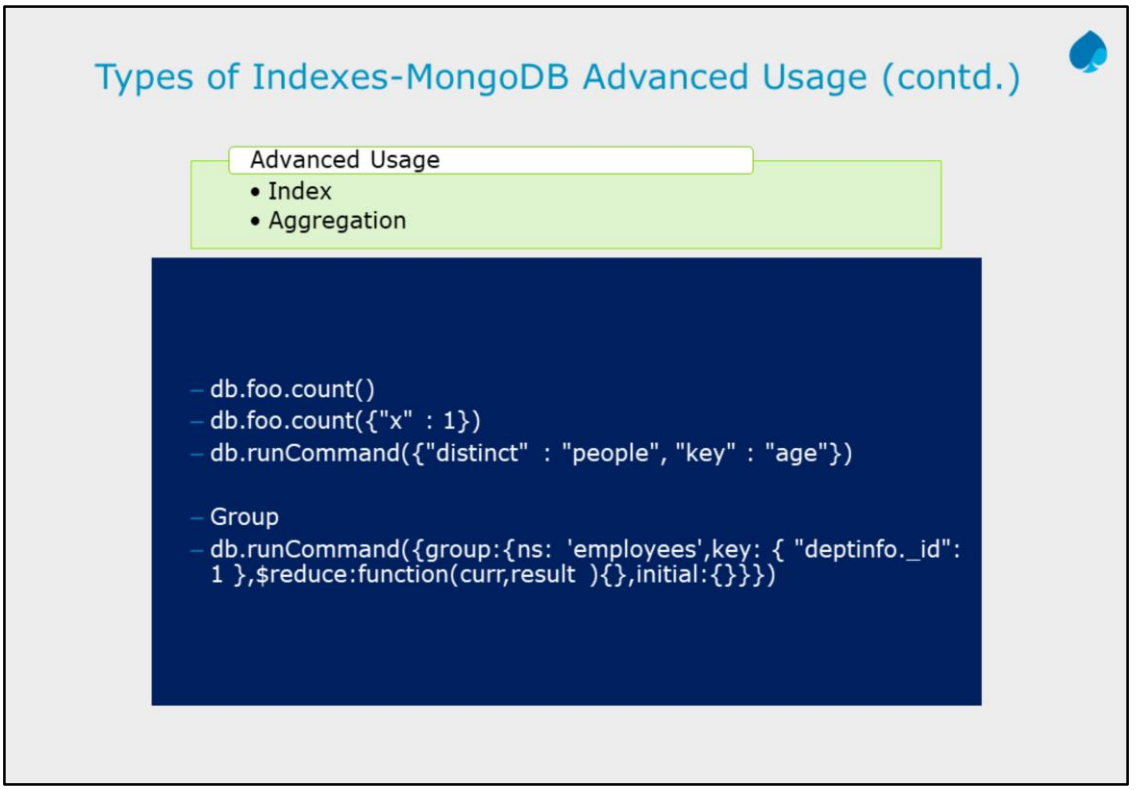
Description - A database command, specified either in document form or as a string. If specified as a string, db.runCommand() transforms the string into a document.

db.runCommand() runs the command in the context of the current database. Some commands are only applicable in the context of the admin database, and you must change your

db object to before running these commands or use db.adminCommand().

Response¶

The method returns a response document that contains the following fields:



The db.runCommand() accepts a document with the following fields:

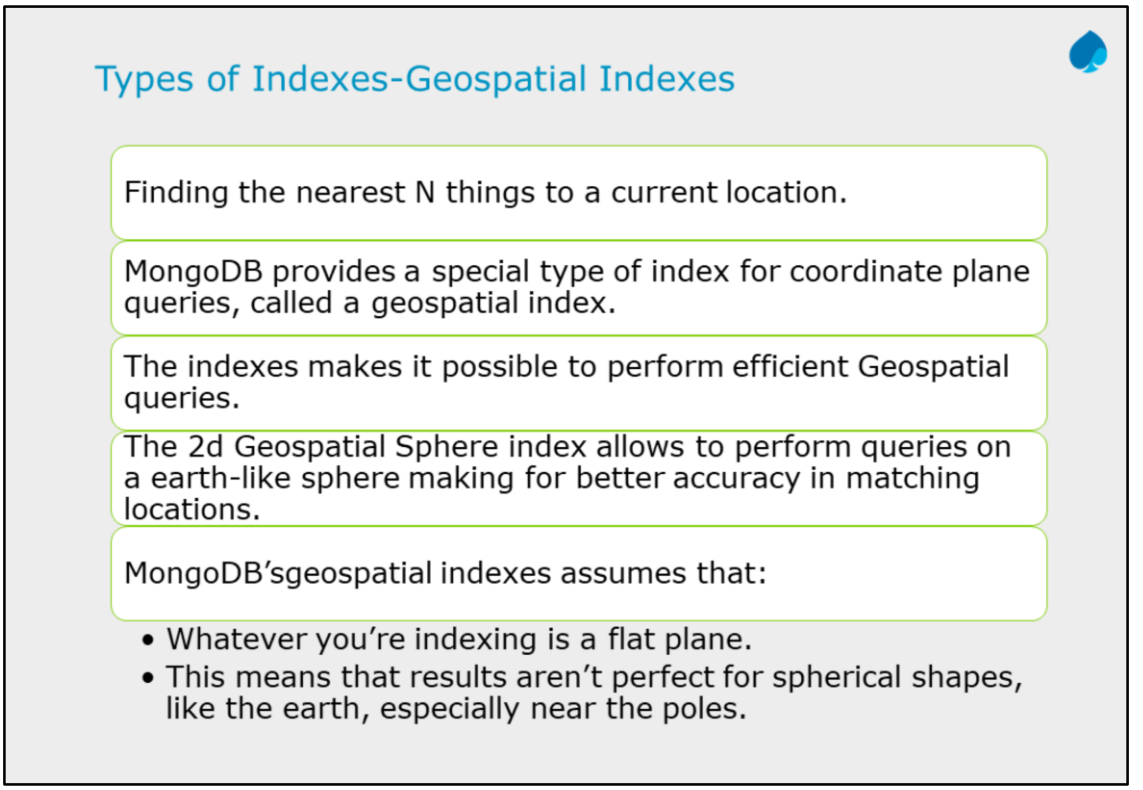
- Ns - string- The collection from which to perform the group by operation.
- Key – document-The field or fields to group. Returns a "key object" for use as the grouping key.
- $reduce – function -An aggregation function that operates on the documents during the grouping operation. These functions may return a sum or a count. The function takes two arguments: the current document and an aggregation result document for that group.
- Initial- document- Initializes the aggregation result document.
- $keyf – function - Optional. Alternative to the key field. Specifies a function that creates a "key object" for use as the grouping key. Use $keyf instead of key to group by calculated fields rather than existing document fields.
- Cond -document - Optional. The selection criteria to determine which documents in the collection to process. If you omit the cond field, group processes all the documents in the collection for the group operation.
- Finalize - function ---Optional. A function that runs each item in the result set before group returns the final value. This function can either modify the result document or replace the result document as a whole. Unlike the $keyf and
- $reduce fields that also specify a function, this field name is finalize, *not* $finalize.

The following Command will execute the distinct command and will display distinct firname.

```
>db.runCommand({"distinct" : "employees", "key" : "firstname"})
```

Group Department in employees collection

```
> db.runCommand({group:{ns: 'employees',key: { "deptinfo._id": 1 },$reduce:function(curr,result
){},initial:{}}})
```



MongoDB's 2d geospatial indexes.

Calculation of Geohash Values for 2d Indexes¶-

When you create a geospatial index on legacy coordinate pairs, MongoDB computes geohash values for the coordinate pairs within the specified location range and then indexes the geohash values.

To calculate a geohash value, recursively divide a two-dimensional map into quadrants. Then assign each quadrant a two-bit value. For example, a two-bit representation of four quadrants would be:

copycopied

01 11 00 10

These two-bit values (00, 01, 10, and 11) represent each of the quadrants and all points within each quadrant. For a geohash with two bits of resolution, all points in the bottom left quadrant would have a geohash of 00. The top left quadrant would have the geohash of 01. The bottom right and top right would have a geohash of 10 and 11, respectively.

To provide additional precision, continue dividing each quadrant into sub-quadrants. Each sub-quadrant would have the geohash value of the containing quadrant concatenated with the value of the sub-quadrant. The geohash for the upper-right quadrant is 11, and the geohash for the sub-quadrants would be (clockwise from the top left): 1101, 1111, 1110, and 1100, respectively.

Multi-location Documents for 2d Indexes¶

Note

2dsphere indexes can cover multiple geospatial fields in a document, and can express lists of points using MultiPoint embedded documents.

While 2d geospatial indexes do not support more than one geospatial field in a document, you can use a multi-key index to index multiple coordinate pairs in a single document. In the simplest example you may have a field (e.g. locs) that holds an array of coordinates, as in the following example:

copy

Types of Indexes-Geospatial Indexes (contd.)

