Olá, aluno(a)!
Seja bem-vindo(a) à aula interativa!

Você entrará na reunião com a câmera e o microfone desligados.

Sua presença será computada através da enquete.
Fique atento(a) e não deixe de respondê-la!

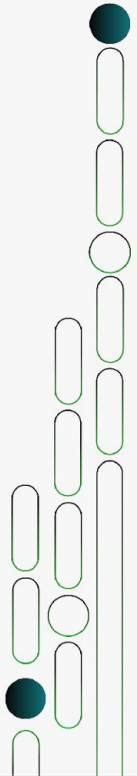
XPe
Aprenda com quem faz

> Performance & Otimização

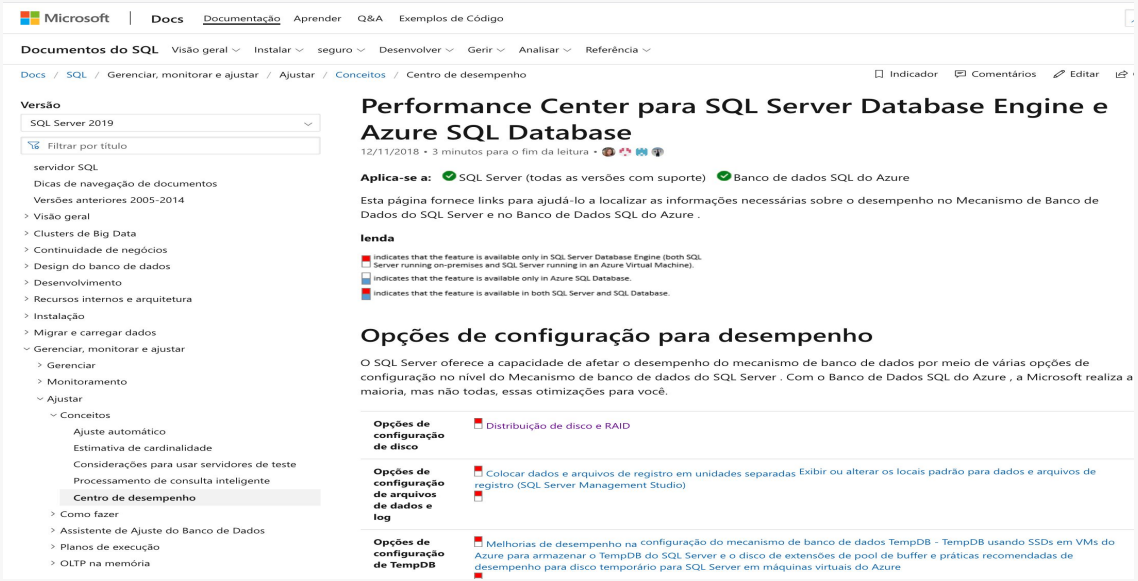
Segunda Aula Interativa
Prof. Adriano G. D. Ferreira

Nesta aula

- Centro de Desempenho SQL Server
- Export e Import Wizard do SQL Server
- Análise do Plano de Execução de Query e Operadores
- Dicas de Otimização
- Sobre o Desafio

