

UT5.

PRÁCTICA 25-1 MONGODB

RENDIMIENTO

Curso: Especialización Big Data e Inteligencia Artificial

Autor: Carlos Pérez Astorquiza

Curso Académico: 2023 - 2024

INDICE

Contenido

<i>Rendimiento en MongoDB</i>	2
1. Ejercicio de rendimiento	2
a) Carga los datos almacenados en restaurants.zip en la colección iabd.restaurants:	2
b) Habilita la auditoria de MongoDB para registrar las consultas que tardan más de 90ms. 3	
c) Ejecuta mongotop 3 para comprobar el rendimiento de la colección iabd.restaurants.. 4	
d) Comprueba el rendimiento mediante el plan de ejecución de las diferentes consultas. 5	
e) Comprueba qué consulta es la que más tarda utilizando db.system.profile.	1
f) Crea los índices necesarios para optimizar las consultas y justifica la elección del tipo de índices y sus campos.	1
g) Comprueba que se utilizan los índices y compara el rendimiento actual respecto al previo al crear los índices.	2
2. Ejercicio de rendimiento	2
3. Ejercicio de ampliación.....	2
4. Ejercicio de ampliación.....	4

Rendimiento en MongoDB

1. Ejercicio de rendimiento

Para la siguiente actividad, se recomienda trabajar con una instalación de MongoDB en local, ya sea mediante Docker o una instalación propia.

Tenemos los datos de 1.000.000 de restaurantes con una estructura similar al siguiente documento:

```
1  {
2    _id: ObjectId("65291f2d09ae7e23eba2b3df"),
3    name: 'Perry Street Brasserie',
4    cuisine: 'French',
5    stars: 0.3,
6    address: {
7      street: '959 Iveno Square',
8      city: 'Fokemlid',
9      state: 'AL',
10     zipcode: '18882'
11   }
12 }
```

Nos han dicho que les interesa optimizar las siguientes consultas:

- Obtener la cantidad de restaurantes de un determinado tipo de cocina para un estado determinado que tenga una calificación superior a 3 estrellas.
- Obtener los datos de los restaurantes de un determinado tipo de cocina para un estado determinado que tenga una calificación comprendida entre dos valores y ordenar dichos datos por la ciudad.
- Obtener los datos de los restaurantes de una determinada ciudad que sea de un tipo de cocina, ordenados por la cantidad de estrellas.

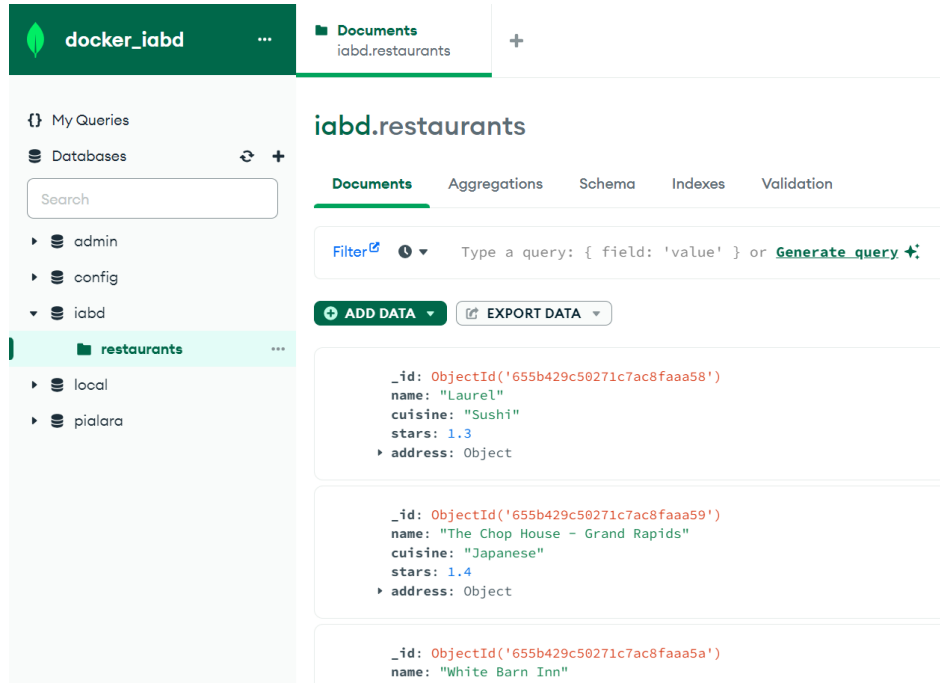
Así pues, se pide adjuntar los comandos empleados y capturas de pantalla para:

