

15 Indexación: Estrategias

The best indexes for your application must take a number of factors into account, including the kinds of queries you expect, the ratio of reads to writes, and the amount of free memory on your system.

When developing your indexing strategy you should have a deep understanding of your application's queries. Before you build indexes, map out the types of queries you will run so that you can build indexes that reference those fields. Indexes come with a performance cost, but are more than worth the cost for frequent queries on large data sets. Consider the relative frequency of each query in the application and whether the query justifies an index.

The best overall strategy for designing indexes is to profile a variety of index configurations with data sets similar to the ones you'll be running in production to see which configurations perform best. Inspect the current indexes created for your collections to ensure they are supporting your current and planned queries. If an index is no longer used, drop the index.

Generally, MongoDB only uses one index to fulfill most queries. However, each clause of an \$or query may use a different index, and in addition, MongoDB can use an intersection of multiple indexes.

Create Indexes to Support Your Queries

Create a Single-Key Index if All Queries Use the Same, Single Key

If you only ever query on a single key in a given collection, then you need to create just one single-key index for that collection.

Práctica

> use maratón

```
let nombres = ['Laura', 'Juan', 'Fernando', 'María'];
let apellidos = ['Fernández', 'González', 'Pérez', 'López'];
let letras = ['A', 'B', 'P', 'C', 'X', 'D'];
let participantes = []
for (i=0; i < 100000; i++) {
  participantes.push({
    _id: i,
    nombre: nombres[Math.floor(Math.random()*nombres.length)],
    apellido1: apellidos[Math.floor(Math.random()*apellidos.length)],
    apellido2: apellidos[Math.floor(Math.random()*apellidos.length)],
    edad: Math.floor(Math.random()*100),
    dni: Math.floor(Math.random()* 100000000 ) +
letras[Math.floor(Math.random()*letras.length)]
  })
}

db.participantes.insert(participantes);
```

Consulta por DNI sin índice:

The screenshot shows the MongoDB Compass interface. On the left, the 'Simple Query' panel is configured with the following settings:

- Filter conditions:** Attribute: `dni`, Operator: `$eq`, Value to filter: `"99828905A"`.
- Project Fields:** (Empty)
- SortKeys:** (Empty)
- Limit:** 0 (No limit)
- Count:** ☐

At the bottom of the panel are 'Close' and 'Execute' buttons. The main interface shows the 'Query Input' with the following JavaScript code:

```
1 db.getSiblingDB("maraton").getCollection("participantes").find(
2   {
3     "dni": { $eq: "99828905A" }
4   },
5   {}
6 )
```

The 'Query Output' tab is selected, showing the 'Explain' view. A red arrow indicates the operation is **COLLSCAN**. Below this, a table summarizes the execution statistics:

Seq	Step	ms	Examined	Return	Comment
1	COLLSCAN	16	100000	1	Documents were scanned looking for these criteria: {"dni": {"\$eq": "99828905A"}}. Consider creating an index on these attributes.

Below the table, a 'Statistic' section shows the following values:

Statistic	Value
Total Docs Returned	1
Total Keys Examined	0
Total Docs Examined	100000

The status bar at the bottom indicates 'dbKoda: v1.1.0' and 'Execution Time: 207ms'.

The 'Query Output' tab is selected, showing the 'Explain' view. The output is a JSON document detailing the query plan and execution statistics:

```
▼ queryPlanner: { 6 keys
  plannerVersion: 1
  namespace: "maraton.participantes"
  indexFilterSet: false
  ▼ parsedQuery: { 1 key
    ▶ dni: { 1 key
  }
  ▼ winningPlan: { 3 keys
    stage: "COLLSCAN"
    ▶ filter: { 1 key
      direction: "forward"
    }
    rejectedPlans: [] 0 items
  }
  ▼ executionStats: { 6 keys
    executionSuccess: true
    nReturned: 1
    executionTimeMillis: 81
    totalKeysExamined: 0
    totalDocsExamined: 100000
    ▼ executionStages: { 13 keys
      stage: "COLLSCAN"
      ▶ filter: { 1 key
    }
  }
}
```

The 'Explain' tab is also visible at the bottom.