A geospatial index can be created using the ensureIndex function, but by passing "2d" as a value instead of 1 or -1:

- > db.map.ensureIndex({"gps" : "2d"})
- "gps" : [0, 100] }
- { "gps" : { "x" : -30, "y" : 30 } }
- { "gps" : { "latitude" : -180, "longitude" : 180 } }
- > db.star.trek.ensureIndex({"light-years" : "2d"}, {"min" : -1000, "max" : 1000})
- > db.map.find({"gps" : {"$near" : [40, -73]}})
- > db.map.find({"gps" : {"$near" : [40, -73]}}).limit(10)

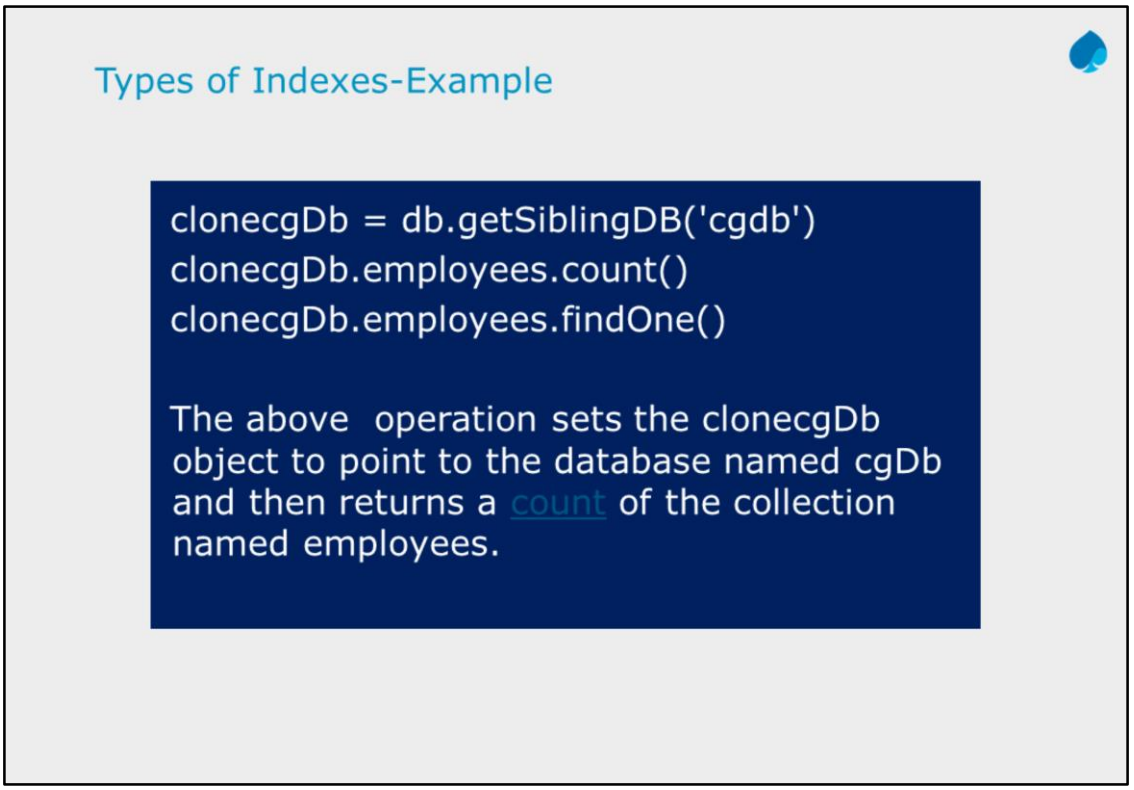
Create GeoSpatial Index

>db.products.ensureIndex({"orderInfo.address.coords": "2d" })

Find Out all the places new coordinates [8,4]

>db.products.find({"orderInfo.address.coords" : {"$near" : [8,4]}})

2d Index is supported in Mongo DB 2.2 and below version.



- A 2dsphere index supports queries that calculate geometries on an earth-like sphere.
- 2dsphere index supports all MongoDB geospatial queries: queries for inclusion, intersection and proximity.
- The 2dsphere index supports data stored as GeoJSON objects *and* legacy coordinate pairs
- For legacy coordinate pairs, the index converts the data to GeoJSON Point.

- MongoDB 3.2 introduces a version 3 of 2dsphere indexes.
- MongoDB 2.6 introduces a version 2 of 2dsphere indexes.

- To override the default version and specify a different version, include the option { "2dsphereIndexVersion": <version> } when creating the index.

---

db.getSiblingDB() as an alternative to the use <database> helper. This is particularly useful when writing scripts using the mongo shell where the use helper is not available.
Consider the following sequence of operations:

clonecgDb = db.getSiblingDB('cgdb')
clonecgDb.employees.count()
clonecgDb.employees.findOne()

The above operation sets the clonecgDb object to point to the database named cgDb and then returns a count of the collection named employees.