a) Carga los datos almacenados en restaurants.zip en la colección iabd.restaurants:

```
C:\Users\carpe>docker cp "C:\Users\carpe\OneDrive\Escritorio\restaurants.json" iabd-mongo:/
Successfully copied 151MB to iabd-mongo:/
```

```
C:\Users\carpe>docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                               NAMES
e02ae3ee8fae   mongo    "docker-entrypoint.s..." 4 days ago    Up 31 minutes  127.0.0.1:27017->27017/tcp         iabd-mongo
```

```
root@e02ae3ee8fae:/# mongoimport --uri "mongodb://localhost:27017/" --db iabd --collection restaurants --file /restaurants.json
2023-11-20T11:27:24.060+0000 connected to: mongodb://localhost:27017/
2023-11-20T11:27:27.060+0000 [#####.....] iabd.restaurants 37.6MB/144MB (26.2%)
2023-11-20T11:27:30.060+0000 [#####.....] iabd.restaurants 75.0MB/144MB (52.2%)
2023-11-20T11:27:33.060+0000 [#####.....] iabd.restaurants 112MB/144MB (78.1%)
2023-11-20T11:27:35.624+0000 [#####.....] iabd.restaurants 144MB/144MB (100.0%)
2023-11-20T11:27:35.624+0000 1000000 document(s) imported successfully. 0 document(s) failed to import.
root@e02ae3ee8fae:/#
```




b) Habilita la auditoria de MongoDB para registrar las consultas que tardan más de 90ms.

```
> db.setProfilingLevel(2,90);
< { was: 0, slowms: 100, sampleRate: 1, ok: 1 }
iabd> |
```

Indicamos el nivel de las consultas a auditar mediante el nivel 2 (todas las consultas) y como segundo parámetro le indicamos el número mínimo de milisegundos (90ms) de las consultas para ser auditadas.

c) Ejecuta mongotop 3 para comprobar el rendimiento de la colección iabd.restaurants

iabd-mongo					
<div>  mongo </div> <div> e02ae3ee8fae </div> <div> 27017:27017 </div>					
Logs	Inspect	Bind mounts	Exec	Files	Stats
	iabd.restaurants	0ms	0ms	0ms	
	local.system.replset	0ms	0ms	0ms	
	ns	total	read	write	2023-11-20T11:38:08Z
	admin.\$cmd.aggregate	0ms	0ms	0ms	
	admin.atlascli	0ms	0ms	0ms	
	admin.system.version	0ms	0ms	0ms	
	config.collections	0ms	0ms	0ms	
	config.system.sessions	0ms	0ms	0ms	
	config.transactions	0ms	0ms	0ms	
	iabd.restaurants	0ms	0ms	0ms	
	local.system.replset	0ms	0ms	0ms	
	ns	total	read	write	2023-11-20T11:38:11Z
	admin.\$cmd.aggregate	0ms	0ms	0ms	
	admin.atlascli	0ms	0ms	0ms	
	admin.system.version	0ms	0ms	0ms	
	config.collections	0ms	0ms	0ms	
	config.system.sessions	0ms	0ms	0ms	
	config.transactions	0ms	0ms	0ms	
	iabd.restaurants	0ms	0ms	0ms	
	local.system.replset	0ms	0ms	0ms	

Muestra el tiempo empleado por MongoDB en las diferentes colecciones, indicando tanto el tiempo empleado en lectura como en escrituras.

d) Comprueba el rendimiento mediante el plan de ejecución de las diferentes consultas.

