MongoDB Index

CS185C: Introduction to NoSQL Databases
Suneuy Kim

References

- https://docs.mongodb.com/manual/indexes/
- https://docs.mongodb.com/v3.2/reference/explain-results/

Indexes

- A data structure that collects information about the values of specified fields in the documents of a collection.
- An index as a predefined query that was executed and had its results stored.
- Adding an index potentially increases query speed, but it reduces insertion or deletion speed. (Maximum 64 indexes per collection)
- Beneficial when the number of reads is higher than the number of writes.

Default _id index

- MongoDB creates a unique index on the _id field during the creation of a collection.
- The _id index prevents clients from inserting two documents with the same value for the _id field.
- You cannot drop this index on the _id field.

Index Administration

- Create Indexes
- List Indexes
- Change indexes

Create an Index

```
db.collection.createIndex(keys, options)
```

- Keys: a document `{index_key:value}`
  - For ascending index and descending index, specify 1 and -1 , respectively.
- Options: background, unique, name, partialFilterExpression, sparse, etc

Get the index name or the index specification document

```
db.test.getIndexes()
[   {
            "v" : 1,
            "key" : {
                  "_id" : 1
            },
            "name" : "_id_",
            "ns" : "test.test"
      },
      {
            "v" : 1,
            "key" : {
                  "ratings" : 1
            },
            "name" : "ratings_1",
            "ns" : "test.test"
      }
]
```

Drop an index

`db.collection.dropIndex(index)`

Drops or removes the specified index from a collection.

Cannot drop "_id_"

Example:

`db.test.dropIndex("rating_1")` // using the index name

`db.test.dropIndex({"ratings" : 1})` // using the index specification document

Index Types

- Single Field Indexes
- Compound Index
- Multikey Index
- Unique Index
- Sparse Index
- Partial Index
- TTL
- Full Text Index
- Geospatial Index

Single Field Indexes

Consider a collection called records consisting of documents looking like this:

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

Single Field Indexes

```
db.records.createIndex({score: 1 } )
```

Supports queries like these:

- `db.records.find({score:2})`
- `db.records.find({score:{$gt: 10}})`

Create an Index on Embedded **Fields**

```
db.records.createIndex({"location.state":1})
```

Supports queries like these:

- `db.records.find( { "location.state": "CA" } )`
- `db.records.find( { "location.city": "Albany", "location.state": "NY" } )`

Create an Index on Embedded **Document**

```
db.records.createIndex( { location: 1 } )
```

- Supports queries like this:

```
db.records.find({location:{state:"NY",city: "New York"} } )
```

- When performing equality matches on embedded documents, field order matters and the embedded documents must match exactly. For example, the following query cannot find the document.

```
db.records.find({location:{city:"New York", state: "NY" } } )
```

Compound Index: Running example

```
db.comments.insertOne({timestamp:1, anonymous:false, rating:3})
db.comments.insertOne({timestamp:2, anonymous:false, rating:5})
db.comments.insertOne({timestamp:3, anonymous:true, rating:1})
db.comments.insertOne({timestamp:4, anonymous:false, rating:2})
```

Ref: Three-Step Compound Indexes By A. Jesse Jiryu Davis

Full Collection Scan without Index

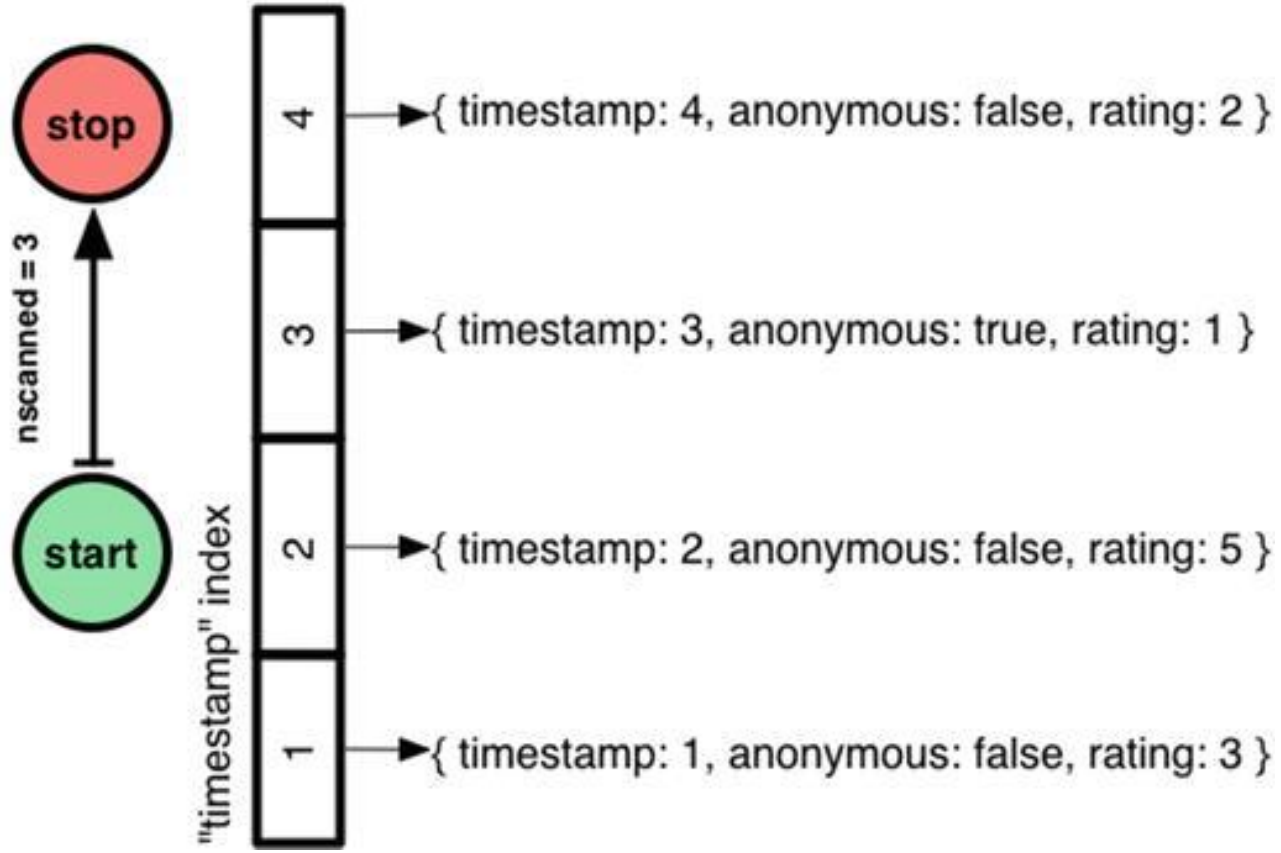
```
db.comments.find({timestamp: {$gte:2,$lte:4}}).explain(true)
```

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 3,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 0,      // nScanned
        "totalDocsExamined" : 4,      // totalDocs Scanned
        "executionStages" : {
              "stage" : "COLLSCAN", // Basic Cursors
```

Note: No child stage exists for this execution stages.

With Index `db.comments.createIndex( { timestamp: 1 } )`

- totalKeysExamined(nscanned) the number of index keys in the range that Mongo scanned
- totalDocsExamined(nscannedObjects) : the number of documents it looked at to get the final result
- nReturned (n): the number of the returned documents



totalKeysExamined >= totalDocsExamined >= nReturned
With an ideal index for the query, these three are equal.

With Index `db.comments.createIndex( { timestamp: 1 } )`

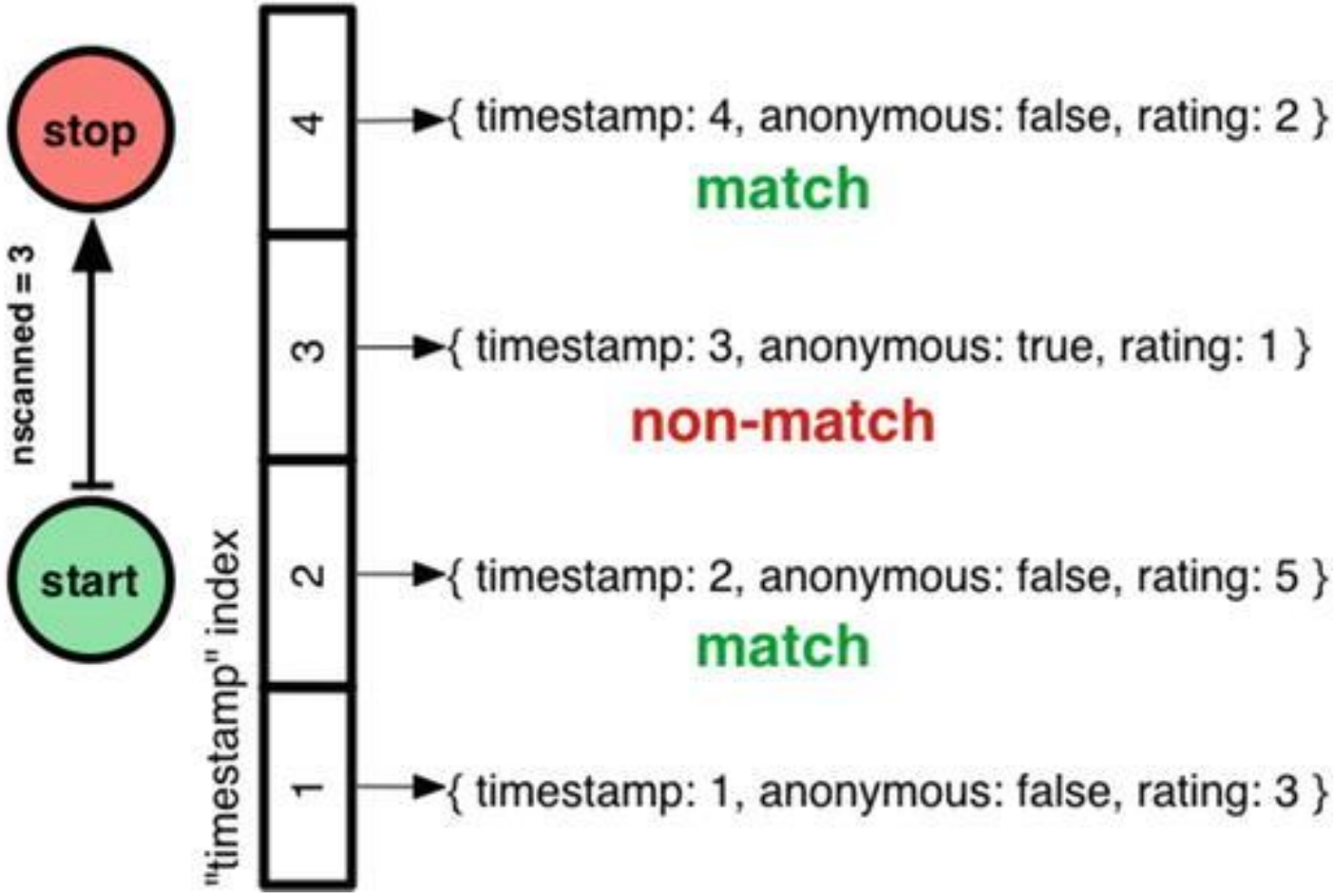
```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 3,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 3,
        "totalDocsExamined" : 3,
        "executionStages" : {
            "stage" : "FETCH",
            "nReturned" : 3,
. . .
```

```
"inputStage" : {
        "stage" : "IXSCAN", // BtreeCursors
        "nReturned" : 3,
. . .
        "works" : 4,
        "advanced" : 3,
        "direction" : "forward",
        "indexBounds" :
         { "timestamp" : [ "[2.0, 4.0]" ] },
         "keysExamined" : 3,
```

- totalDocsExamined includes at least all the documents returned, even if Mongo could tell just by looking at the index that the document was definitely a match.
- inputStage is a child of executionStats

Equality Plus Range Query
with db.comments.createIndex({ timestamp: 1 })

```
db.comments.find({ timestamp: { $gte: 2, $lte: 4 }, anonymous: false }).explain(true)
```



Equality Plus Range Query with db.comments.createIndex({ timestamp: 1 })

```
db.comments.find({ timestamp: { $gte: 2, $lte: 4 }, anonymous: false }).explain(true)
```

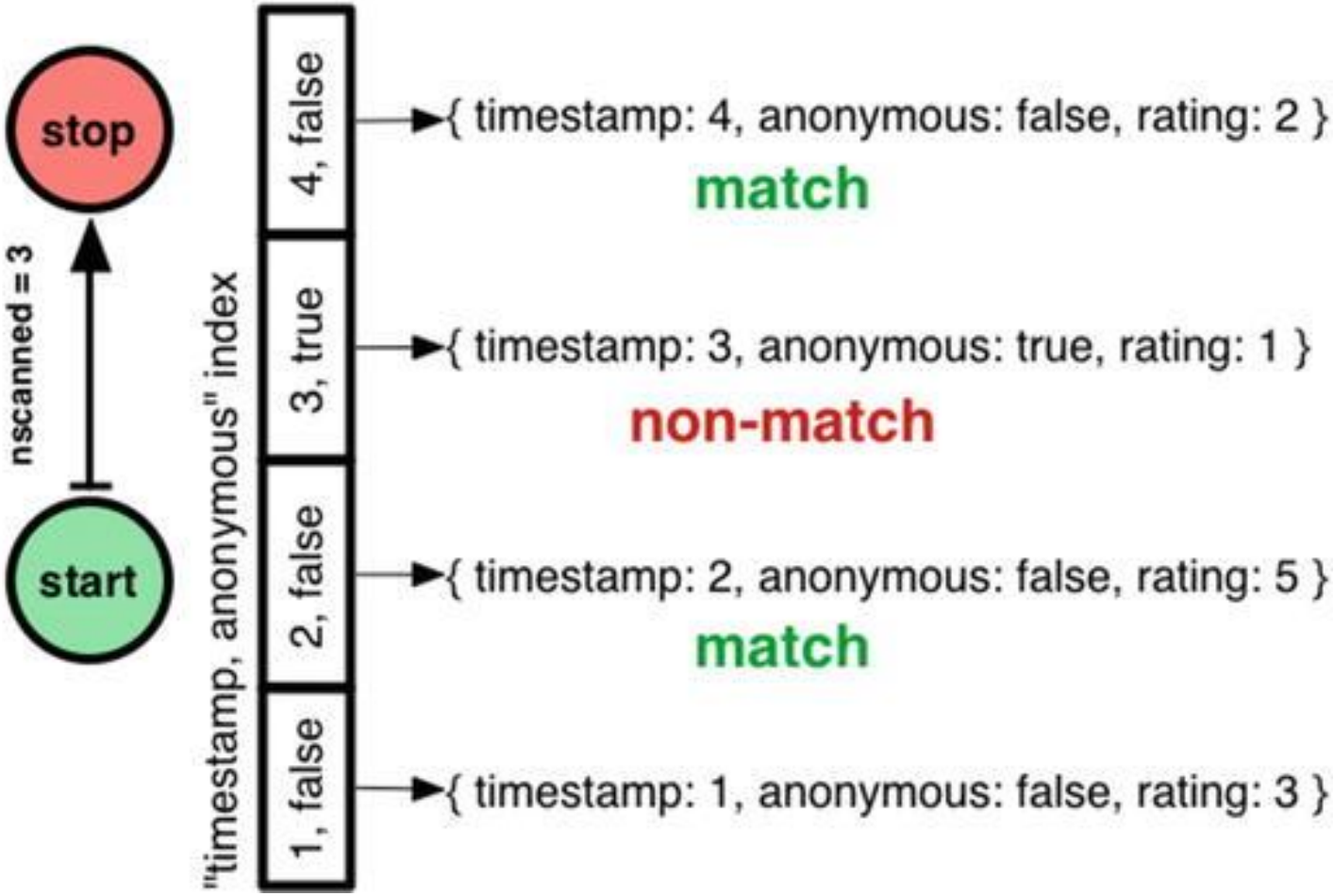
```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 2,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 3,
        "totalDocsExamined" : 3,
        "executionStages" : {
            "stage" : "KEEP_MUTATIONS",
            "nReturned" : 2,
            "inputStage" : {
                "stage" : "FETCH",
                "filter" : {
                    "anonymous" : {
                        "$eq" : false
                    }
                },
                "nReturned" : 2,
. . . "inputStage" : {
                    "stage" : "IXSCAN",
                    "nReturned" : 3,
```

Note: totalKeysExamined > nReturned due to the filtering done by anonymous: false.

Equality Plus Range Query with a Compound Index

```
db.comments.createIndex( { timestamp:1, anonymous:1 } )
```

```
db.comments.find({ timestamp: { $gte: 2, $lte: 4 }, anonymous: false }).explain(true)
```



Equality Plus Range Query with a Compound Index

```
db.comments.createIndex( { timestamp:1, anonymous:1 } )
```

```
db.comments.find({ timestamp: { $gte: 2, $lte: 4 }, anonymous: false }).explain(true)
```

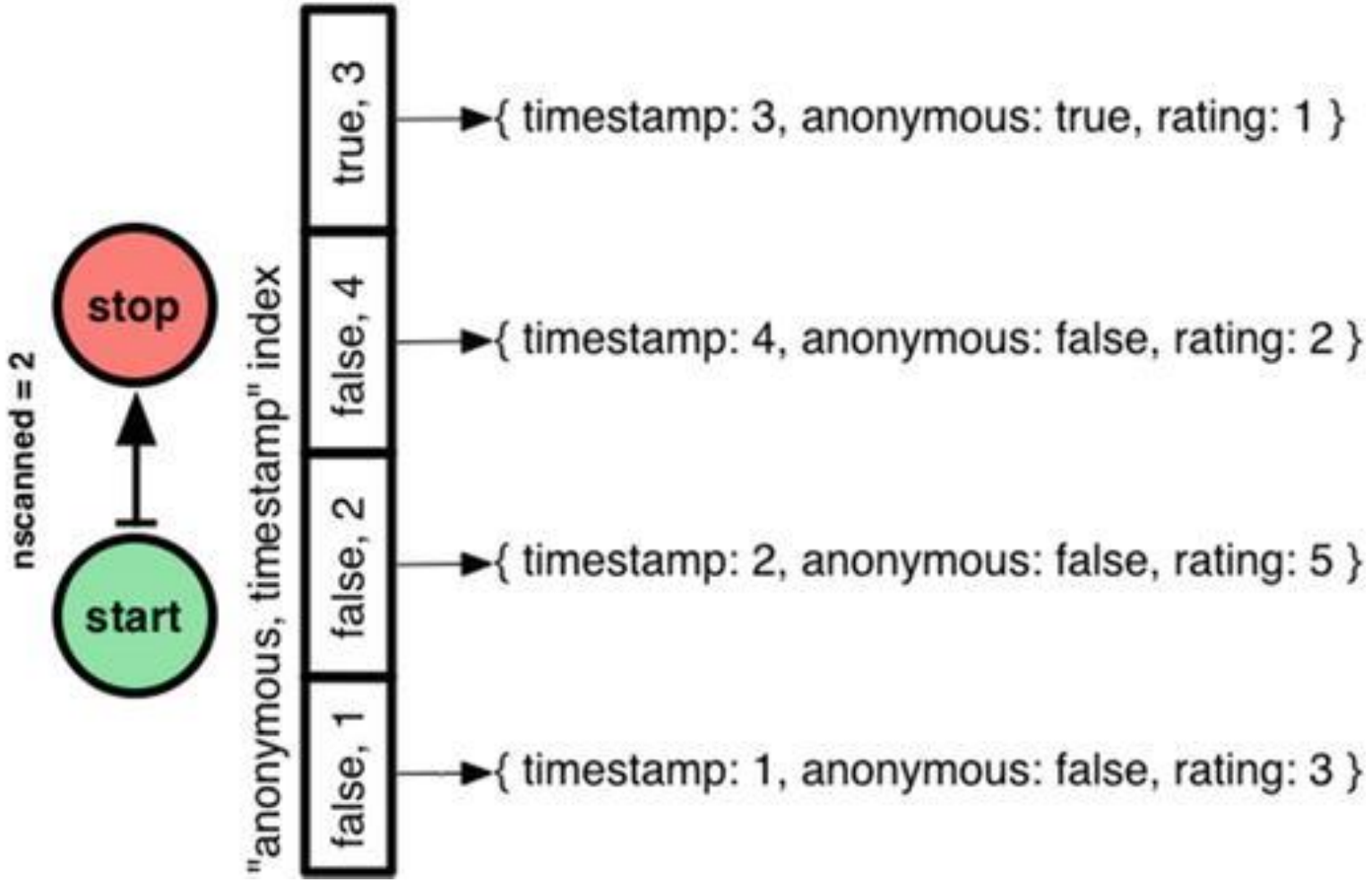

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 2,
        "executionTimeMillis" : 84,
        "totalKeysExamined" : 3,
        "totalDocsExamined" : 2,
        "executionStages" : {
            "stage" : "FETCH",
            "nReturned" : 2,
            "inputStage" : {
                "stage" : "IXSCAN",
                "nReturned" : 2,
                "keyPattern" : {"timestamp" : 1,
                                "anonymous" : 1 },
                "indexBounds" : {
                    "timestamp" : ["[2.0, 4.0]"],
                    "anonymous" : [
                                "[false, false]"
                            ]
                        },
```


Note: totalDocsExamined is dropped from 3 to 2. However, totalKeysExamined is still 3 because the index keys 2, 3, 4 still have to be examined. Would the order of index keys matter?

Compound key: order matters

```
db.comments.createIndex( { anonymous:1, timestamp:1 } )
```

```
db.comments.find({ timestamp: { $gte: 2, $lte: 4 }, anonymous: false }).explain(true)
```



Compound key: order matters

```
db.comments.createIndex( { anonymous:1, timestamp:1 } )
```

```
db.comments.find({ timestamp: { $gte: 2, $lte: 4 }, anonymous: false }).explain(true)
```


```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 2,
        "executionTimeMillis" : 1,
        "totalKeysExamined" : 2,
        "totalDocsExamined" : 2,
        "executionStages" : {
            "stage" : "FETCH",
            "nReturned" : 2,
            "inputStage" : {
                "stage" : "IXSCAN",
                "nReturned" : 2,
                "keyPattern" : {
                    "anonymous" : 1,
                    "timestamp" : 1
                },
                "indexBounds" : {
                    "anonymous" : ["[false, false]" ],
                    "timestamp" : ["[2.0, 4.0]"]
                },
                "keysExamined" : 2,
```

Equality, Range Query and Sort

```
db.comments.find({timestamp:{$gte:2,$lte:4},anonymous:false}).sort({rating:-1}).explain(true)
```

db.comments.createIndex({anonymous:1, timestamp:1})

- Finds all the results using the given index, batches them up in memory, sort and return them.
  - Costs RAM and CPU on the DB server
  - Costs RAM on application server
  - 32MB limits on data to sort them on RAM
- "totalKeysExamined" : 2,

  "totalDocsExamined" : 2
- scanAndorder.txt

db.comments.createIndex({anonymous:1, rating:1})

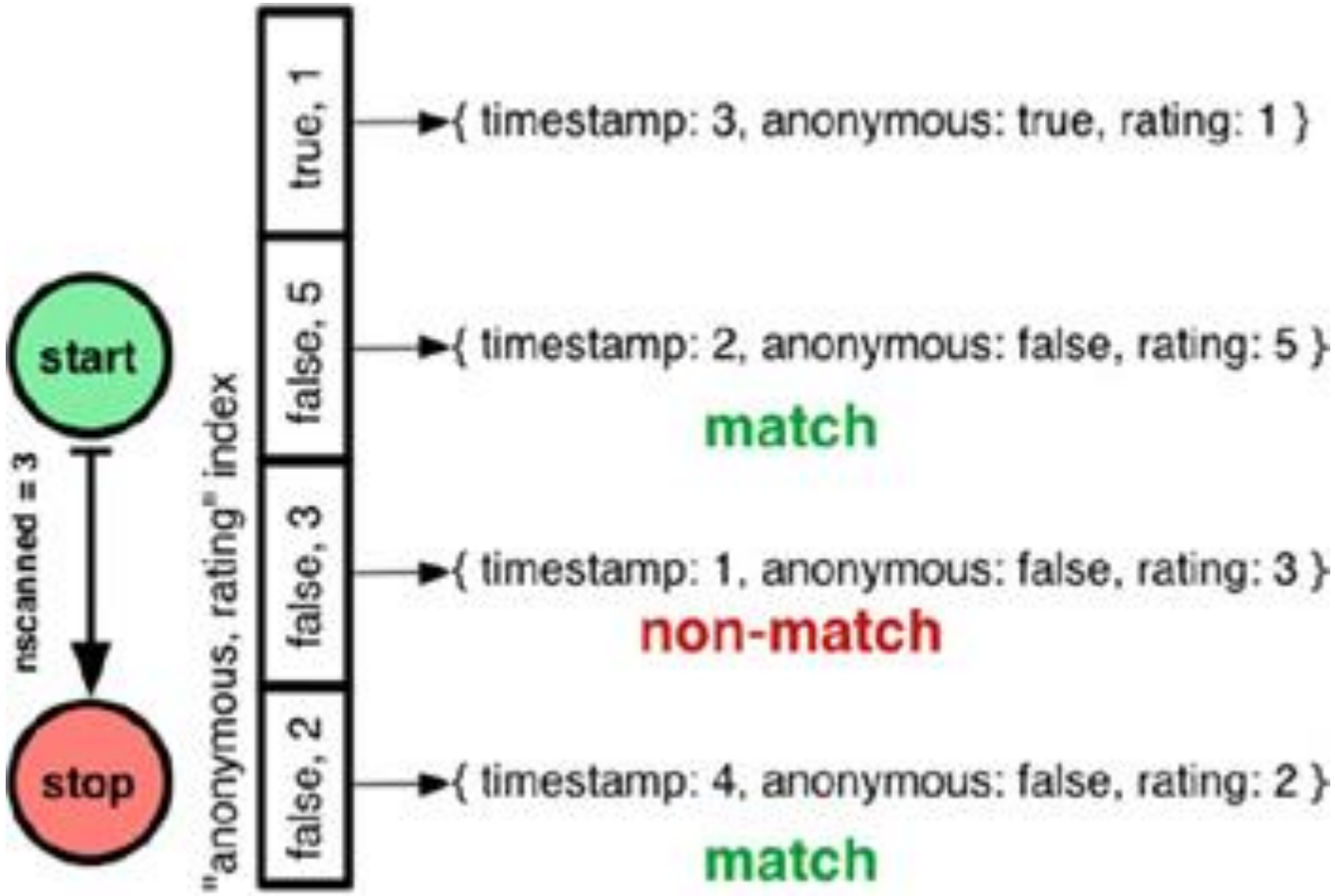
- The given index preordered data in rating.
- "totalKeysExamined" : 3,

  "totalDocsExamined" : 3
- preOrdered.txt

Equality, Range Query, and Sort

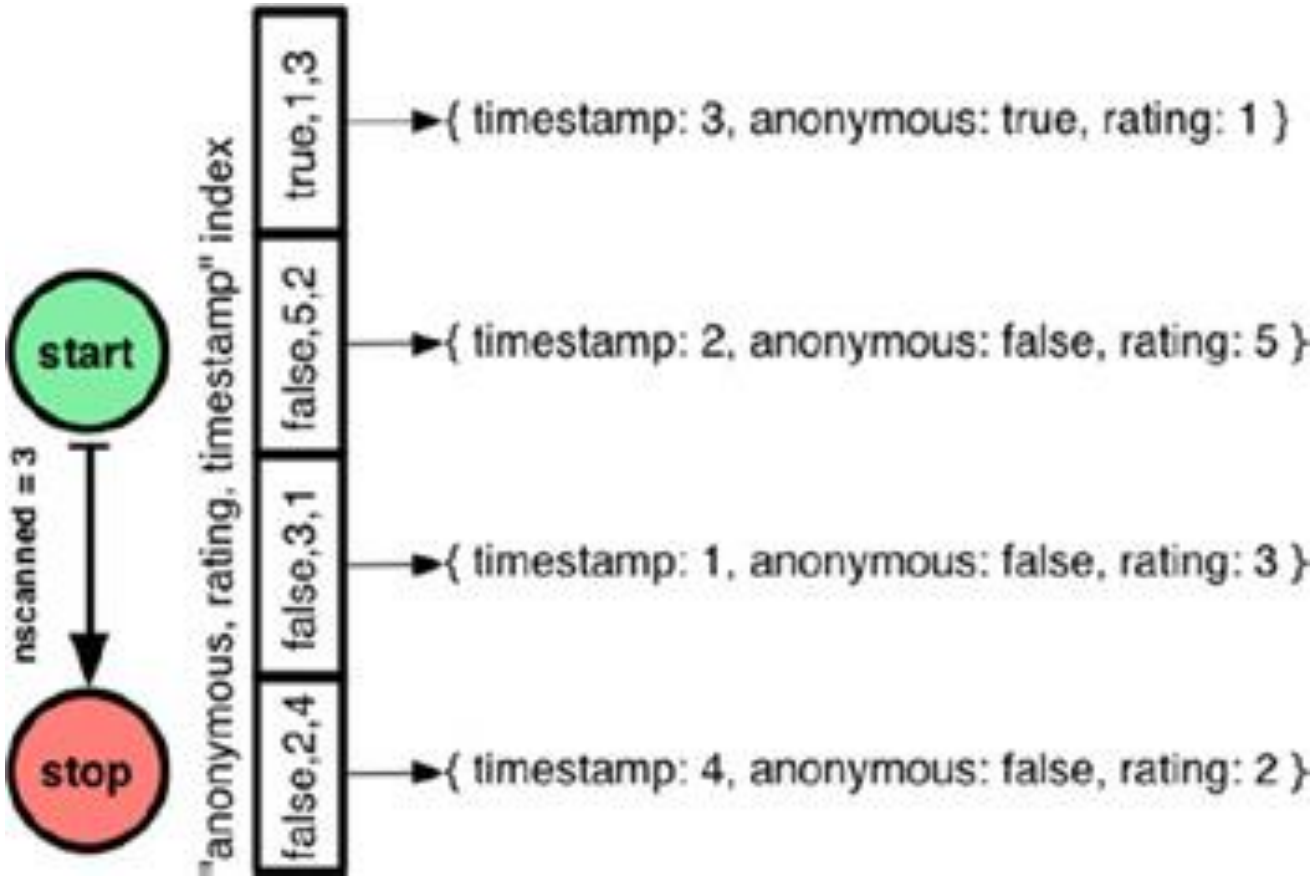
```
db.comments.find({timestamp:{$gte:2,$lte:4},anonymous:false}).sort({rating:-1}).explain(true)
```

db.comments.createIndex({anonymous:1, rating:1})



db.comments.createIndex({ anonymous: 1, rating: 1, timestamp: 1 })

```
db.comments.find({timestamp:{$gte:2,$lte:4},anonymous:false}).sort({rating:-
1}).explain(true)
```



"nReturned" : 2
"totalKeysExamined" : 3,
"totalDocsExamined" :2

Multikey Indexes to search for specific array element efficiently

To index arrays with scalar values

```
{ _id: 1, item: "ABC", ratings: [ 2, 5, 9 ] }

db.survey.createIndex( { ratings: 1 } )
```

An index key is created for each element in the array.
Since the ratings field contains an array, the index on ratings is multikey. The multikey index contains the following three index keys, each pointing to the same document:

- 2,
- 5, and
- 9.

Multikey Index Bounds

- Consider a collection survey containing the following documents:

{ _id: 1, item: "ABC", ratings: [2, 9] }

{ _id: 2, item: "XYZ", ratings: [4, 3] }

- Create a multikey index on the ratings array

db.survey.createIndex({ ratings: 1 })

- Consider the following two queries:

  Q1: db.survey.find({ ratings : { $elemMatch: { $gte: 3, $lte: 6 } } })

  Index Bound: ratings: [[3, 6]] (Intersect Bounds for Multikey Index)

  A1: { _id: 2, item: "XYZ", ratings: [4, 3] }

  Q2: db.survey.find({ ratings : { $gte: 3, $lte: 6 } })

  Index Bounds: either "ratings" : ["[-inf.0, 6.0]"] or "ratings" : ["[3.0, inf.0]"]

  A2:

  { _id: 1, item: "ABC", ratings: [2, 9] }

  { _id: 2, item: "XYZ", ratings: [4, 3] }

Query on the Array Field as a Whole

- Cannot use the multikey index scan to find the whole array.
- Using the multikey index to look up the first element of the query array, MongoDB retrieves the associated documents and filters for documents whose array matches the array in the query.
- Example

With db.survey.createIndex({ ratings: 1 }), to answer

db.survey.find({ ratings: [5, 9] }),

Use the multikey index to find documents that have 5 at any position in the ratings array, retrieves these documents and filters for documents whose ratings array equals the query array [5, 9].

Compound Multikey Index: a compound index where one of the indexed field is an array

- With db.survey.createIndex({ item: 1, ratings: 1 })

Consider db.survey.find({ item: "XYZ", ratings: { $gte: 3 } })

the bounds for the item: "XYZ" predicate are [["XYZ", "XYZ"]];

the bounds for the ratings: { $gte: 3 } predicate are [[3, Infinity]].

Combined index bounds of

{ item: [["XYZ", "XYZ"]], ratings: [[3, Infinity]] }

Compound Multikey Indexes

- To index arrays holding embedded documents

```
{
   _id: 1,
   item: "abc",
   stock: [ { size: "S", color: "red", quantity: 25 }, { size: "S",
color: "blue", quantity: 10 }, { size: "M", color: "blue", quantity:
50 } ]
}
```

db.inventory.createIndex({ "stock.size": 1, "stock.quantity": 1 })

- The compound multikey index can support queries like these:

db.inventory.find({ "stock.size": "M" })

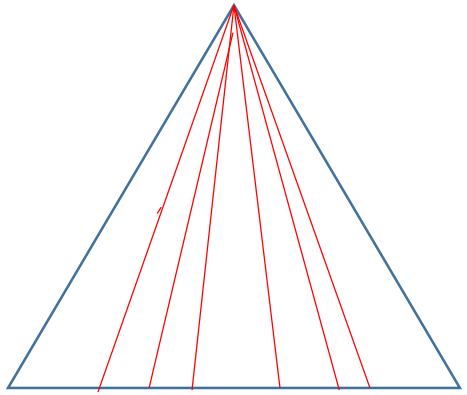
db.inventory.find({ "stock.size": "S", "stock.quantity": { $gt: 20 } })

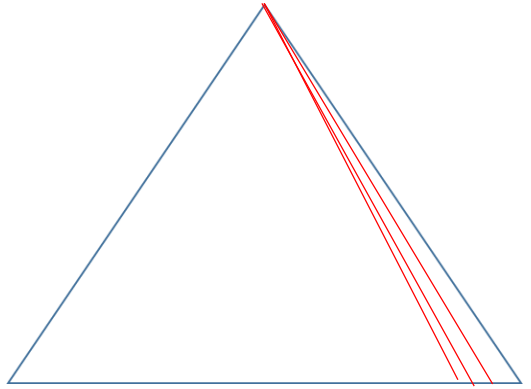
Compound Multikey indexes

Restriction on indexing parallel arrays.

```
> db.multi.insert({"x" : [1, 2, 3], "y" : [4, 5, 6]})
WriteResult({ "nInserted" : 1 })
> db.multi.createIndex({"x" : 1, "y" : 1})
{
        "ok" : 0,
        "errmsg" : "cannot index parallel arrays [y] [x]",
        "code" : 10088
}
```

Right Balanced Index Access





The entire index has to be in RAM.

Only small portion (right most branches) needs to be in RAM.

Example: If a DATE is the first index key in a compound index, the right most branches represent most recent data. For applications that tend to use recent data more than older data, MongoDB only has to keep the right most branches of the index in RAM.

Implicit Indexes

- If an index has N keys, you get a “free” index on any prefix of those keys. Suppose we have an index that looks like {"a": 1, "b": 1, "c": 1, ..., "z": 1}, we effectively have indexes on {"a": 1}, {"a": 1, "b" : 1}, {"a": 1, "b": 1, "c": 1}, and so on.
- Example: If we have an index on {"age" : 1, "username" : 1}, an index on {"age": 1} is implied.
- Not for *any* subset of keys: For example, indexes on {"b":1} or {"a": 1, "c": 1} are not implied.

Unique Indexes

- Each value must appear at most once in the index
- Usage: If you want to make sure no two documents can have the same value for a field
- Example

```
> db.users.createIndex({"username" : 1}, {"unique" : true})
> db.users.insert({username: "bob"})
WriteResult({ "nInserted" : 1 })
> db.users.insert({username: "bob"})
WriteResult({
        "nInserted" : 0,
        "writeError" : {
                "code" : 11000,
                "errmsg" : "E11000 duplicate key error index:
test.users.$username_1 dup key: { : \"bob\" }"
        }
})
```

Unique Indexes

- If a key does not exist, the index stores its value as null for that document. (Remember, not all documents have the indexed field.)
- Inserting more than one document that is missing the indexed field will fail a document with a value of null already exists.
- Use a sparse Index

Unique Indexes

- If an index entry exceeds 1024 bytes., it won't be indexed.
- It makes a document "invisible" to queries that use the index.
- This implies that the keys longer than 1 KB will not be subject to the unique index constraints.
- Compound unique indexes: individual keys can have the same values, but the combination of values across all keys in an index an appear in the index at most once.

```
db.members.createIndex( { groupNumber: 1,
lastname: 1, firstname: 1 }, { unique: true
} )
```

Sparse Indexes

- Indexes that need not include every document as an entry

```
db.addresses.createIndex( { "xmpp_id": 1 }, { sparse: true } )
```

The index does not index documents that do not include the xmpp_id field. By default, a sparse index is non-unique.

- Unique sparse index

```
db.users.createIndex({"email" : 1}, {"unique" : true, "sparse" : true})
```

Use a unique sparse index if you want to enforce the unique index only if the indexed key exists.

- Sparse index in MongoDB is different form sparse index (block level index) of RDBMS.

With/without a sparse index, the results of the same query can be different.

```
> db.foo.find()
{ "_id" : 0 }
{ "_id" : 1, "x" : 1 }
{ "_id" : 2, "x" : 2 }
{ "_id" : 3, "x" : 3 }
```

Without a sparse index

```
{ "_id" : 0 }
{ "_id" : 1, "x" : 1 }
{ "_id" : 3, "x" : 3 }
```

With

```
db.foo.createIndex({"x":1},
{"sparse":true})
db.foo.find({"x":{"$ne":2}})
.hint({"x":1})
{ "_id" : 1, "x" : 1 }
{ "_id" : 3, "x" : 3 }
```

Query:

```
db.foo.find({"x":{"$ne":2}})
```

Without a hint, MongoDB is aware of that a data can be missing in the result and chooses CollectionScan, not IndexScan

Partial Indexes

- Only index the documents in a collection that meet a specified filter expression with the new partialFilterExpression option.
- The partialFilterExpression option accepts a document that specifies the filter condition using:
  - equality expressions (i.e. field: value or using the $eq operator),
  - $exists: true expression,
  - $gt, $gte, $lt, $lte expressions,
  - $type expressions,
  - $and operator at the top-level only
- Lower storage requirements and reduced performance costs for index creation and maintenance by indexing a subset of documents in a collection.

Partial Indexes

```
db.restaurants.createIndex(
    { cuisine: 1},
    { partialFilterExpression: {rating:{ $gt: 5 }}}
)
```

- Indexes only the documents with a rating field greater than 5.
- To use the partial index, a query must contain the filter expression as part of its query condition.

```
db.restaurants.find({cuisine:"Italian",rating:
{ $gte: 8 } } )
```

Partial Indexes

- MongoDB will not use the partial index for a query if using the index results in an incomplete result set.
- Examples

```
db.restaurants.find({ cuisine:"Italian", rating:
{$lt:8}})
db.restaurants.find({cuisine:"Italian"})
```

Partial Index vs. Sparse Index

- Partial indexes is a superset of sparse indexes and should be preferred over sparse indexes.
- Partial indexes are more expressive:
- The filter can specify conditions other than just an existence check.
- Can also specify filter expressions on fields other than the index key.

Partial index serving as spare index

- A partial index can also specify filter expressions on fields other than the index key.

```
db.contacts.createIndex(
    { name: 1 },
    { partialFilterExpression: {email: { $exists: true }}}
)
```

- In order for query optimizer to use this index, the query predicate must include a non-null match on the email field as well as a condition on the name field.

```
db.contacts.find( { name: "xyz", email: { $regex: /\.org$/ } } ) //can
db.contacts.find( { name: "xyz", email: { $exists: false } } ) // cannot
```

Partial Index with Unique Constraint

- If you specify both the partialFilterExpression and a unique constraint, the unique constraint only applies to the documents that meet the filter expression. With the following existing documents,

```
{ "_id" : 1, "username" : "david", "age" : 29 }
{ "_id" : 2, "username" : "amanda", "age" : 35 }
{ "_id" : 3, "username" : "rajiv", "age" : 57 }
db.users.createIndex({ username: 1 },
{{unique:true, partialFilterExpression:{age:{$gte:21 } } })
```

db.users.insert({ username: "**david**", age: 27 }) // rejected.

db.users.insert({ username: "david", age: 20 }) // accepted

db.users.insert({ username: "amanda" }) //accepted

db.users.insert({ username: "rajiv", age: null }) //accepted

TTL Indexes (Time-To-Live Indexes)

- Single-field indexes to automatically remove documents from a collection after a certain amount of time.
- Useful for machine generated event data, logs, and session information that only need to persist in a database for a finite amount of time.

```
db.eventlog.createIndex({"lastModifiedDate":1},
{ expireAfterSeconds: 3600 } )
```

- Expiration threshold = indexed field value + specified number of seconds.
- If the indexed field in a document is not a date or an array that holds a date value(s), the document will not expire.

Full-Text Indexes

- Indexes to text search on string content
- Text indexes can include any field whose value is a string or an array of string elements.
- A collection can have at most one text index.

```
> db.blog.createIndex({content: "text"})
> db.blog.createIndex({content: "text", keywords: "text"})
> db.blog.find({"$text": {$search:"Morning"}})
> db.blog.find({"$text": {$search:"Morning", $caseSensitive:true} })
```

Example Collection

> db.blog.find()

{ "_id" : 1, "content" : "This morning I had a cup of coffee.", "about" : "beverage", "keywords" : ["coffee"] }

{ "_id" : 2, "content" : "Who doesn't like cake?", "about" : "morning", "keywords" : ["cake", "food", "dessert"] }

{ "_id" : 3, "content" : "Who doesn't like cake?", "about" : "food", "keywords" : ["cake", "food", "dessert", "morning"] }

{ "_id" : 4, "content" : "Who doesn't like cake in the morning?", "about" : "food", "keywords" : ["cake", "food", "dessert", "morning"] }

Control Search Results with Weights

```
> db.blog.createIndex(
{content: "text", keywords: "text", about: "text"} ,
 {weights: {content: 10,keywords: 5} }
 )
```

- For a text index, the weight of an indexed field denotes the significance of the field relative to the other indexed fields in terms of the text search score.
- Formula to calculate a text search score:

https://groups.google.com/forum/#!topic/mongodb-user/99t5WXmUUAg

```
> db.blog.find( {"$text": {$search:"Morning"}}, { score: { $meta: "textScore" }})
{ "_id" : 1, "content" : "This morning I had a cup of coffee.", "about" :
"beverage", "keywords" : [ "coffee" ], "score" : 0.6666666666666666 }
{ "_id" : 3, "content" : "Who doesn't like cake?", "about" : "food", "keywords" : [
"cake", "food", "dessert", "morning" ], "score" : 1 }
{ "_id" : 2, "content" : "Who doesn't like cake?", "about" : "morning",
"keywords" : [ "cake", "food", "dessert" ], "score" : 1 }
{ "_id" : 4, "content" : "Who doesn't like cake in the morning?", "about" :
"food", "keywords" : [ "cake", "food", "dessert", "morning" ], "score" :
1.6666666666666665 }
```

Wildcard Text Indexes

- Wildcard text index creates a full text index on all string fields in a document.

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

- Useful with highly unstructured data if it is unclear which fields to include in the text index or for ad-hoc querying.

Geospatial Indexes and Queries

Reference:

[1] https://docs.mongodb.com/manual/applications/geospatial-indexes/

[2] http://tugdualgrall.blogspot.com/search?q=geospatial

Type of Surface and Location Data

- Spherical
  - To calculate geometry over an Earth-like sphere, store your location data on a spherical surface and use2dsphere index.
  - Store location data as either GeoJSON objects or legacy Coordinate Pairs

  e.g.) location : {
  type : "Point" ,
  coordinates : [10,45]
  }
- Flat: To calculate distances on a Euclidean plane, store your location data as legacy coordinate pairs and use a 2dindex.

  e.g.) location : [10,45]

GeoJSON

- http://geojson.org/
- A format for encoding, in JSON, a variety of geographic data structures
- The format consists of two attributes: type and coordinates
- GeoJSON Types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, GeometryCollection
- To validate and view a GeoJSON: http://geojson.io/

Example: GeoJSON types

```
{
  "type": "Point",
  "coordinates": [
    -2.5469,
    48.5917
  ]
}
```

```
{
  "type": "LineString",
  "coordinates": [
    [-2.551082,48.5955632],
    [-2.551229,48.594312],
    [-2.551550,48.593312],
    [-2.552400,48.592312],
    [-2.553677, 48.590898]
  ]
}
```

Example: GeoJSON types

```
{ "type": "Polygon",
   "coordinates": [
     [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ]
     ]
  }
```

```
{ "type": "MultiLineString",
   "coordinates": [
      [ [100.0, 0.0], [101.0, 1.0] ],
      [ [102.0, 2.0], [103.0, 3.0] ]
    ]
  }
```

Example: GeoJSON types

```
{ "type": "MultiPolygon",
  "coordinates": [
    [[[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0, 2.0]]],
    [[[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]],
     [[100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2]]]
    ]
  }
```

```
{ "type": "GeometryCollection",
  "geometries": [
    { "type": "Point",
      "coordinates": [100.0, 0.0]
      },
    { "type": "LineString",
      "coordinates": [ [101.0, 0.0], [102.0, 1.0] ]
      }
  ]
}
```

To store geospatial information

```
db.airports.insert(
  {
    "name" : "John F Kennedy Intl",
    "type" : "International",
    "code" : "JFK",
    "loc" : {
      "type" : "Point",
      "coordinates" : [ -73.778889, 40.639722 ]
    }
}
```

To build Geospatial Index

- `db.airports.createIndex( { "loc" : "2dsphere" } );`
- A geospatial query will run without index but it will be more efficient with one. ($near requires an index.)
- Run a query with the explain() to see the detail.

To query geospatial information

- https://docs.mongodb.com/manual/reference/operator/query-geospatial/
- Geometry Query Selectors

  `$geoWithin, $geoIntersects, $near, $nearSphere`

- Geometry Specifiers

  `$geometry, $minDistance, $maxDistance, $center, $centerSphere, $box, $polygon`

$geoWithin

- Selects documents with geospatial data that exists entirely within a specified shape.

```
{
  <location field>: {
    $geoWithin: {
      $geometry: {
        type: <"Polygon" or "MultiPolygon"> ,
        coordinates: [ <coordinates> ]
      }
}}}
```

$geoIntersects

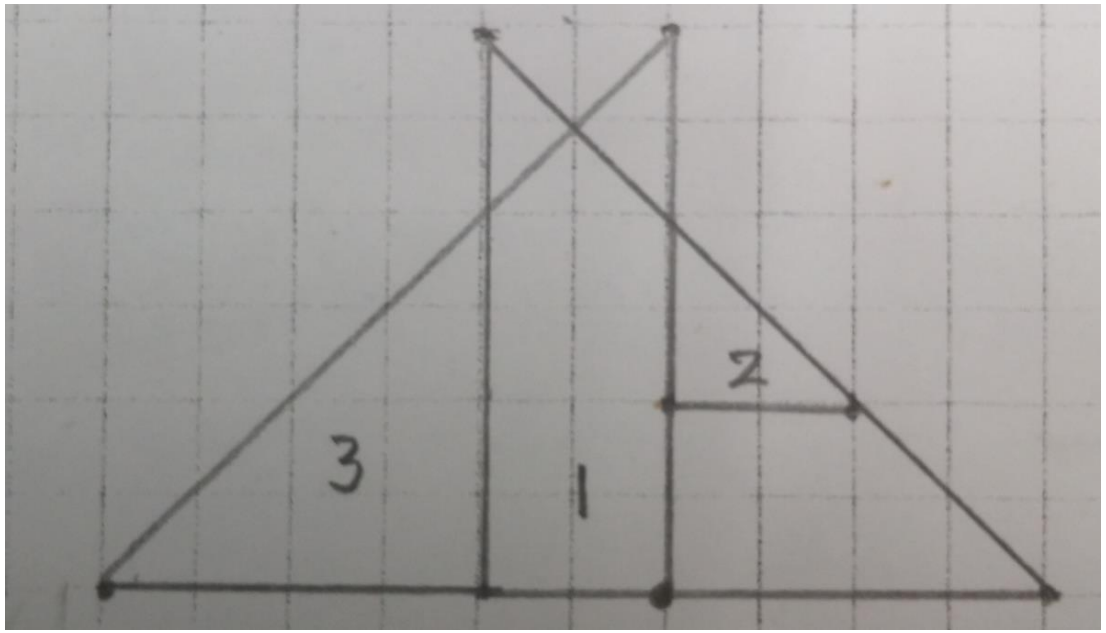
- Selects documents whose geospatial data intersects with a specified GeoJSON object. This includes cases where the data and the specified object share an edge.

```
{
  <location field>: {
    $geoIntersects: {
      $geometry: {
        type: "<GeoJSON object type>" ,
        coordinates: [ <coordinates> ]
  }}}
}
```

Example:

```
p1 = { "type" : "Polygon", "coordinates" : [[[0, 0], [3, 0], [0, 3], [0, 0]]] }
p2 = { "type" : "Polygon", "coordinates" : [[[1, 1], [2, 1], [1, 2], [1, 1]]] }
p3 = { "type" : "Polygon", "coordinates" : [[[1, 0], [-2, 0], [1, 3], [1, 0]]]
}
db.test.insert({ "loc" : p2 })
db.test.insert({ "loc" : p3 })
db.test.createIndex({ "loc" : "2dsphere" })
```

Example




```
db.test.find({ "loc" : {
    "$geoIntersects" : {
        "$geometry" : p1
    }
} }) // p2 and p3 return
db.test.find({ "loc" : {
    "$geoWithin" : {
        "$geometry" : p1
    }
} }) // p2 returns
```

$near

- Specifies a point for which a geospatial query returns the documents from nearest to farthest.
- $near requires a geospatial index

```
{
  <location field>: {
    $near: {
      $geometry: {
        type: "Point" ,
        coordinates: [ <longitude> , <latitude> ]
      },
      $maxDistance: <distance in meters>,
      $minDistance: <distance in meters>
    }
  }
}
```

Example: $near

```
db.places.find(
  {
    location:
      { $near :
        {
          $geometry: { type: "Point",  coordinates: [ -73.9667, 40.78 ] },
          $minDistance: 1000,
          $maxDistance: 5000
        }
      } }
)
```

Returns documents that are at least 1000 meters from and at most 5000 meters from the specified GeoJSON point, sorted from nearest to farthest:

How MongoDB Chooses an Index

- First it looks for a prima facie "optimal index" for the query.
- If no such index exists, it gathers all the indexes relevant to the query and pits them against each other in a race to see who finishes, or finds 101 documents, first.

hint() to force using a specific index

```
db.comments.find({ timestamp: { $gte: 2, $lte: 4 }, anonymous: false } ).
sort( { rating: -1 } ... ).hint( { anonymous: 1, rating: 1 } ).explain()
```