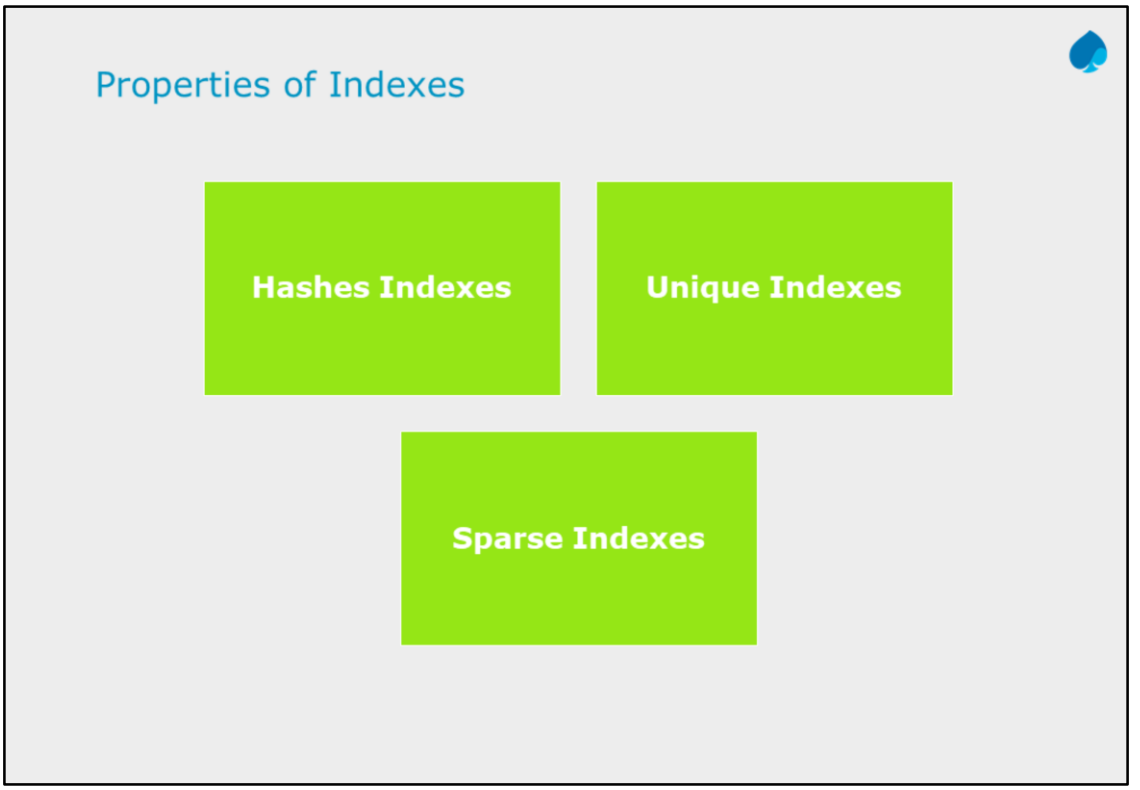
Types of Indexes-Text Indexes

MongoDB provides text indexes to support query operations that perform a text search of string content. text indexes can include any field whose value is a string or an array of string elements.

To index a field that contains a string or an array of string elements, include the field and specify the string literal "text" in the index document, as in the following example:

- db.employees.ensureIndex({lastname: "text" })

Properties of Indexes



Properties of Indexes-Hashed Indexes

Hashed indexes maintain entries with hashes of the values of the indexed field. The hashing function collapses embedded documents and computes the hash for the entire value but does not support multi-key (i.e. arrays) indexes.

Create a hashed index using an operation that resembles the following:

- db.employees.ensureIndex({ lastname: "hashed" })
- This operation creates a hashed index for the active collection on the a field.

Properties of Indexes-Hashed Indexes (contd.)

Unique Indexes

- The unique property for an index causes MongoDB to reject duplicate values for the indexed field. To create a unique index on a field that already has duplicate values.

```
db.collection.ensureIndex( { "a.b": 1 }, { unique: true } )
```

Create Unique index on last name column

>db.employees.ensureIndex({ lastname: "unique" },{"unique":true})

Properties of Indexes-Unique Index

Unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

By default, unique is false on MongoDB indexes.

To solve this issue MongoDB allows to add the *unique* option to the *ensureIndex* statement when created, making the target field unique across the entire collection.