Consulta 1:

Cantidad de restaurantes de un determinado tipo de cocina para un estado determinado que tenga una calificación superior a 3 estrellas:

```
>_MONGOSH
> db.restaurants.aggregate([
  { $match: {
    cuisine: "Japanese",
    state: "HI",
    "rating.stars": { $gt: 3 }
  }},
  { $count: "total_restaurants" }
]).explain("executionStats")
< {
  explainVersion: '2',
  stages: [
    {
      '$cursor': {
        queryPlanner: {
          namespace: 'iabd.restaurants',
          indexFilterSet: false,
          parsedQuery: {
            '$and': [
              {
                cuisine: {
                  '$eq': 'Japanese'
                }
              }
            ]
          }
        }
      }
    }
  ]
  executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 241,
    totalKeysExamined: 0,
    totalDocsExamined: 1000000,
    executionStages: {
      stage: 'mkobj',
      planNodeId: 2,
      nReturned: 0,
      executionTimeMillisEstimate: 240,
      opens: 1,
      closes: 1,
      saveState: 1001,
      restoreState: 1001,
      isEOF: 1,
      objSlot: 15,
      fields: [],
      projectFields: [
        '_id',
      ]
    }
  }
}
```

Vemos que esta primera consulta realiza un COLLSCAN (escaneo completo de la colección) y tiempo empleado de 241.

Consulta 2:

Obtener los datos de los restaurantes de un determinado tipo de cocina para un estado determinado que tenga una calificación comprendida entre dos valores y ordenar dichos datos por la ciudad:

```
> db.restaurants.find({
  cuisine: "Japones",
  state: "HI",
  "rating.stars": { $gte: 3, $lte: 4 }
}).sort({ city: 1 }).explain("executionStats")
< {
  explainVersion: '2',
  queryPlanner: {
    namespace: 'iabd.restaurants',
    indexFilterSet: false,
    parsedQuery: {
      '$and': [
        {
          cuisine: {
            '$eq': 'Japones'
          }
        },
        {
          state: {
            '$eq': 'HI'
          }
        }
      ]
    }
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 255,
    totalKeysExamined: 0,
    totalDocsExamined: 1000000,
    executionStages: {
      stage: 'sort',
      planNodeId: 2,
      nReturned: 0,
      executionTimeMillisEstimate: 247,
      opens: 1,
      closes: 1,
      saveState: 1000,
      restoreState: 1000,
      isEOF: 1,
      memLimit: 104857600,
      totalDataSizeSorted: 0,
      usedDisk: false,
      spills: 0,
    }
  }
}
```

Vemos que esta primera consulta realiza un COLLSCAN (escaneo completo de la colección) y tiempo empleado de 255.

Consulta 3:

Obtener los datos de los restaurantes de una determinada ciudad que sea de un tipo de cocina, ordenados por la cantidad de estrellas:

```
> db.restaurants.find({
  city: "HI",
  cuisine: "Japones"
}).sort({ "rating.stars": -1 }).explain("executionStats")
< {
  explainVersion: '2',
  queryPlanner: {
    namespace: 'iabd.restaurants',
    indexFilterSet: false,
    parsedQuery: {
      '$and': [
        {
          city: {
            '$eq': 'HI'
          }
        },
        {
          cuisine: {
            '$eq': 'Japones'
          }
        }
      ]
    }
  },
  executionStats: {
    nreturned: 0,
    executionTimeMillis: 237,
    totalKeysExamined: 0,
    totalDocsExamined: 1000000,
    executionStages: {
      stage: 'sort',
      planNodeId: 2,
      nReturned: 0,
      executionTimeMillisEstimate: 237,
      opens: 1,
      closes: 1,
      saveState: 1000,
      restoreState: 1000,
      isEOF: 1,
      memLimit: 104857600,
      totalDataSizeSorted: 0,
      usedDisk: false,
      spills: 0,
      spilledDataStorageSize: 0,
      orderBySlots: {
        '11': 'desc'
      }
    }
  }
}
```

Vemos que esta primera consulta realiza un COLLSCAN (escaneo completo de la colección) y tiempo empleado de 237.

e) Comprueba qué consulta es la que más tarda utilizando db.system.profile.

```
> db.system.profile.find({}, { millis: 1, _id: 0 })
< {
  millis: 255
}
{
  millis: 238
}
{
  millis: 0
}
{
  millis: 239
}
{
  millis: 237
}
```

La consulta que más tarda es aquella que tiene que comparar entre dos valores, es decir la consulta 2 que tarda 255 Millis.

f) Crea los índices necesarios para optimizar las consultas y justifica la elección del tipo de índices y sus campos.