Ahora creamos un índice

```
> db.participantes.createIndex({dni: 1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 2,
  "numIndexesAfter" : 3,
  "ok" : 1
}
```

Query Output

Raw Table × Explain ×

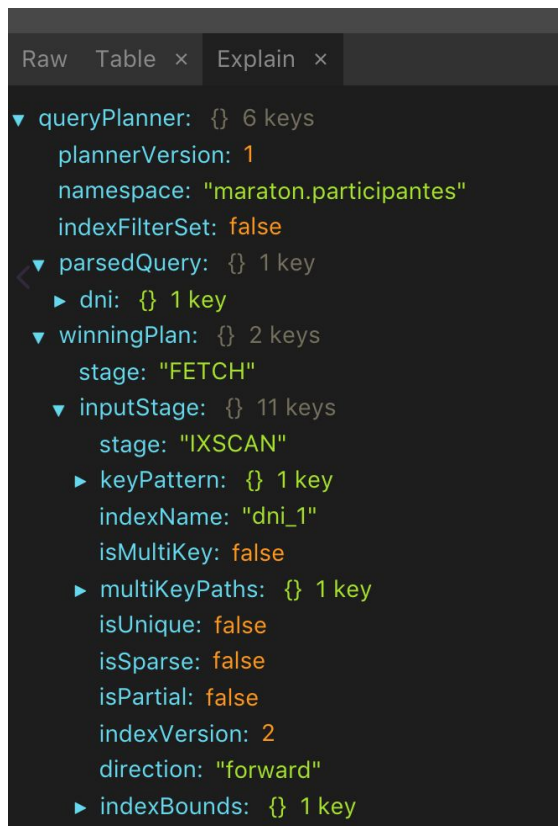
Explain

Raw Json Index Advisor

IXSCAN → FETCH

Seq	Step	ms	Examined	Return	Comment
1	IXSCAN	3	1	1	Index dni_1 was used to find matching values for dni
2	FETCH	3	1	1	Retrieved documents from index fetch

Statistic	Value
Total Docs Returned	1
Total Keys Examined	1
Total Docs Examined	1



Create Compound Indexes to Support Several Different Queries

If you sometimes query on only one key and at other times query on that key combined with a second key, then creating a compound index is more efficient than creating a single-key index. MongoDB will use the compound index for both queries.

Práctica

Si busco por apellido1 y edad

```
> db.participantes.createIndex({edad: 1, apellido1: 1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

Y ejecutamos una consulta:

Simple Query

participantes

Filter Operator

- ☒ Default
- ☐ Use \$or
- ☐ Use \$and

Filter conditions

Attribute	Operator	Value to filter
edad	\$gte	18
apellido1	\$eq	"López"

Query Input

```

1 db.getSiblingDB("maraton").getCollection("participantes").find(
2   {
3     "edad": { $gte: 18 },
4     "apellido1": { $eq: "López" }
5   },
6   {}
7 )

```

Query Output

Raw Table Explain

Explain

IXSCAN FETCH

Seq	Step	ms	Examined	Return	Comment
1	IXSCAN	4	20654	20491	Index edad_1_apellido1_1 was used to find matching values for edad,apellido1
2	FETCH	13	20491	20491	Retrieved documents from index fetch

Statistic	Value
Total Docs Returned	20491
Total Keys Examined	20654
Total Docs Examined	20491

Namespace: maraton.participantes

Query:

Execution Time: 95ms

```

winningPlan: { 2 keys
  stage: "FETCH"
  inputStage: { 11 keys
    stage: "IXSCAN"
    keyPattern: { 2 keys
      edad: 1
      apellido1: 1
      indexName: "edad_1_apellido1_1"
      isMultiKey: false
      multiKeyPaths: { 2 keys

```

Con tres campos:

```

> db.participantes.createIndex({edad: 1, apellido1: 1, apellido2: 1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 2,
  "numIndexesAfter" : 3,
  "ok" : 1
}

```

Y creamos una consulta:

The screenshot displays the MongoDB Simple Query interface. On the left, the 'Filter conditions' panel shows a query built with the following conditions:

- Attribute: edad, Operator: \$gte, Value: 18
- Attribute: apellido1, Operator: \$eq, Value: "López"
- Attribute: apellido2, Operator: \$eq, Value: "Pérez"

The 'Project Fields' section is empty. The 'Execute' button is visible at the bottom right of the filter panel.

The 'Query Input' section shows the following JSON query:

```
db.getSiblingDB("maraton").getCollection("participantes").find({
  "edad": { $gte: 18 },
  "apellido1": { $eq: "López" },
  "apellido2": { $eq: "Pérez" }
})
```

The 'Query Output' section shows the 'Explain' tab selected. The execution plan is visualized as a sequence of steps: IXSCAN (green arrow) followed by FETCH (red arrow).

The 'Explain' table shows the following data:

Seq	Step	ms	Examinated	Return	Comment
1	IXSCAN	7	5292	5129	Index edad_1_apellido1_1_apellido2_1 was used to find matching values for edad,apellido1,apellido2
2	FETCH	13	5129	5129	Retrieved documents from index fetch

The 'Statistics' table shows the following data:

Statistic	Value
Total Docs Returned	5129
Total Keys Examined	5292
Total Docs Examined	5129

The bottom status bar indicates 'Execution Time: 111ms'.

Plan ganador:

```
▼ winningPlan: {} 2 keys
  stage: "FETCH"
▼ inputStage: {} 11 keys
  stage: "IXSCAN"
  ► keyPattern: {} 3 keys
    indexName: "edad_1_apellido1_1_apellido2_1"
    isMultiKey: false
  ► multiKeyPaths: {} 3 keys
    isUnique: false
    isSparse: false
    isPartial: false
    indexVersion: 2
    direction: "forward"
  ► indexBounds: {} 3 keys
```

Plan desechado:


```
▼ rejectedPlans: [] 1 item
  ▼ 0: {} 3 keys
    stage: "FETCH"
    ▼ filter: {} 1 key
      ► apellido2: {} 1 key
    ▼ inputStage: {} 11 keys
      stage: "IXSCAN"
      ► keyPattern: {} 2 keys
        indexName: "edad_1_apellido1_1"
        isMultiKey: false
      ► multiKeyPaths: {} 2 keys
        isUnique: false
        isSparse: false
        isPartial: false
        indexVersion: 2
        direction: "forward"
      ► indexBounds: {} 2 keys
```

¿Qué ocurre si interviene un tercer campo diferente al utilizado en el último índice?

The screenshot shows the MongoDB Compass interface. On the left, the 'Simple Query' panel has filter conditions: 'edad' \$gte 18, 'apellido1' \$eq 'López', and 'nombre' \$eq 'Fernández'. The 'Project Fields' section is empty. The 'Query Input' panel shows the following query:

```
db.getSiblingDB('maraton').getCollection('participantes').find(
  {
    'edad': { '$gte': 18 },
    'apellido1': { '$eq': 'López' },
    'nombre': { '$eq': 'Fernández' }
  })
```

The 'Query Output' panel shows the 'Explain' tab. It displays a visual plan with 'IXSCAN' and 'FETCH' stages. Below this is a table with the following data:

Seq	Step	ms	Examined	Return	Comment
1	IXSCAN	3	20654	20491	Index edad_1_apellido1_1_apellido2_1 was used to find matching values for edad,apellido1,apellido2
2	FETCH	6	20491	0	Documents from previous step were scanned looking for these criteria: {'nombre':{'\$eq':'Fernández'}}.

At the bottom, a 'Statistics' table shows:

Statistic	Value
Total Docs Returned	0
Total Keys Examined	20654

The status bar at the bottom indicates 'dbKoda: v1.1.0' and 'Execution Time: 111ms'.

En principio también utiliza el de 3 campos. En allPlansExecution se pueden ver los datos:

The screenshot shows the 'allPlansExecution' output in the MongoDB Compass interface. It displays a tree structure with the following data:

- allPlansExecution: [] 2 items
 - 0: {} 5 keys
 - nReturned: 0
 - executionTimeMillisEstimate: 6
 - totalKeysExamined: 20654
 - totalDocsExamined: 20491
 - executionStages: {} 14 keys
 - 1: {} 5 keys
 - nReturned: 0
 - executionTimeMillisEstimate: 7
 - totalKeysExamined: 20654
 - totalDocsExamined: 20491
 - executionStages: {} 14 keys

Index Use and Collation

To use an index for string comparisons, an operation must also specify the same collation. That is, an index with a collation cannot support an operation that performs string comparisons on the indexed fields if the operation specifies a different collation.

Creemos un índice con colación sobre el campo nombre:

```
> db.participantes.createIndex({nombre: 1},{collation:{locale: "es", strength: 1}})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 3,
  "numIndexesAfter" : 4,
  "ok" : 1
}
```

La creación del índice solo es a los efectos de que este pueda ser utilizado por la consulta con colación.

Si pasamos la consulta sin colación, ni usa la colación:

```
> db.participantes.find({nombre: "juan"})
Vacío
```

Ni usa el índice:

```
> db.participantes.find({nombre: "juan"}).explain()
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "maraton.participantes",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "nombre" : {
        "$eq" : "juan"
      }
    },
    "queryHash" : "F53262C1",
    "planCacheKey" : "C1628CCF",
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "nombre" : {
          "$eq" : "juan"
        }
      }
    }
  }
```

```

    },
    "direction" : "forward"
  },
  ...

```

En cambio con la colación en este ejemplo encuentra minúsculas:

```

> db.participantes.find({nombre: "juan"}).collation({locale: "es", strength: 1})
{ "_id" : 0, "nombre" : "Juan", "apellido1" : "González", "apellido2" : "López", "edad" : 7,
  "dni" : "13593548A" }
{ "_id" : 1, "nombre" : "Juan", "apellido1" : "López", "apellido2" : "Pérez", "edad" : 4,
  "dni" : "21030172C" }
{ "_id" : 5, "nombre" : "Juan", "apellido1" : "González", "apellido2" : "López", "edad" : 16,
  "dni" : "72284457P" }

```

Y utiliza el índice:

```

> db.participantes.find({nombre: "juan"}).collation({locale: "es", strength:
1}).explain()
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "maraton.participantes",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "nombre" : {
        "$eq" : "juan"
      }
    },
    "collation" : {
      "locale" : "es",
      "caseLevel" : false,
      "caseFirst" : "off",
      "strength" : 1,
      "numericOrdering" : false,
      "alternate" : "non-ignorable",
      "maxVariable" : "punct",
      "normalization" : false,
      "backwards" : false,
      "version" : "57.1"
    },
    "queryHash" : "A98837A4",
    "planCacheKey" : "8BBECEBD",
    "winningPlan" : {
      "stage" : "FETCH",

```

```
"inputStage" : {  
  "stage" : "IXSCAN",
```

Con una colación diferente ya no usa el índice:

```
> db.participantes.find({nombre: "juan"}).collation({locale: "en", strength:  
1}).explain()
```

```
{  
  "queryPlanner" : {  
    "plannerVersion" : 1,  
    "namespace" : "maraton.participantes",  
    "indexFilterSet" : false,  
    "parsedQuery" : {  
      "nombre" : {  
        "$eq" : "juan"  
      }  
    },  
    "collation" : {  
      "locale" : "en",  
      "caseLevel" : false,  
      "caseFirst" : "off",  
      "strength" : 1,  
      "numericOrdering" : false,  
      "alternate" : "non-ignorable",  
      "maxVariable" : "punct",  
      "normalization" : false,  
      "backwards" : false,  
      "version" : "57.1"  
    },  
    "queryHash" : "36F91AFC",  
    "planCacheKey" : "F875A2AD",  
    "winningPlan" : {  
      "stage" : "COLLSCAN",  
      "filter" : {  
        "nombre" : {  
          "$eq" : "juan"  
        }  
      },  
      "direction" : "forward"  
    },  
    "rejectedPlans" : []  
  },  
  ...
```

Si la colección tiene una colación por defecto se usará el índice siempre que la colación de la consulta sea la misma de la colación.

```

> db.createCollection("fuuu", {collation: {locale: "es", strength: 1}})
{ "ok" : 1 }
> db.fuuu.createIndex({nombre: 1},{collation:{locale: "es", strength: 1}})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
> db.fuuu.insert([{nombre: "Tomás"},{nombre: "Carlos"}])

> db.fuuu.find({nombre: "tomas"})
{ "_id" : ObjectId("5e386e28a1491be19abf6c5f"), "nombre" : "Tomás" }
> db.fuuu.find({nombre: "tomas"}).explain()
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "maraton.fuuu",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "nombre" : {
        "$eq" : "tomas"
      }
    },
    "collation" : {
      "locale" : "es",
      "caseLevel" : false,
      "caseFirst" : "off",
      "strength" : 1,
      "numericOrdering" : false,
      "alternate" : "non-ignorable",
      "maxVariable" : "punct",
      "normalization" : false,
      "backwards" : false,
      "version" : "57.1"
    },
    "queryHash" : "A98837A4",
    "planCacheKey" : "8BBECEBD",
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",

```

Use Indexes to Sort Query Results

In MongoDB, sort operations can obtain the sort order by retrieving documents based on the ordering in an index. If the query planner cannot obtain the sort order from an index, it will sort the results in memory.

Sort operations that use an index often have better performance than those that do not use an index. In addition, sort operations that do not use an index will abort when they use 32 megabytes of memory.

NOTE. As a result of changes to sorting behavior on array fields in MongoDB 3.6, when sorting on an array indexed with a multikey index the query plan includes a blocking SORT stage. The new sorting behavior may negatively impact performance.

In a blocking SORT, all input must be consumed by the sort step before it can produce output. In a non-blocking, or indexed sort, the sort step scans the index to produce results in the requested order.

Sort with a Single Field Index

If an ascending or a descending index is on a single field, the sort operation on the field can be in either direction.

Práctica

Reinicializamos índices para observar mejor:

```
> db.participantes.dropIndexes()
{
  "nIndexesWas" : 4,
  "msg" : "non-_id indexes dropped for collection",
  "ok" : 1
}
```

Sobre un índice simple, la ordenación se puede realizar en cualquier sentido:

```
> db.participantes.createIndex({dni: 1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

Comprobamos:

The screenshot shows the MongoDB Compass interface. On the left, the 'Simple Query' panel is visible with the following configuration:

- Attribute: dni
- Operator: \$regex
- Value to filter: "P"
- Project Fields: (empty)
- SortKeys: dni (Ascending?)
- Limit (0=No limit): 0

On the right, the 'Query Input' panel shows the following query:

```
1 db.getSiblingDB("maraton").getCollection("participantes").find(
2 {
3   "dni": { $regex: "P" }
4 },
5 {})
```

The 'Query Output' panel shows the 'Explain' tab selected. It displays the execution plan with two steps: IXSCAN and FETCH.

Seq	Step	ms	Examined	Return	Comment
1	IXSCAN	7	100000	16621	Index dni_1 was used to find matching values for dni
2	FETCH	11	16621	16621	Retrieved documents from index fetch

Below the execution plan, a 'Statistic' table is shown:

Statistic	Value
Total Docs Returned	16621
Total Keys Examined	100000
Total Docs Examined	16621

At the bottom, the 'Namespace' is 'maraton.participantes' and the 'Query' is the same as in the input panel. The 'Execution Time' is 145ms.

Sort on Multiple Fields

You can specify a sort on all the keys of the index or on a subset; however, the sort keys must be listed in the same order as they appear in the index. For example, an index key pattern { a: 1, b: 1 } can support a sort on { a: 1, b: 1 } but not on { b: 1, a: 1 }.

For a query to use a compound index for a sort, the specified sort direction for all keys in the cursor.sort() document must match the index key pattern or match the inverse of the index key pattern. For example, an index key pattern { a: 1, b: -1 } can support a sort on { a: 1, b: -1 } and { a: -1, b: 1 } but not on { a: -1, b: -1 } or { a: 1, b: 1 }.