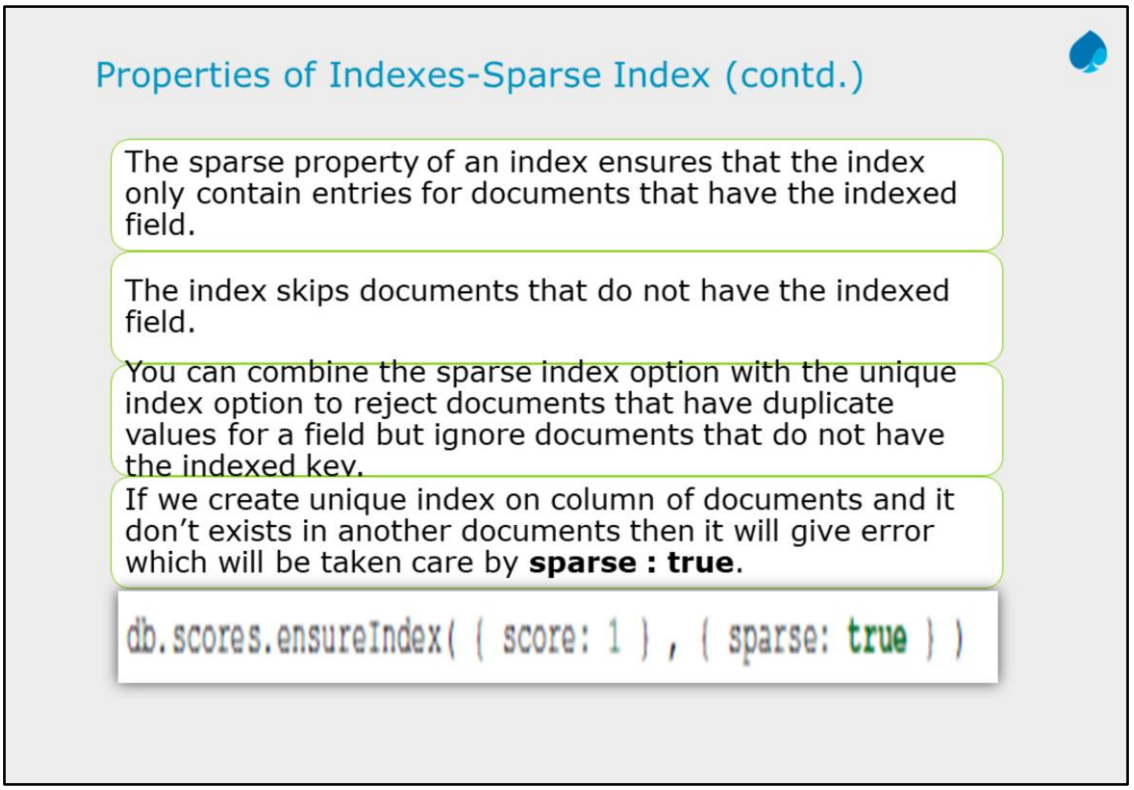
```
db.users.ensureIndex( { "userId": 1 }, { unique: true } )
```

Properties of Indexes-Sparse Index

Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value.

The index skips over any document that is missing the indexed field. The index is "sparse" because it does not include all documents of a collection.

By contrast, non-sparse indexes contain all documents in a collection, storing null values for those documents that do not contain the indexed field.



Create Sparse Index To ignore Null Values

>db.employees.ensureIndex({ lastname: 1 }, { sparse: true });

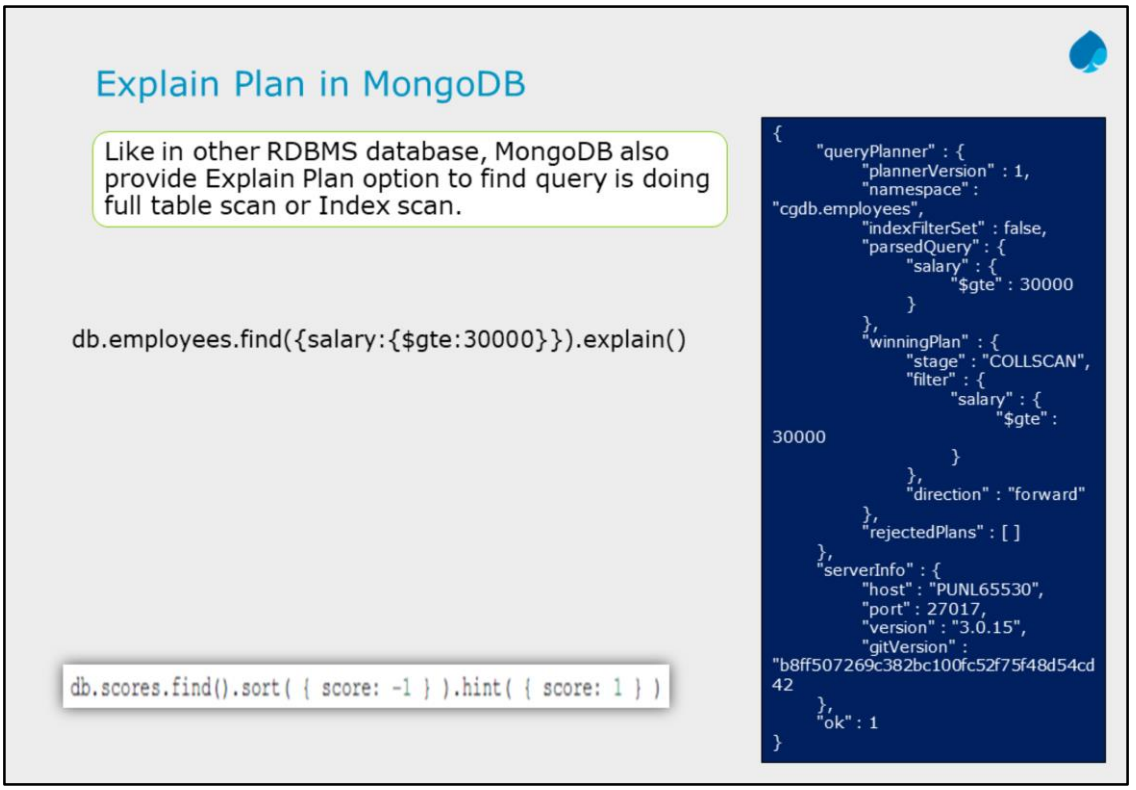
>db.employees.find().sort({ lastname: -1 }).hint({ lastname: 1 })

Properties of Indexes-TTL Indexes

TTL indexes are special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. This is ideal for certain types of information like machine generated event data, logs, and session information that only need to persist in a database for a finite amount of time.