Trato de aplicar la regla ESR, que es la regla que define el orden a la hora de crear un índice compuesto, empezando por la igualdad (*Equality*), luego la ordenación (*Sort*), y finalmente el rango (*Range*).

```
> db.restaurants.createIndex({ cuisine: 1, state: 1, "rating.stars": 1 })
< cuisine_1_state_1_rating.stars_1
iabd> |
```

Equality (Igualdad): Los campos cuisine y state son utilizados para igualdad en la consulta ({ cuisine: "<TipoDeCocina>", state: "<Estado>" }).

Sort (Ordenación): No hay un campo de ordenación en esta consulta.

Range (Rango): El campo rating.stars se utiliza en una condición de rango ({ \$gt: 3 }).

```
> db.restaurants.createIndex({ cuisine: 1, state: 1, "rating.stars": 1, city: 1 })
< cuisine_1_state_1_rating.stars_1_city_1
iabd> |
```

Equality (Igualdad): cuisine y state.

Sort (Ordenación): city (ordenación al final de la consulta con .sort({ city: 1 })).

Range (Rango): rating.stars (condición de rango con { \$gte: <ValorInferior>, \$lte: <ValorSuperior> }).

```
> db.restaurants.createIndex({ city: 1, cuisine: 1, "rating.stars": -1 })
< city_1_cuisine_1_rating.stars_-1
iabd> |
```

Equality (Igualdad): city y cuisine.

Sort (Ordenación): rating.stars (ordenación en la consulta con .sort({ "rating.stars": -1 })).

Range (Rango): No hay un campo de rango específico en esta consulta.

g) Comprueba que se utilizan los índices y compara el rendimiento actual respecto al previo al crear los índices.

Comparativas del rendimiento actual con los índices:

```
},
executionStats: {
  executionSuccess: true,
  nReturned: 0,
  executionTimeMillis: 6,
  totalKeysExamined: 0,
  totalDocsExamined: 0,
  executionStages: {
    stage: 'nlj',
    planNodeId: 2,
    nReturned: 0,
    executionTimeMillisEstimate: 0,
    opens: 1,
    closes: 2,
    saveState: 0,
    restoreState: 0,
    isEOF: 1,
    totalDocsExamined: 0,
    totalKeysExamined: 0,
    collectionScans: 0,
    collectionSeeks: 0,
    indexScans: 0
  }
}
```

```
},
executionStats: {
  executionSuccess: true,
  nReturned: 0,
  executionTimeMillis: 25,
  totalKeysExamined: 0,
  totalDocsExamined: 0,
  executionStages: {
    stage: 'sort',
    planNodeId: 3,
    nReturned: 0,
    executionTimeMillisEstimate: 0,
    opens: 1,
    closes: 2,
    saveState: 0,
    restoreState: 0,
    isEOF: 1,
    memLimit: 104857600,
    totalDataSizeScanned: 0
  }
}
```

Vemos que ahora las consultas las realiza con IXSCAN(escaneo a partir de un índice) y tiempo empleado baja a 6, 5 y 25, lo que evidencia un cambio significativo.

2. Ejercicio de rendimiento

Supongamos que tenemos un índice similar al siguiente:

```
1 | { "nombre": 1, "dir.provincia": -1, "dir.ciudad": -1, "cp": 1 }
```

Identifica los prefijos que se utilizarán (si lo hacen) para las siguientes consultas y justifica el motivo, tanto al filtrar como al ordenar los resultados:

- `db.alumnado.find({ "nombre": { $gt: "P" } }).sort({ "dir.ciudad": -1 })`
- `db.alumnado.find({ "nombre": "Marina" }).sort({ "dir.provincia": 1, "dir.ciudad": 1 })`

- `db.alumnado.find({ "nombre": "Marina", "dir.provincia": { $lt: "S" } }).sort({ "dir.provincia": 1 })`
- `db.alumnado.find({ "dir.ciudad": "Elche" }).sort({ "dir.ciudad": -1 })`
- `db.alumnado.find({ "dir.provincia": "Alicante", "nombre": "Marina" }).sort({ "dir.ciudad": -1 })`

Partiendo de lo que hemos visto en clases este índice compuesto tiene cuatro campos, por tanto, podríamos utilizarlo al consultar sobre el primer campo, sobre el primer y segundo campo, sobre el primer y segundo campo y tercer campo, o sobre los cuatro campos.