Práctica


```
> db.participantes.createIndex({apellido1: 1, edad: -1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

Comprobamos que la siguientes consultas utilizan el índice para ordenar y no tienen que usar la ordenación en memoria:

En el orden del índice

The screenshot shows the MongoDB Query Editor interface. On the left, the 'Simple Query' panel has the following configuration:

- Value to filter:**
 - Field: `edad`, Operator: `$gte`, Value: `50`
 - Field: `apellido1`, Operator: `$eq`, Value: `"Pérez"`
- Project Fields:** (Empty)
- SortKeys:**
 - Field: `apellido1`, Order: `1` (Ascending)
 - Field: `edad`, Order: `-1` (Descending)

The 'Query Input' tab shows the following query:

```
1 db.getSiblingDB("maraton").getCollection("participantes").find(
2   {
3     "edad": { $gte: 50 },
4     "apellido1": { $eq: "Pérez" }
5   },
6   {
7     "sort": {
8       "apellido1": 1,
9       "edad": -1
10    }
11  })
```

The 'Query Output' tab shows the 'Explain' view. The execution plan is as follows:

Seq	Step	ms	Examined	Return	Comment
1	IXSCAN	7	12502	12502	Index apellido1_1_edad_-1 was used to find matching values for apellido1,edad
2	FETCH	11	12502	12502	Retrieved documents from index fetch

The 'Statistics' section shows the following values:

Statistic	Value
Total Docs Returned	12502
Total Keys Examined	12502
Total Docs Examined	12502

The 'Namespace' is `maraton.participantes` and the 'Query' is the same as in the input. The 'Execution Time' is 220ms.

Y en el inverso:

Simple Query

50

apellid... Seq

"Pérez"

Project Fields

SortKeys

Attribute Ascending?

apellid... -1

edad 1

Limit (0=No limit)

0

Close Execute

Query Input 02 - localhost:27017

```
1 db.getSiblingDB("maraton").getCollection("participantes").find(
2   {
3     "edad": { $gte: 50 },
4     "apellido1": { $eq: "Pérez" }
5   },
6   {
7     "sort": { "edad": -1 }
8   }
9 )
```

Query Output

Raw Table Explain

Raw Json Index Advisor

Explain

IXSCAN FETCH

Seq	Step	ms	Examined	Return	Comment
1	IXSCAN	5	12502	12502	Index apellido1_edad_-1 was used to find matching values for apellido1,edad
2	FETCH	13	12502	12502	Retrieved documents from index fetch

Statistic Value

Total Docs Returned	12502
Total Keys Examined	12502
Total Docs Examined	12502

Namespace: maraton.participantes

En cambio si no todos los campos se ordenan con el orden inverso del índice, el índice se sigue usando pero es necesario ordenar en memoria:

Simple Query

50

apellid... Seq

"Pérez"

Project Fields

SortKeys

Attribute Ascending?

apellid... 1

edad 1

Limit (0=No limit)

0

Close Execute

Query Input 02 - localhost:27017

```
1 db.getSiblingDB("maraton").getCollection("participantes").find(
2   {
3     "edad": { $gte: 50 },
4     "apellido1": { $eq: "Pérez" }
5   },
6   {
7     "sort": { "apellido1": 1, "edad": 1 }
8   }
9 )
```

Query Output

Raw Table Explain

Raw Json Index Advisor

Explain

IXSCAN SORT_KEY_GENERATOR SORT FETCH

Seq	Step	ms	Examined	Return	Comment
1	IXSCAN	82	12502	12502	Index apellido1_edad_-1 was used to find matching values for apellido1,edad
2	SORT_KEY_GENERATOR	145	12502	12502	Generate keys for the next sort step
3	SORT	167	12502	12502	Documents were sorted on : apellido1. Consider creating index on \$apellido1 to support the sort
4	FETCH	189	12502	12502	Retrieved documents from index fetch

Statistic Value

Total Docs Returned	12502
---------------------	-------

Si la ordenación del set de datos supera 32 megas se interrumpe la operación.

Sort and Index Prefix

If the sort keys correspond to the index keys or an index prefix, MongoDB can use the index to sort the query results. A prefix of a compound index is a subset that consists of one or more keys at the start of the index key pattern.

Práctica

```
> db.participantes.createIndex({apellido1: 1, apellido2: 1, edad: -1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 2,
  "numIndexesAfter" : 3,
  "ok" : 1
}
```

El índice podrá ser utilizado con los siguientes prefijos en el método sort():