Note : *By default, creating an index blocks all other operations on a database. When building an index on a collection, the database that holds the collection is unavailable for read or write operations until the index build completes. Any operation that requires a read or write lock databases and will wait for the foreground index build to complete.*

```
db.people.ensureIndex( { zipcode: 1}, {background: true} )
```



By default, db.collection.explain() runs in queryPlanner verbosity mode.

explain.queryPlanner -
Contains information on the selection of the query plan by the query optimizer.
explain.queryPlanner.namespace-
A string that specifies the namespace (i.e., <database>.<collection>) against which the query is run.
explain.queryPlanner.indexFilterSet-
A boolean that specifies whether MongoDB applied an index filter for the query shape.
explain.queryPlanner.winningPlan-
A document that details the plan selected by the query optimizer. MongoDB presents the plan as a tree of stages; i.e. a stage can have an inputStage or, if the stage has multiple child stages,
inputStages.
explain.queryPlanner.winningPlan.stage-
A string that denotes the name of the stage.
Each stage consists of information specific to the stage. For instance, an IXSCAN stage will include the index bounds along with other data specific to the index scan. If a stage has a child stage or multiple child stages, the stage will have an inputStage or inputStages.
explain.queryPlanner.winningPlan.inputStage¶ -
A document that describes the child stage, which provides the documents or index keys to its parent. The field is present *if* the parent stage has only

one child.

explain.queryPlanner.winningPlan.inputStages -An array of documents describing the child stages. Child stages provide the documents or index keys to the parent stage. The field is present *if* the parent stage has multiple child nodes. For example, stages for $or expressions or index intersection consume input from multiple sources.

explain.queryPlanner.rejectedPlans- Array of candidate plans considered and rejected by the query optimizer. The array can be empty if there were no other candidate plans.

Explain Plan in MongoDB-Why MongoDB: Performance

No Joins + No multi-row transactions

- = Fast Reads
- = Fast Writes (b/c you write to fewer tables, no trans. log)

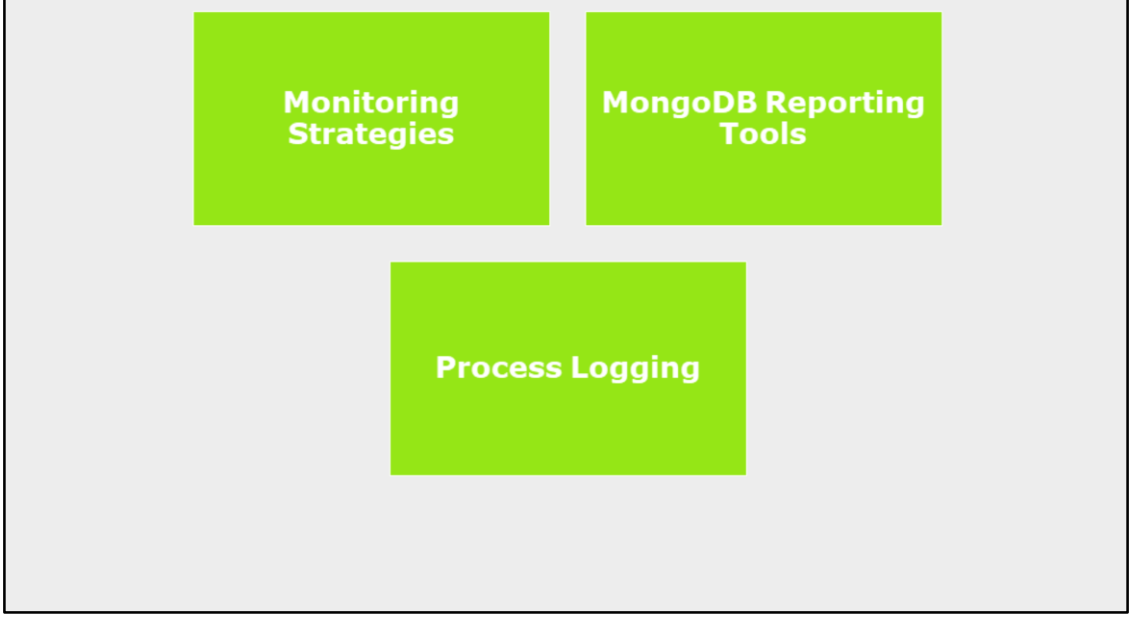
Async writes

- = you don't wait for inserts to complete.
- (optional, though)

Secondary Indexes

- = Index on embedded document fields for superfast ad-hoc queries.
- Indexes live in RAM.

Explain Plan in MongoDB-Monitoring for MongoDB



Explain Plan in MongoDB-MongoDB Strategies

There are three methods for collecting data about the state of a running MongoDB instance:

First, there is a set of utilities distributed with MongoDB that provides real-time reporting of database activities.

Second, ***database commands*** return statistics regarding the current database state with greater fidelity.

Third, **MongoDB Cloud Manager**, a hosted service, and **Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced**, provide monitoring to collect data from running MongoDB deployments as well as providing visualization and alerts based on that data.

Explain Plan in MongoDB- MongoDB Reporting Tools

Utilities

- The MongoDB distribution includes a number of utilities that quickly return statistics about instances' performance and activity. Typically, these are most useful for diagnosing issues and assessing normal operation.

Mongostat

Mongostat captures and returns the counts of database operations by type (e.g. insert, query, update, delete, etc.).

These counts report on the load distribution on the server.

Use **mongostat** to understand the distribution of operation types and to inform capacity planning.

Mongostat is functionally similar to the UNIX/Linux file system utility vmstat, but provides data regarding **mongod** and **mongos** instances.

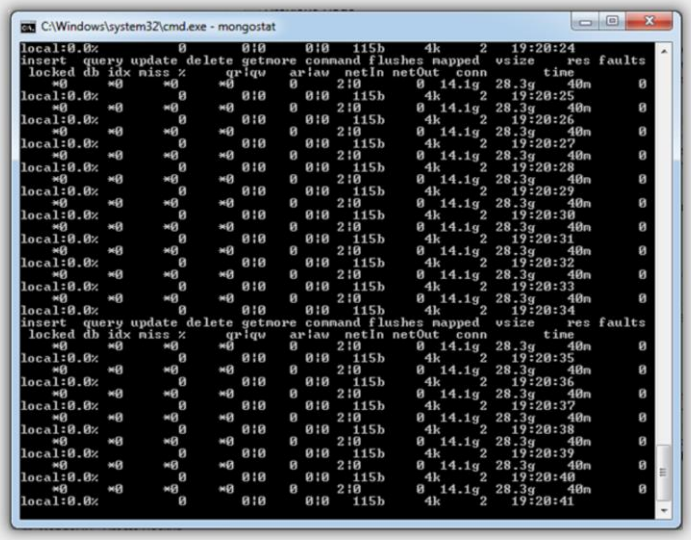
Mongostat (contd.)

Command also shows when you're hitting page faults, and showcase your lock percentage. This means that you're running low on memory, hitting write capacity or have some performance issue.

To run the command start your mongod instance. In another command prompt go to **bin directory of your mongodb** installation and type **mongostat.**

Example: D:\set up\mongodb\bin>mongostat

Mongostat (contd.)-Output of the Command



Mongotop Command

This command track and report the read and write activity of MongoDB instance on a collection basis.

By default mongotop returns information in each second, by you can change it accordingly.

We can check that this read and write activity matches your application intention, and you're not firing too many writes to the database at a time.

Reading too frequently from disk, or are exceeding your working set size.

Mongotop Command (contd.)

To run the command start your mongod instance.

In another command prompt go to **bin directory of your mongodb** installation and type **mongotop.**

```
D:\set up\mongodb\bin>mongotop 30
```

The above example will return values every 30 seconds.

ServerStatus Command

The **serverStatus** command, or **db.serverStatus()** from the shell, returns a general overview of the status of the database, detailing disk usage, memory use, connection, journaling, and index access.

The command returns quickly and does not impact MongoDB performance.

serverStatus outputs an account of the state of a MongoDB instance.

Dbstats Command

The **dbStats** command, or **db.stats()** from the shell, returns a document that addresses storage use and data volumes.

The **dbStats** reflect the amount of storage used, the quantity of data contained in the database, and object, collection, and index counters.

Collstats Command

The **collStats** or **db.collection.stats()** from the shell that provides statistics that resemble **dbStats** on the collection level, including a count of the objects in the collection, the size of the collection, the amount of disk space used by the collection, and information about its indexes.

ReplSetGetStatus Command

The **replSetGetStatus** command **(rs.status()** from the shell) returns an overview of your replica set's status. The ***replSetGetStatus*** document details the state and configuration of the replica set and statistics about its members.

Logging Slow queries-Process Logging

During normal operation, **mongod** and **mongos** instances report a live account of all server activity and operations to either standard output or a log file. The following runtime settings control these options.

Quiet:	Verbosity:	Path:	LogAppend:
Limits the amount of information written to the log or output.	Increases the amount of information written to the log or output. You can also modify the logging verbosity during runtime with the **logLevel** parameter or the **db.setLogLevel( )** method in the shell.	Enables logging to a file, rather than the standard output. You must specify the full path to the log file when adjusting this setting.	Adds information to a log file instead of overwriting the file.

Profiling

The database profiler collects fine grained data about MongoDB write operations, cursors, database commands on a running **mongod** instance.

You can enable profiling on a per-database or per-instance basis.

The ***profiling level*** is also configurable when enabling profiling. The profiler is *off* by default.

Profiling-Profiling Levels

The database profiler writes all the data it collects to the **system.profile** collection, which is a ***capped collection***.

Profiling Levels - The following profiling levels are available:

- **0:** The profiler is off, does not collect any data. **Mongod** always writes operations longer than the **slowOpThresholdMs** threshold to its log. This is the default profiler level.
- **1:** Collects profiling data for slow operations only. By default slow operations are those slower than 100 milliseconds.
- You can modify the threshold for "slow" operations with the **slowOpThresholdMs** runtime option or the **setParameter** command. See the ***Specify the Threshold for Slow Operations*** section for more information.
- **2:** Collects profiling data for all database operations.

Profiling-Enable Database Profiling and Set the Profiling Level

You can enable database profiling from the **mongo** shell or through a driver using the **profile** command.

When you enable profiling, you also set the ***profiling level***. The profiler records data in the **system.profile** collection.

MongoDB creates the **system.profile** collection in a database after you enable profiling for that database.

To enable profiling and set the profiling level, use the **db.setProfilingLevel()** helper in the **mongo** shell, passing the profiling level as a parameter.

Profiling-Enable Database Profiling and Set the Profiling Level (contd.)

To enable profiling for all database operations, consider the following operation in the **mongo** shell:

- **db.setProfilingLevel(2)**

The shell returns a document showing the *previous* level of profiling. The "ok" : 1 key-value pair indicates the operation succeeded:

- **{ "was" : 0, "slowms" : 100, "ok" : 1 }**

Profiling-Thresholds for Slow Operations

The threshold for slow operations applies to the entire **mongod** instance. When you change the threshold, you change it for all databases on the instance.

By default the slow operation threshold is 100 milliseconds. Databases with a profiling level of 1 will log operations slower than 100 milliseconds.

To change the threshold, pass two parameters to the **db.setProfilingLevel()** helper in the **mongo** shell. The first parameter sets the profiling level for the current database, and the second sets the default slow operation threshold *for the entire* **mongod** *instance*.

Profiling-Setting Profiling Level

The following command sets the profiling level for the current database to 0, which disables profiling, and sets the slow-operation threshold for the **mongod** instance to 20 milliseconds.

Any database on the instance with a profiling level of 1 will use this threshold:

- **db.setProfilingLevel(0,20)**

Profiling-Checking Profiling Level

To view the ***profiling level***, issue the following from the **mongo** shell:

- **db.getProfilingStatus()**

The shell returns a document similar to the following:

- **{ "was" : 0, "slowms" : 100 }**

The was field indicates the current level of profiling.

The slowms field indicates how long an operation must exist in milliseconds for an operation to pass the "slow" threshold. MongoDB will log operations that take longer than the threshold if the profiling level is 1.

Profiling-Get Profiling Level

To return only the profiling level, use the **db.getProfilingLevel()** helper in the **mongo** as in the following:

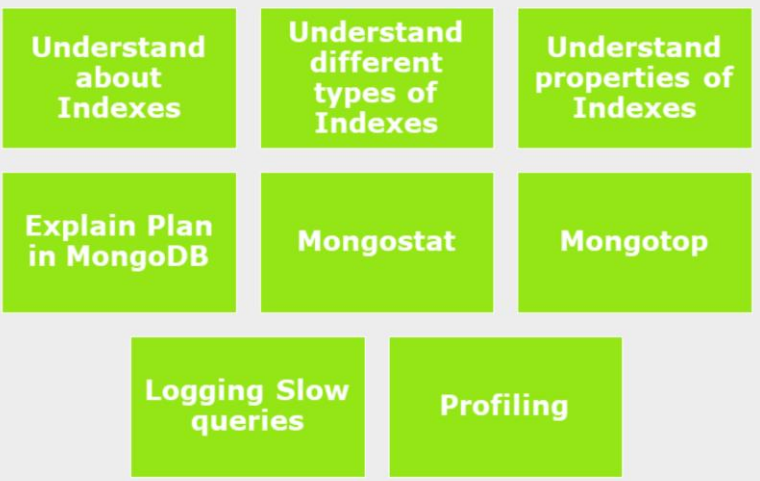
- db.getProfilingLevel()

Profiling-Disable Profiling

To disable profiling, use the following helper in the **mongo** shell:

- db.setProfilingLevel(0)

Summary


Understand about Indexes
Understand different types of Indexes
Understand properties of Indexes
Explain Plan in MongoDB
Mongostat
Mongotop
Logging Slow queries
Profiling