Suponiendo que representamos el índice de la siguiente forma (A, -B, -C, D) podemos hacer una consulta que filtre por (A,B) y que ordene por (C), y estaría utilizando el índice compuesto.

Además, si el orden de los campos es inverso, mientras el orden del índice al completo se respete, se seguirá empleado el índice, recorriendo el índice en modo backward (hacia atrás) (en vez del modo forward, hacia adelante).

Partiendo de esta premisa vamos a analizar las consultas:

- `db.alumnado.find({ "nombre": { $gt: "P" } }).sort({ "dir.ciudad": -1 })`

Análisis: La consulta filtra por nombre y ordena por dir.ciudad. El índice compuesto { "nombre": 1, "dir.provincia": -1, "dir.ciudad": -1, "cp": 1 } puede ser utilizado para la operación de filtrado. Para la ordenación, aunque dir.ciudad es parte del índice, dir.provincia no está siendo usado en la consulta, por lo cual MongoDB no utilizara el índice porque no sigue el orden establecido.

- `db.alumnado.find({ "nombre": "Marina" }).sort({ "dir.provincia": 1, "dir.ciudad": 1 })`

Análisis: Filtra por nombre y ordena por dir.provincia y dir.ciudad en orden ascendente. El índice compuesto { "nombre": 1, "dir.provincia": -1, "dir.ciudad": -1, "cp": 1 } puede ser utilizado para la operación de filtrado. Aunque el orden de clasificación en la consulta es opuesto al del índice, MongoDB puede recorrer el índice en sentido inverso para soportar la ordenación.

- `db.alumnado.find({ "nombre": "Marina", "dir.provincia": { $lt: "S" } }).sort({ "dir.provincia": 1 })`

Análisis: Esta consulta utiliza dos campos del índice para filtrar y uno para ordenar. El índice compuesto {"nombre": 1, "dir.provincia": -1, "dir.ciudad": -1, "cp": 1 } puede ser utilizado para la operación de filtrado. Aunque la dirección de dir.provincia en la consulta es opuesta a la del índice, es probable que MongoDB aún utilice el índice para la operación de ordenación.

- `db.alumnado.find({ "dir.ciudad": "Elche" }).sort({ "dir.ciudad": -1 })`

Filtrado: La consulta filtra por dir.ciudad, que no es el primer campo del índice compuesto. En este caso, el índice no se puede utilizar.

Ordenación: La consulta ordena por dir.ciudad en orden descendente, que coincide con la dirección en el índice. Sin embargo, dado que el filtrado no utiliza el índice de manera correcta no se puede usar.

- `db.alumnado.find({ "dir.provincia": "Alicante", "nombre": "Marina" }).sort({ "dir.ciudad": -1 })`

Tal como vimos en clases *“si hacemos una consulta por ambos campos, da igual el orden que utilizamos en la búsqueda que siempre va a utilizar el índice”*.

Filtrado: La consulta no utiliza los campos del índice en el orden en que están definidos. El índice comienza con nombre, seguido por dir.provincia, pero la consulta invierte este orden, no obstante el índice se aplicara.

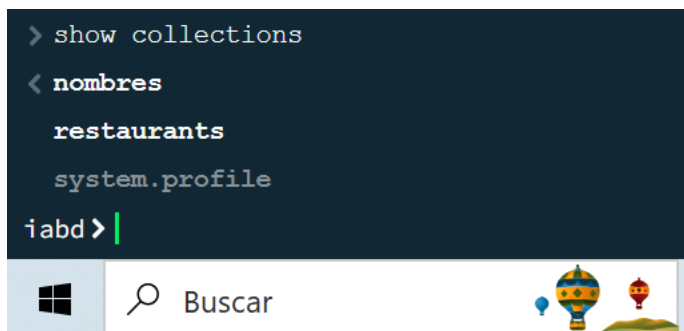
Ordenación: La consulta ordena por dir.ciudad en orden descendente, que es parte del índice.

3. Ejercicio de ampliación

Realiza las siguientes acciones anotando el comando necesario:

Crea una colección llamada nombres.

```
> show collections
< nombres
  restaurants
  system.profile
iabd> |
```



Añade 5 documentos con tu nombre con diferentes combinaciones de mayúsculas y minúsculas, pero ninguno que esté todo en minúsculas.

```
>_MONGOSH
> db.nombres.insertMany([
  { "nombre": "Carlos" },
  { "nombre": "CARLOS" },
  { "nombre": "caRlos" },
  { "nombre": "CarLos" },
  { "nombre": "carLOS" }
])
< {
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("655c899781d950c99f7e26c6"),
    '1': ObjectId("655c899781d950c99f7e26c7"),
    '2': ObjectId("655c899781d950c99f7e26c8"),
    '3': ObjectId("655c899781d950c99f7e26c9"),
    '4': ObjectId("655c899781d950c99f7e26ca")
  }
}
iabd>
```

Crea una consulta que busque tu nombre en minúsculas.

```
> db.nombres.find({ "nombre": "carlos" })
<
iabd> |
```

Crea un índice con el locale en español sobre el nombre.

```
> db.nombres.createIndex({ "nombre": 1 }, { collation: { locale: "es", strength: 1 } })
< nombre_1
```

Repite la consulta del punto 3 y comprueba cómo devuelve todos los documentos existentes.

```
>_MONGOSH
> db.nombres.find({ "nombre": "carlos" }).collation({ locale: "es", strength: 1 })
< {
  _id: ObjectId("655c899781d950c99f7e26c6"),
  nombre: 'Carlos'
}
{
  _id: ObjectId("655c899781d950c99f7e26c7"),
  nombre: 'CARLOS'
}
{
  _id: ObjectId("655c899781d950c99f7e26c8"),
  nombre: 'caRlos'
}
{
  _id: ObjectId("655c899781d950c99f7e26c9"),
  nombre: 'CarLos'
}
{
```

4. Ejercicio de ampliación

Investiga para qué sirven los índices ocultos en MongoDB, indicando al menos dos casos de uso donde tiene sentido su utilización.

Los índices ocultos en MongoDB son una característica que permite a los administradores de bases de datos "ocultar" un índice para que no sea utilizado en las operaciones de planificación de consultas, sin necesidad de eliminarlo completamente. Esta funcionalidad es útil en varias situaciones:

1. Evaluación de Impacto de Eliminar un Índice

Un caso común de uso para los índices ocultos es cuando estás considerando eliminar un índice, pero primero quieres evaluar el impacto que tendría su eliminación en el rendimiento del sistema. Al ocultar el índice en lugar de eliminarlo, puedes:

- **Probar el Rendimiento sin el Índice:** Observar cómo se comporta la base de datos sin ese índice específico. Esto incluye ver cómo se afectan las consultas que anteriormente lo utilizaban.

- **Revertir Cambios Fácilmente:** Si descubres que eliminar el índice perjudica el rendimiento, puedes volver a hacerlo visible con un cambio de configuración simple, sin necesidad de recrearlo desde cero.

2. Realizar Mantenimiento o Modificaciones en Índices

En algunos casos, puede ser necesario realizar mantenimiento o modificaciones en un índice existente (por ejemplo, cambiar su tamaño o sus opciones de colación). Los índices ocultos pueden ser útiles en estos escenarios:

- **Minimizar el Tiempo de Inactividad:** Puedes ocultar temporalmente un índice mientras realizas cambios en él o construyes un índice de reemplazo. Esto puede ayudar a reducir el tiempo de inactividad, ya que no eliminas el índice original hasta que el nuevo índice esté listo para ser utilizado.
- **Probar Nuevos Índices:** Si estás introduciendo un nuevo índice y quieres comparar su rendimiento con un índice existente, puedes ocultar temporalmente el índice antiguo para evaluar cómo el nuevo índice afecta las operaciones de consulta.