```
{apellido1: 1}
{apellido1: -1}
{apellido1: 1, apellido2: 1}
{apellido1: -1, apellido2: -1}
{apellido1: 1, apellido2: 1, edad: -1}
{apellido1: -1, apellido2: -1, edad: 1}
```

Particularidades:

Aunque la consulta tenga un campo no contenido en el índice, usará el índice si la ordenación contiene el prefijo, primero para seleccionar todos los documentos en el orden de la consulta y posteriormente en el fetch para recorrerlos y encontrar las coincidencias. Por ejemplo:

The screenshot shows the MongoDB Compass interface. On the left, the 'Simple Query' panel is visible with the following configuration:

- Value to filter: nombre, \$regex
- Filter value: "Juan"
- Project Fields: (empty)
- SortKeys: (empty)
- Limit: 0

The main panel displays the 'Query Input' and 'Query Output'. The 'Query Input' shows the following query:

```
db.getSiblingDB('maraton').getCollection('participantes').find(
  {
    'nombre': { '$regex': 'Juan' }
  },
  {}
)
```

The 'Query Output' shows the 'Explain' tab, which displays the execution plan. The plan consists of two steps:

Seq	Step	ms	Examined	Return	Comment
1	IXSCAN	5	100000	100000	Index apellido1_1_apellido2_1_edad_-1 was used to find matching values for apellido1,apellido2,edad
2	FETCH	23	100000	25032	Documents from previous step were scanned looking for these criteria: ('nombre': {'\$regex': 'Juan'})

The 'Explain' tab also shows a 'Statistic' table:

Statistic	Value
Total Docs Returned	25032
Total Keys Examined	100000

The bottom of the interface shows the 'Execution Time: 76ms'.

Si no coinciden ni los campos ni el prefijo del índice en la ordenación, se crea un scan de toda la colección y se ordena en memoria. Por ejemplo:

The screenshot shows the MongoDB Compass interface. On the left, the 'Simple Query' panel has 'nombre' selected as the filter field with a '\$regex' operator and the value 'Juan'. The 'SortKeys' section has 'apellido' selected as the sort field in ascending order. The 'Limit' is set to 0. The 'Query Input' on the right shows the following query:

```
db.getSiblingDB('maraton').getCollection('participantes').find(
  {
    'nombre': { '$regex': 'Juan' }
  },
  {
    'sort': { 'apellido': 1 }
  }
)
```

The 'Query Output' tab shows the 'Explain' results. The execution plan consists of three steps: COLLSCAN, SORT_KEY_GENERATOR, and SORT. The table below summarizes the execution steps:

Seq	Step	ms	Examined	Return	Comment
1	COLLSCAN	6	100000	25032	Documents were scanned looking for these criteria: ("nombre": {"\$regex": "Juan"}). Consider creating an index on these attributes.
2	SORT_KEY_GENERATOR	8		25032	Generate keys for the next sort step
3	SORT	60		25032	Documents were sorted on : apellido1. Consider creating index on \$apellido1 to support the sort

Una última particularidad es que si en la consulta existe un campo con consulta de igualdad que pueda complementar el prefijo en el sort, podrá utilizar el índice para no tener que ordenar en memoria.

Por ejemplo esta consulta no usa el índice:

The screenshot shows the MongoDB Compass interface. On the left, the 'Simple Query' panel has 'apellido2' selected as the filter field with an equality operator. The 'SortKeys' section has 'apellido2' selected as the sort field in ascending order. The 'Limit' is set to 0. The 'Query Input' on the right shows the following query:

```
db.getSiblingDB('maraton').getCollection('participantes').find(
  {
    'apellido2': 'Juan'
  },
  {
    'sort': { 'apellido2': 1 }
  }
)
```

The 'Query Output' tab shows the 'Explain' results. The execution plan consists of three steps: COLLSCAN, SORT_KEY_GENERATOR, and SORT. The table below summarizes the execution steps:

Seq	Step	ms	Examined	Return	Comment
1	COLLSCAN	9	100000	100000	All documents in the collection where scanned. No filter condition was specified
2	SORT_KEY_GENERATOR	18		100000	Generate keys for the next sort step
3	SORT	184		100000	Documents were sorted on : apellido2. Consider creating index on \$apellido2 to support the sort

Si le añadimos en la consulta un campo de igualdad que se pueda añadir a los campos de la ordenación para completar el prefijo, la consulta usará el índice y no ordenará en memoria.

The screenshot shows the MongoDB Simple Query Builder interface on the left and the Query Output on the right. The query is: `db.getSiblingDB("maraton").getCollection("participantes").find({ "apellido1": { $eq: "López" } })`.

Simple Query Builder:

- Attribute: `apellido1`, Operator: `$eq`, Value to filter: `"López"`
- SortKeys: `apellido2` (Ascending: 1)
- Limit: 0

Query Output:

Explain

IXSCAN → FETCH

Seq	Step	ms	Examined	Return	Comment
1	IXSCAN	3	25098	25098	Index apellido1_1_apellido2_1_edad_-1 was used to find matching values for apellido1,apellido2,edad
2	FETCH	13	25098	25098	Retrieved documents from index fetch

Statistic Value

Total Docs Returned	25098
Total Keys Examined	25098

Execution Time: 136ms

Siempre y cuando sea de igualdad:

The screenshot shows the MongoDB Simple Query Builder interface on the left and the Query Output on the right. The query is: `db.getSiblingDB("maraton").getCollection("participantes").find({ "apellido1": { $ne: "López" } })`.

Simple Query Builder:

- Attribute: `apellido1`, Operator: `$ne`, Value to filter: `"López"`
- SortKeys: `apellido2` (Ascending: 1)
- Limit: 0

Query Output:

Explain

IXSCAN → SORT_KEY_GENERATOR → SORT → FETCH

Seq	Step	ms	Examined	Return	Comment
1	IXSCAN	18	74903	74902	Index apellido1_1_apellido2_1_edad_-1 was used to find matching values for apellido1,apellido2,edad
2	SORT_KEY_GENERATOR	18		74902	Generate keys for the next sort step
3	SORT	138		74902	Documents were sorted on : apellido2. Consider creating index on \$apellido2 to support the sort
4	FETCH	144	74902	74902	Retrieved documents from index

Ensure Indexes Fit in RAM

For the fastest processing, ensure that your indexes fit entirely in RAM so that the system can avoid reading the index from disk.

To check the size of your indexes, use the `db.collection.totalIndexSize()` helper, which returns data in bytes.

Práctica

```
> db.participantes.totalIndexSize()
```

Indexes that Hold Only Recent Values in RAM

Indexes do not have to fit entirely into RAM in all cases. If the value of the indexed field increments with every insert, and most queries select recently added documents; then MongoDB only needs to keep the parts of the index that hold the most recent or “right-most” values in RAM. This allows for efficient index use for read and write operations and minimize the amount of RAM required to support the index.

Create Queries that Ensure Selectivity

Selectivity is the ability of a query to narrow results using the index. Effective indexes are more selective and allow MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

To ensure selectivity, write queries that limit the number of possible documents with the indexed field. Write queries that are appropriately selective relative to your indexed data.

Index intersection

MongoDB can use the intersection of multiple indexes to fulfill queries. In general, each index intersection involves two indexes; however, MongoDB can employ multiple/nested index intersections to resolve a query.

To determine if MongoDB used index intersection, run `explain()`; the results of `explain()` will include either an `AND_SORTED` stage or an `AND_HASH` stage.

Covered querys

The MongoDB covered query is one which uses an index and does not have to examine any documents. An index will cover a query if it satisfies the following conditions:

- All fields in a query are part of an index.
- All fields returned in the results are of the same index.

Práctica

```
> db.participantes.find({apellido1: "López"},{apellido1: 1, createdAt: 1, _id: 0}).explain()
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "maraton.participantes",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "apellido1" : {
        "$eq" : "López"
      }
    },
    "winningPlan" : {
      "stage" : "PROJECTION",
      "transformBy" : {
        "apellido1" : 1,
        "createdAt" : 1,
        "_id" : 0
      },
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "apellido1" : 1,
          "createdAt" : -1
        },
        "indexName" : "apellido1_1_createdAt_-1",
        ...
      }
    }
  }
}
```