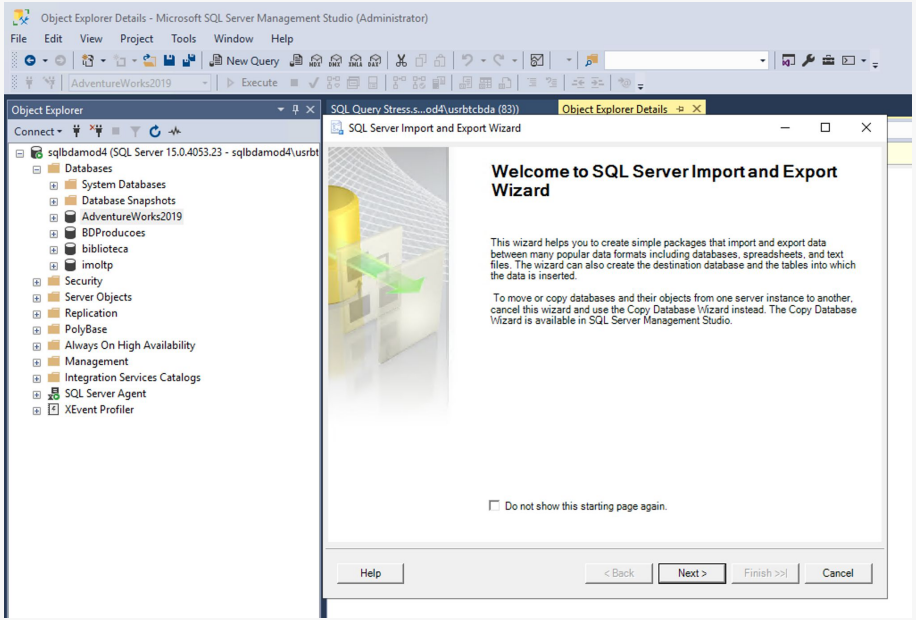


Centro de Desempenho SQL Server

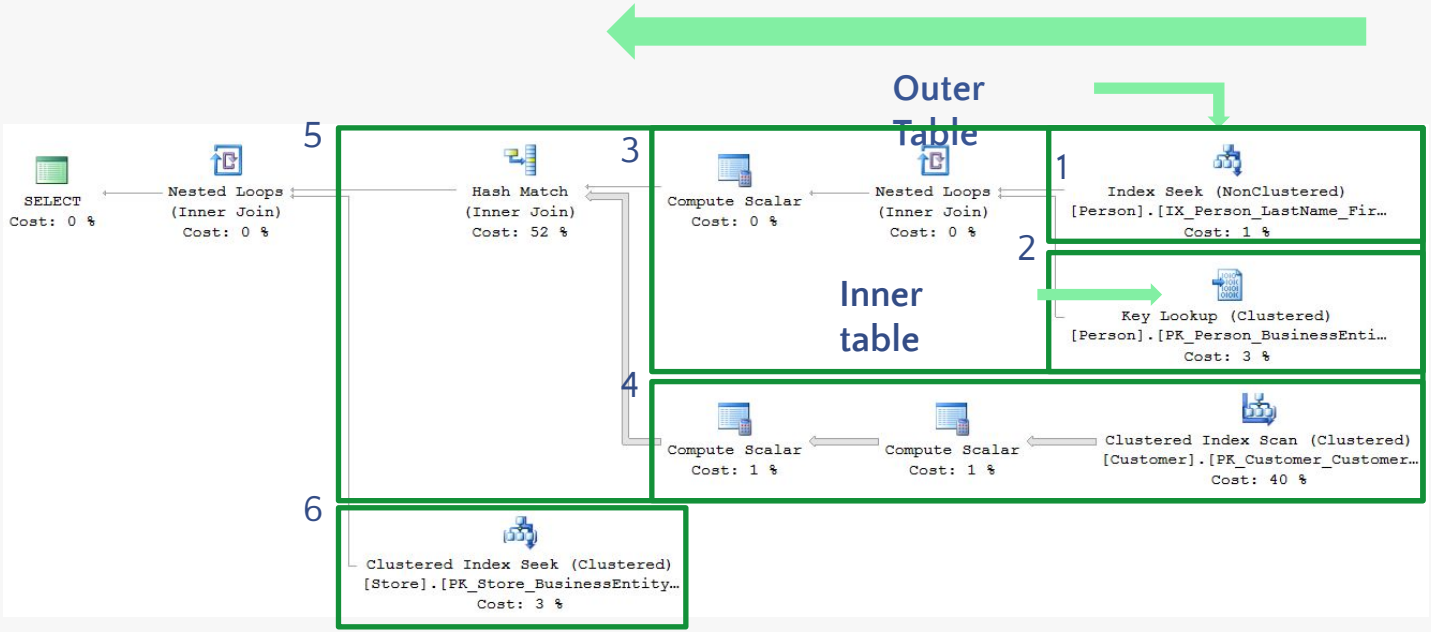


https://docs.microsoft.com/pt-br/sql/relational-databases/performance/performance-center-for-sql-server-database-engine-and-azure-sql-database?view=sql-server-ver15

Export e Import Wizard



Análise do Plano de Execução (SQL)



Operadores

- **Operadores Seek**
  - ✔ *Index Seek*
  - ✔ *Clustered Index Seek*

  - O operador *Clustered Index Seek* usa a capacidade dos índices para recuperar linhas de um índice clusterizado (clustered index);
  - O operador *Index Seek* usa a capacidade de busca de índices para recuperar linhas de um índice não clusterizado.

Operadores

- **Operadores Lookup**
  - ✔ *Row Identifier (RID) Lookup*
  - ✔ *Key Lookup*
    - *RID Lookup* é uma pesquisa em uma tabela ***HEAP*** usando um identificador de linha fornecido (RID) para pesquisar a linha na tabela e o nome da tabela na qual a linha é pesquisada.
    - *Key Lookup* é uma pesquisa em uma tabela com um índice clusterizado. É sempre acompanhada por um operador de loops aninhados. Esse operador pode indicar que a consulta pode se beneficiar se for criado um índice mais abrangente.

Operadores

- **Operadores Scan**
  - ✔ *Table Scan*
  - ✔ *Clustered Index Scan*
  - ✔ *Index Scan*
  - ✔ *Columnstore Index Scan*

  - – O operador *Table scan* percorre todas as linhas da tabela para retornar as linhas desejadas na query.
  - – Demais operadores percorrem toda a estrutura do índice para retornar as linhas desejadas na query.

Operadores

- **Outros operadores**

  - ✔ *Table Spool*
  - ✔ *Index Spool*
  - ✔ *Sort*

    - *Table Spool* coloca uma cópia de cada linha em uma tabela de spool oculta que é armazenada no banco de dados *tempdb* e existe apenas durante o tempo de vida da query.
    - *Index Spool* coloca uma cópia de cada linha em um arquivo de spool oculto (armazenado no *tempdb* e existindo apenas durante o tempo de vida da consulta) e cria um índice não cluster □permite usar a capacidade de busca de índices (*Index Seek*).

Dicas para Otimizar Plano de Query

Revisar e otimizar os JOINS

- Nested loops apenas para conjuntos pequenos de dados.

Rever operadores SCANS e RID

- Removê-los usando índices que contemplem toda a query.

Rever operadores Key Lookups

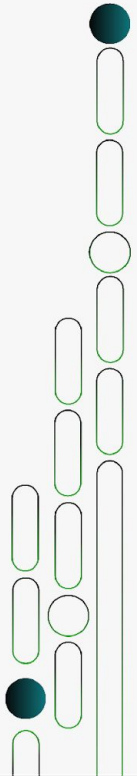
- Removê-los usando índices que contemplem toda a query.

Outras considerações

- Eliminar spools.

Dicas para Otimizar Plano de Query

- Limitar a quantidade de JOINS
- Limitar a quantidade de linhas nos JOINS
- Criar índices nas colunas dos JOINS
- Fazer JOINS usando as colunas mais únicas
- Fazer JOINS com colunas do mesmo tipo de dados (evitar conversão implícita)
- Evitar usar SELECT *
  - Sempre selecione explicitamente as colunas que você deseja retornar
- Usar a sintaxe ANSI nos JOINS
  - SELECT fname, lname, department
    FROM names **INNER JOIN** departments
    ON names.employeeid = departments.employeeid



Dicas para Otimizar Plano de Query

Evitar lógicas negativas, como !=, <>, NOT (…)

- Isso introduz contenção adicional, porque muitas vezes resulta na avaliação de cada linha (index scan).

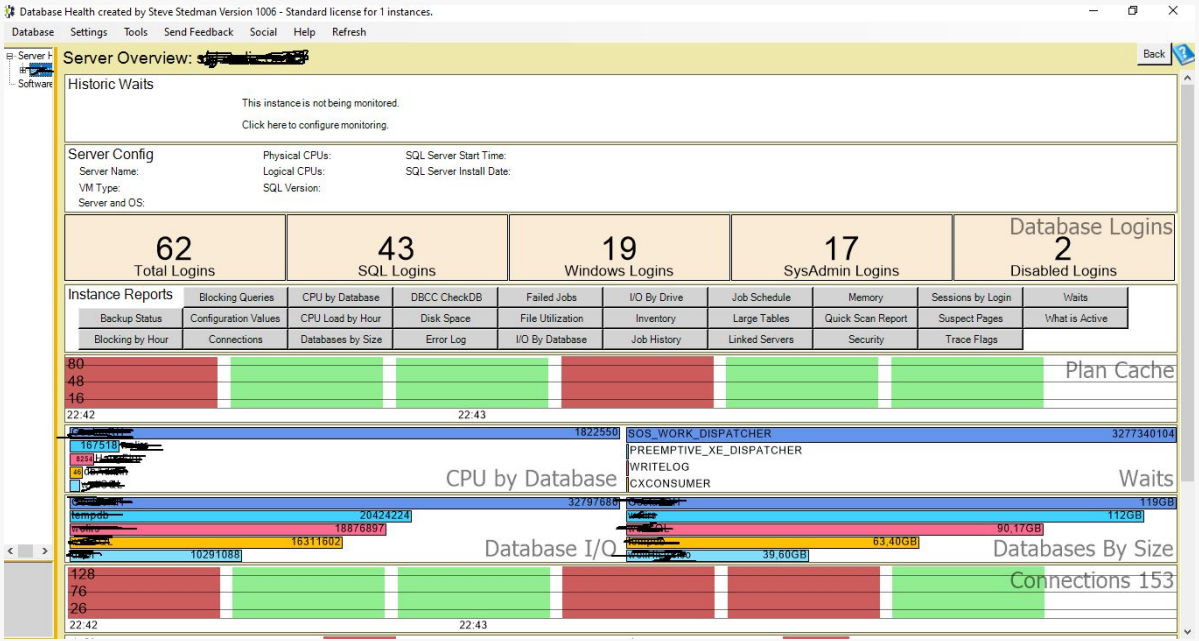
Só usar ORDER BY se for estritamente necessário

- Se estiver coberto por um índice, garanta que a ordenação do índice é a mesma da cláusula ORDER BY.

Evitar o operador LIKE com caracteres curingas ('%VALOR%'), pois sempre causará um table scan

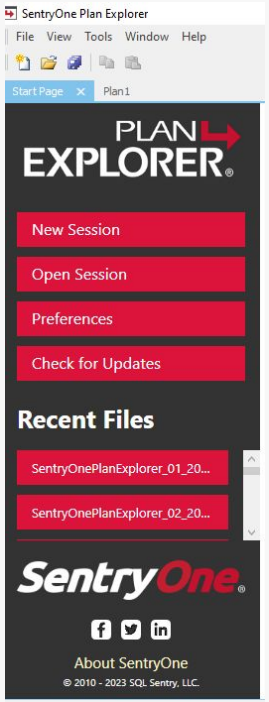
- Se tiver que usar LIKE, torne o primeiro caracter um literal ('VALOR%').

Dbs Relacionais: Performance

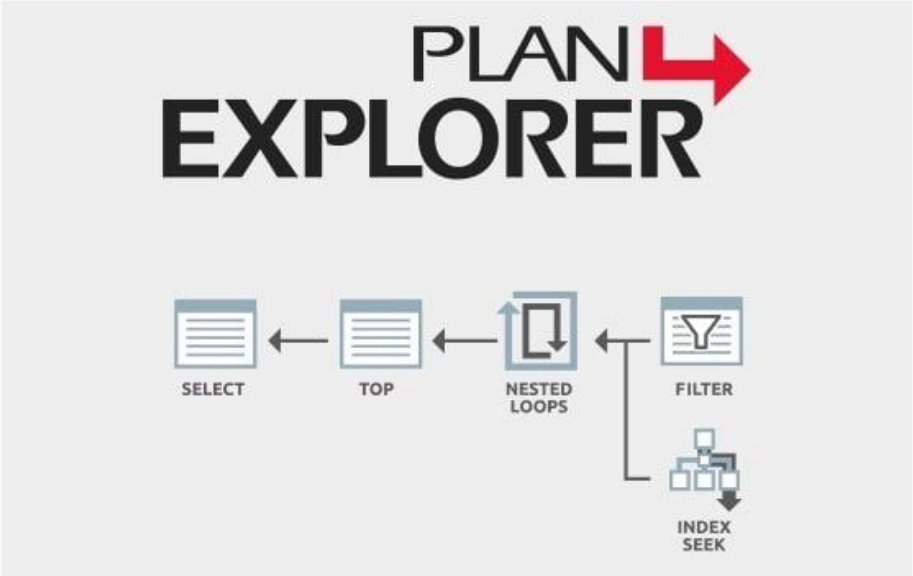


Fonte: http://databasehealth.com/

Dbs Relacionais: Performance



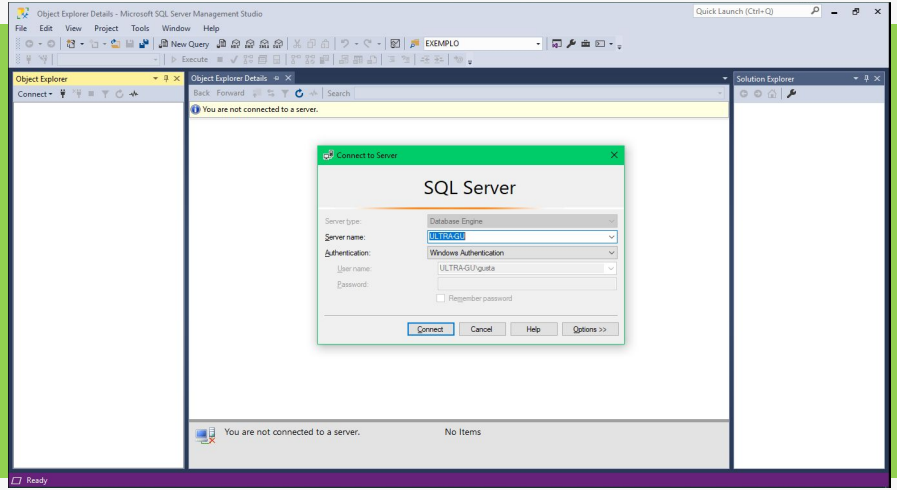
- Integrado ao Plano de Execução (Execution Plan) do Sql Server.



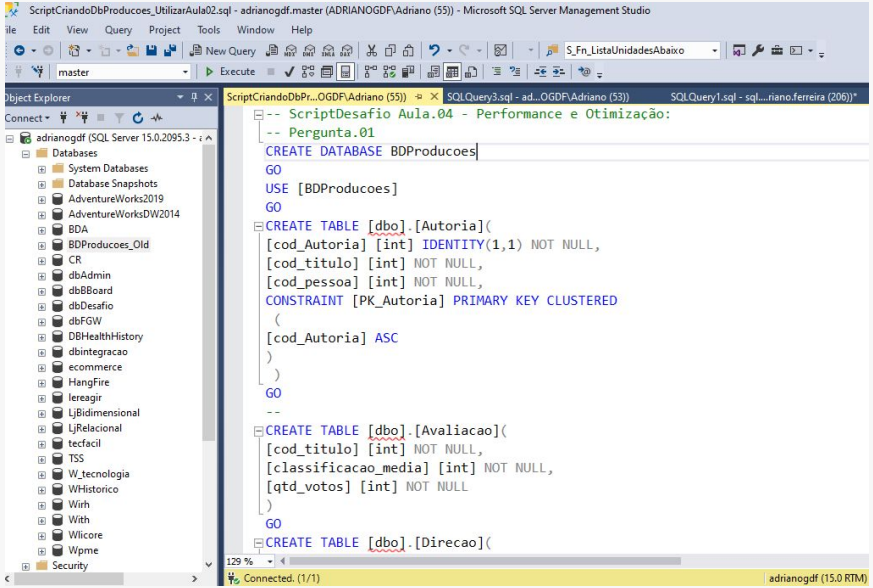
https://www.sentryone.com/plan-explorer

Análise do Plano de Execução

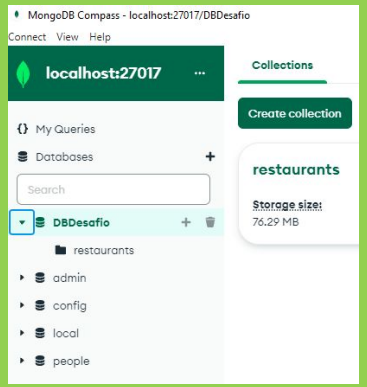
Demo



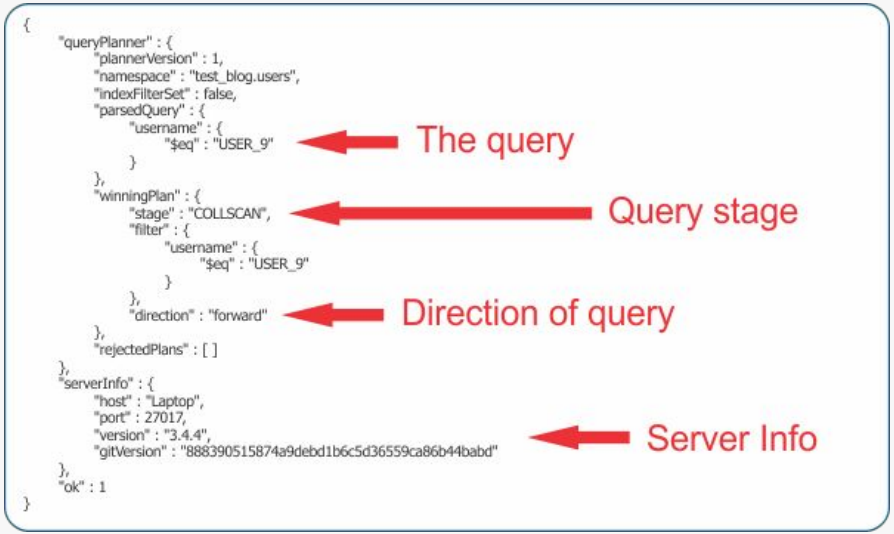
Primeiro: Criar o banco ‘Desafio’ e fazer as cargas:



Análise do Plano de Execução



Análise do Plano de Execução (MongoDB)



Dicas para Otimizar Plano de Query (MongoDB)

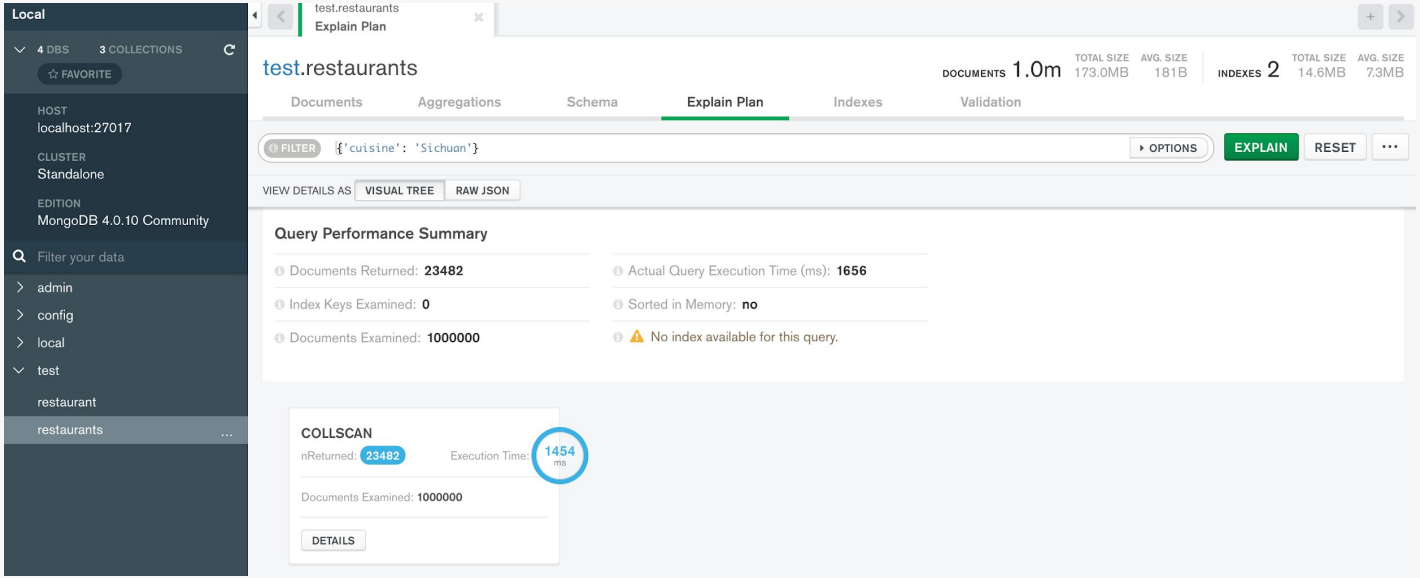
Ordem dos Campos no Índice:

- **1º. Equalidade**
- **2º. Ordenação**
- **3º. