NoSQL Databases

MongoDB – Indexes

Indexes

- Indexes are the key for MongoDB performance
  - Without indexes, MongoDB must scan every document in a collection to select matching documents
- Indexes store some fields in easily accessible form
  - Stores values of a specific field(s) ordered by the value

- Defined per collection
- Purpose:
  - To speed up common queries
  - To optimize performance of other specific operations

Indexes

mongoDB

- B+ tree indexes
- An index is automatically created on the _id field (the primary key)
- **Users can create other indexes** to improve query performance or **to enforce** Unique values for a particular field
- Supports single field index as well as Compound index
  - Like SQL order of the fields in a compound index matters
  - If you **index a field** that holds **an array value**, **MongoDB creates** separate index entries **for every element of the array**
- Sparse property of an index ensures that the index only contain entries for documents that have the indexed field. (**so ignore records that do not have the field defined**)
- **If an index is both unique and sparse** – then the system will **reject reco**rds that **have a duplicate key value** but allow records that **do not have the indexed field defined**

Indexes- High Performance read

mongoDB

- Typically used for frequently used queries
- **Necessary when the total size of the documents exceeds** the amount of available RAM.
- Defined on the collection level
    - Can be defined on 1 or more fields
        - Composite index (SQL) -> Compound index (MongoDB)
- B-tree index
- Only 1 index can be used by the query optimizer when retrieving data
- Index covers a query - match the query conditions and return the results using only the index;
    - Use index to provide the results.

How to create Index

Syntax:

db.COLLECTION_NAME.createIndex({KEY:1})

- In the above syntax, the key is the field name that must be created. 1 is used for the ascending order, and to create a descending order index, use -1.

Example:

db.student.createIndex({student_name: 1})

How to create Index

Optional parameters that can be used in create index method:

- **Backgroun**d: By using this you can make the index in the background so that other database activities do not stop. The data type of this parameter is Boolean.
- **Sparse:** If the value is true, the index mentions documents with certain fields.
- **Name:** It is the name of the index. If you don't specify the name of the index, Mongo DB automatically generates it by concatenating the index fields and arranging it.
- **Expireafterseconds**: It mentions the total time limit to control for how much time a NoSQL database retains a document.

Index Types

mongoDB

- Default: _id
  - Exists by default
    - If applications do not specify _id, it is created.
  - Unique
- Single Field
  - User-defined indexes on a single field of a document
- Compound
  - User-defined indexes on multiple fields
- Multikey index
  - To index the content stored in arrays
  - Creates separate index entry for each array element

Index Types - Single

mongoDB

- To declare an index of this type we must use the following statement:

  db.patients.createIndex({“name”:1})

  db.clients.createIndex({“name":-1})

  - The number 1 indicates that we want the index to be sorted in ascending order.
  - If we wanted a descending order, the parameter will be a -1.

- They can be defined both on top-level documents and on **embedded document fields.**

  db.customers.createIndex({"address.province":1})

Index Types - Compound

- These indexes apply to various fields of the documents in a collection.
- To declare an index of this type we must use a command similar to this:

  db.patients.createIndex({ “name” : 1 , “age”:-1})

- The index that will be generated with the previous instruction will group the data first by the name field and then by the age field.
- That is, something like this would be generated:

  “John", 35

  “John", 18

  "Mary", 30

  "Mary", 21

Index Types - Compound

mongoDB

- Composite indexes can be used to query one or more of the fields, without needing to include all of them.
  - The composite index will sort the first field in the same way as if we created a simple index.
    - What we cannot do is search only on the following fields (e.g. age). It would be necessary to create a specific index.
  - If the composite index had three fields, we could use it when querying on the first field, on the first and second fields, or on all three fields

Index Types - Unique

- Simple and compound indices can be required to contain unique values.
  - We achieve this by adding the unique parameter when creating them.

db.patients.createIndex({“name”:1},{"unique":true})

- In case we try to insert repeated values into some field with a unique index, MongoDB will return an error.

Index Types - Sparse

mongoDB

- The indexes seen so far include **all** documents, including those that **do not have the indexed field**. For documents that do not have the field indexed it stores **a** null value in the index.

- To create indexes that only include documents whose indexed field exists, we will use the **sparse option**.

  db.patients.createIndex({“name”:1},{"sparse":true})

- In this case what we do is create an index that does not have to contain all the documents in the collection.

Index Types - Partial

- Partial indexes only index documents in a collection that satisfy a condition (expression).
- They require less storage space and less cost to create and maintain.
- •To create indexes that only include documents that satisfy the filtering condition, we will use the **partialFilterExpression option.**

db.patients.createIndex({“name”:1},{partialFilterExpression:{“age”:{$gt:18}}})
db.patients.createIndex({“name”:1},{partialFilterExpression:{“name”:{$exists:true}}})

- This option exists as of MongoDB version 3.2: it is recommended to choose this option (superset) over the spread option (more functionality).

Index Types - Partial

- The partialFilterExpression option accepts a document that specifies a condition using:
  - Equality Expressions $eq

    ({“name”:”John”} or {“name”:{$eq:”John”}}
  - Expression $exists: true-> Only documents where a certain field exists

    { partialFilterExpression: { age: { $exists: true }} }
  - Expressions: $gt, $gte, $lt, $lte

    {“age”:{$gte:18}}
  - Expression $type-> Only documents where a field is of a specific type

    {“name”:{$type:”string”}}
  - $and operator only at the top level of the document

Index Types - Others

mongoDB

- Geospatial Index
  - 2d indexes = use planar geometry when returning results
    - For data representing points on a two-dimensional plane
  - 2sphere indexes = spherical (Earth-like) geometry
    - For data representing longitude, latitude
- Text Indexes
  - Searching for string content in a collection
- Hash Indexes
  - Fast indexes the hash of the value of a field
    - Only equality matches

Index Types – 2dsphere

db.collection.createIndex({ <location field> : "2dsphere" })

```
db.places.insert(
{
loc : { type: "Point", coordinates: [ -73.97, 40.77 ] },
name: "Central Park",
category : "Parks"
})
```

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

Index Types - Text

- MongoDB provides text indexes to support text search queries on string content.
- Text indexes can include any field whose value is a string or an array of string elements.
- To perform text search queries, we must have a text index on collection.
  - **A collection can only have one text search in**dex, but that index can cover multiple fields.

```
db.stores.createIndex( { name: "text", description: "text" } )
```

- **Wildcard Text Indexes**
  - When creating a text index on multiple fields, you can also use the wildcard specifier ($**).
  - With a wildcard text index, MongoDB indexes every field that contains string data for each document in the collection

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

This index allows for text search on all fields with string content.

Index Types - Text

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

```
db.blog.find( { $text: { $search: "coffee" } } )
```

```
[
  {
    _id: 1,
    content: 'This morning I had a cup of coffee.',
    about: 'beverage',
    keywords: [ 'coffee' ]
  },
  {
    _id: 3,
    content: 'My favorite flavors are strawberry and coffee',
    about: 'ice cream',
    keywords: [ 'food', 'dessert' ]
  }
]
```

Index Types - Text

mongoDB

```
db.<collection>.createIndex(
   { <field>: "text" },
   { default_language: <language> }
)
```

```
db.quotes.createIndex(  { quote: "text" }, { default_language: "spanish" }
)
```

Index Types - Text

mongoDB

- Wildcard text indexes, as with all text indexes, can be part of a compound indexes

  db.collection.createIndex({ a: 1, "$": "text" })**

  - to perform a $text search with this index, the query predicate must include an equality match conditions a

- **Restrictions**
  - One Text Index Per Collection
  - A collection can have at most one text index.
  - cannot use hint() if the query includes a $text query expression
  - Sort operations cannot obtain sort order from a text index, even from a compound text index; i.e. sort operations cannot use the ordering in the text index.

Index Types - Hash

- Hashed indexes support hashed sharding.
- A hashed index acts as a shard key to distribute data across shards based on hashes of field values.
- **Create a Single-Field Hashed Index**

```
db.<collection>.createIndex(  {
      <field>: "hashed”  })
```

```
db.orders.createIndex( { _id: "hashed" } )
```

- After you create the index, you can shard the orders collection:

```
sh.shardCollection(
   "<database>.orders",
   { _id: "hashed" })
```

Index Types - Hash

- **Create a Compound Hashed Index**

```
db.<collection>.createIndex( {
        <field1>: "hashed",  <field2>: "<sort order>",
        <field3>: "<sort order>",  ...  } )
```

```
db.customers.createIndex(
   {
      "name" : 1
      "address" : "hashed",
      "birthday" : -1
   }
)
```

```
sh.shardCollection(
   "<database>.customers",
   {
      "name" : 1
      "address" : "hashed",
      "birthday" : -1
   }
)
```

Advanced Indexing

mongoDB

Array Fields Indexing

- For making an index based on array one needs to create different index entries for separate fields.

  When the index is created, we can search on the array field of the collection of indexes

- We must do it carefully and taking into account some limitations:

  The first limitation is that only one of the index fields can be an array.

  db.users.createIndex({"categories":1,"tags":1})

  If we type this command

  db.users.insert({"categories":["game","book","movie"],"tags":["horror","scifi","history"]

  Error: ***can not index parallel arrays[categories][tags]***

An entry would have to be created in the index for the Cartesian product of elements. If the arrays were large, it would be an unmanageable operation.

Advanced Indexing

Array Fields Indexing

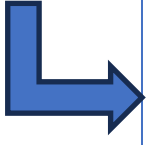
- The other limitation is that the indexed elements of the array do not take into account the order. Therefore, if we perform positional queries on the array, the index will not be used.

```
db.students.insertMany( [
  {
    "name": "Andre Robinson",
    "test_scores": [ 88, 97 ]
  },
  {
    "name": "Wei Zhang",
    "test_scores": [ 62, 73 ]
  },
  {
    "name": "Jacob Meyer",
    "test_scores": [ 92, 89 ]
  }
] )
```

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

[1, 4, 6, 7, 8, 10]

```
db.inventory.find( {  "stock.quantity": { $lt: 5 }} )
```



```
[
  {
    _id:
ObjectId("63449793b1fac2ee2e957ef3"),
    item: 'vest',
    stock: [ { size: 'small', quantity: 6 }, { size:
'large', quantity: 10 } ]
  }, …….
  {
```

MongoDB stores this index as a multikey index.

Advanced Indexing

Index on an Embedded Field in an Array

These indexes improve performance for queries on specific embedded fields that appear in arrays.

```
db.inventory.insertMany( [
  {
    "item": "t-shirt",
    "stock": [
      {
        "size": "small",
        "quantity": 6
      },
      {
        "size": "large",
        "quantity": 10
      },
    ]
  }……..
```

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

[1, 4, 6, 7, 8, 10]

```
db.inventory.find( {  "stock.quantity": { $lt: 5 }} )
```

```
[
  {
    _id:
ObjectId("63449793b1fac2ee2e957ef3"),
    item: 'vest',
    stock: [ { size: 'small', quantity: 6 }, { size:
'large', quantity: 10 } ]
  }, …….
  {
```

MongoDB stores this index as a multikey index.

Indexes – hint

mongoDB

- This method allows you to indicate in a query which index to use.
  - Or by specifying the index

    Example: db.students.find().hint({ age: 1 })
  - Or indicating the name of the index:

    Example: db.students.find().hint({ “age_1”})

- To force a collection scan to be performed in a query (not using any index): **{ $natural : 1 }**

  Example (forward collectionscan): db.students.find().hint({ $natural : 1 })

  Example (reverse collection scan): db.students.find().hint({ $natural: -1 })

- To force the use of an index we use the hint function:

  – db.collection.find({ att1: value}).hint({ att1: 1 })

  - **The index must exist** in the attribute, **otherwise** the system **will return an error**

Advanced Indexing

mongoDB

- Index bounds define the range of index values that MongoDB searches when using an index to fulfill a query. When you specify multiple query predicates on an indexed field, MongoDB attempts to combine the bounds for those predicates to produce an index scan with smaller bounds. Smaller index bounds result in faster queries and reduced resource use.
- In general, if an index is used for a query, the resulting documents are returned in index order
  - However, can specify a different sorting order:

    db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).sort({"username" : 1})

  - MongoDB will use the index to match the criteria
    - BUT: the index doesn't return the usernames in sorted order
    - MongoDB needs to sort the results in memory before returning them
  - Usually less efficient : If you have more than 32 MB of results, MongoDB will show an error

Indexes

mongoDB

Inefficient Queries

- Some queries can use indexes more efficiently than others
- Inefficient
  - Negation: $ne, $not, $nin
  - Try to find another clause using an index to add to the query, so that nonindexed matching is performed on a smaller set
- $or: two separate queries are performed and then merged
  - Merging: needs to look through results of both queries and remove duplicates
  - Less efficient than a single query, so use $in instead

When (and when not) to use indexing

Indexes work well for:

- Large collections
- Large documents
- Selective queries

Indexing is inefficient (Collection scans work better):

- Small collections
- Small documents
- Nonselective queries

When to create anIndex

mongoDB

Scenario	Index Type
A human resources department often needs to look up employees by employee ID. You can create an index on the employee ID field to improve query performance.	Single Field Index
A salesperson often needs to look up client information by location. Location is stored in an embedded object with fields like state, city, and zipcode. You can create an index on the entire location object to improve performance for queries on any field in that object.	Single Field Index on an object
A grocery store manager often needs to look up inventory items by name and quantity to determine which items are low stock. You can create a single index on both the item and quantity fields to improve query performance.	Compound Index

Indexes -Recomendations

Mass insertion recommendation in a collection:

1. Delete indexes with db.collection.dropIndexes()
2. Inserting documents
3. Re-Creating indexes db.collection.createIndex()

The creation time of the indexes is much shorter than the time needed to update the same index already created in each insertion of each document.

Indexes -Recomendations

- Recreating the indexes of a collection using **the reIndex() method**

  Example: db.clientes.reIndex()

  - This method deletes all indexes from a collection and recreates them.
  - This is an expensive operation for large collections and/or with a large number of indexes.
  - Appropriate when the size of the collection has changed a lot or when the disk used by the indices is disproportionately high.

Execution Plans

mongoDB

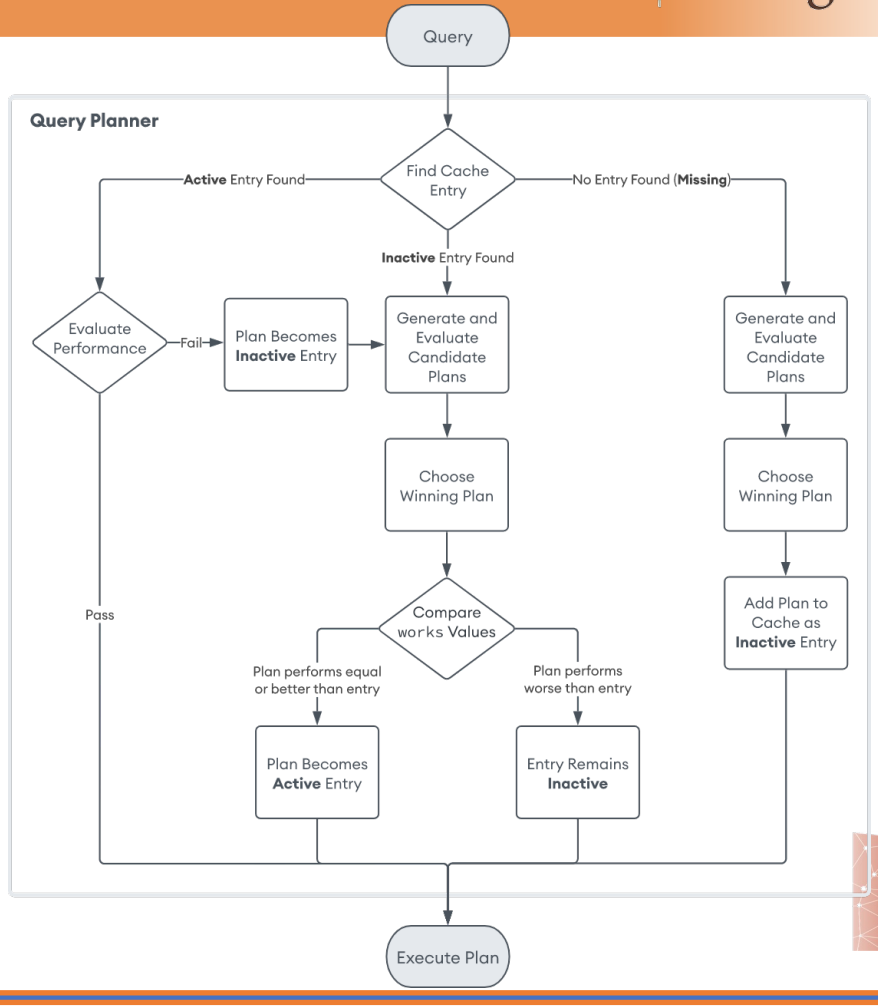
- To know if na indexing was adequate , we can use the explain command to check for it.

```
Db.users.find({
    tags: “programmer”
 })
 .explain()
```

- It will return a JSON document with useful information:
  - number of documents processed to return the result
  - whether the query used the indexes or scanned the collection
  - if you have had to sort the results in memory
  - information about the server, etc.

Execution Plans

- For a query, the MongoDB query optimizer chooses and caches the most efficient query plan given the available indexes.
- The evaluation of the most efficient query plan is based on the number of “work units” (works) performed by the query execution plan when the query planner evaluates candidate plans.
- The associated plan cache entry is used for subsequent queries with the same query shape.



Execution Plans

mongoDB

- To view the query plan information for a given query:
- db.collection_name.explain(<verbosity_parameter>) or the cursor.explain()
  - The **verbosity_parameter** is an optional string value that specifies the verbosity mode for the explain command.
    - This value determines the amount of the returning information and supports the following 3 modes
      - queryPlanner
      - executionStats, and
      - allPlansExecution
  - The default mode for the explain() method is the queryPlanner.

Execution Plans

mongoDB

- **queryPlanner** Mode:
  - the specified query in the **find()** method is put by the query optimizer to find the most efficient plan. The effective plan is then passed to this mode and the information is returned for the evaluated query
- **executionStats** Mode:
  - the query optimizer selects and executes the execution plan to complete and return the statistics describing the winning plan execution
- **allPlansExecution** Mode:
  - the query optimizer selects and executes the execution plan for further processing. In this mode, Mongo database return the statistics describing the winning plan execution and the other candidate plans captured during the plan choice

Execution Plans – Example 1


```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'test.filmes',
    indexFilterSet: false,
    parsedQuery: {
      available: {
        '$eq': 'true'
      }
    },
    queryHash: 'F2D50C97',
    planCacheKey: 'F2D50C97',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'COLLSCAN',
      filter: {
        available: {
          '$eq': 'true'
        }
      },
      direction: 'forward'
    },  …..
```

Search Query

Execution plan operation type

```
…
serverInfo: {
    host: 'ac-vmf3jop-shard-00-
02.lajct2r.mongodb.net',
    port: 27017,
    version: '6.0.11',
    gitVersion:
'f797f841eaf1759c770271ae00c88b92b2766eed'
  },
….
```

Server Information

```
db.filmes.find( { "available" : "true" } ).explain()
```

This command will execute in the default verbosity mode (i.e. “queryPlanner”) to return the query planning information.

Execution Plans – Example 1

mongoDB

- The queryStage attribute provides information about the “execution plan” or “winning plan” operation type.
  - This can be one of the following:
    - **The COLLSCAN** value indicates a full collection scan i.e. To fetch the result, the specified query searched each document in the Mongo collection
    - The IXSCAN value indicates an index search
    - The FETCH value is simply used for retrieving the documents of a collection
    - The SHARD_MERGE value indicates the merged data from different shards
- **The direction attribute** shows whether the query was performed in a forward or reverse order
- **The serverInfo secti**on displays the server information on which the query was executed and the version of the Mongo database.

Execution Plans – Example 2

db.paises.find({ "nome" : "Estados unidos" }).explain("executionStats")

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'test.paises',
    indexFilterSet: false,
    parsedQuery: {
      nome: {
        '$eq': 'Estados unidos'
      }
    },

.....
```

```
executionStats: {
    executionSuccess: true,
    nReturned: 1,
    executionTimeMillis: 0,
    totalKeysExamined: 0,
    totalDocsExamined: 13,
    executionStages: {
      stage: 'COLLSCAN',
      filter: {
        nome: {
          '$eq': 'Estados unidos'
        }
      },…
```

Execution Plans – with index


```
{
  explainVersion: '1',
……
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: {
          'duração': 1
        },
        .......
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 4,
    executionTimeMillis: 1,
    totalKeysExamined: 4,
    totalDocsExamined: 4,
    executionStages: {
……..}
```

```
db.filmes.createIndex ( { duração: 1 } )
```

```
db.filmes.find(
   { duração: { $gte: 2.0, $lte: 4.0 } }
).explain("executionStats"
```

Performance of Indexes

mongoDB

- To manually compare the performance of a query using more than one index, you can use the hint() method in conjunction with the explain() method.

```
db.filmes.find( {
   duração: {
      $gte: 1.0, $lte: 3.0
   },
   realizador: "Shane Black"
} )
```

db.filmes.createIndex({ duração: 1, realizador: 1 })

db.filmes.createIndex({ realizador: 1, duração: 1 })

Performance of Indexes

mongoDB

```
db.filmes.find(
   { duração: { $gte: 1.0, $lte: 3.0 }, realizador: "Shane Black" } ).hint({ duração: 1, realizador:1}
).explain("executionStats")
```

```
executionStats: {
   executionSuccess: true,
   nReturned: 2,
   executionTimeMillis: 0,
   totalKeysExamined: 8,
   totalDocsExamined: 2,
   executionStages: { ….
```

Performance of Indexes

mongoDB

```
db.filmes.find(
   { duração: { $gte: 1.0, $lte: 3.0 }, realizador: "Shane Black" } ).hint({ realizador: 1, duração: 1}
).explain("executionStats")
```

```
executionStats: {
   executionSuccess: true,
   nReturned: 2,
   executionTimeMillis: 0,
   totalKeysExamined: 2,
```

Conclusion: The second compound index, { realizador: 1, duração: 1 }, is therefore the more efficient index for supporting the example query, as the MongoDB server only needs to scan 2 index keys to find all matching documents using this index, compared to 8 when when using the compound index { duração:1, realizador: 1 }.

Remarks

- When designing the database, we have to try to identify the optimal number of indexes.
  - Excessive - reduces the performance of writings;
  - Few – reduces the throughput of readings
- **Things to keep in mind:**
  - Read/Write ratio (BD for analysis ..)
  - Attributes that are used in searches,
  - DB with **heavy load read**, only has many indexes if there are no clear patterns in the queries;
  - DB with **heavy write load**, (e.g. sensors): minimize the number of indexes (identifiers), balance
    - If query latency is still important, you can consider creating a second aggregate database for reads (DW).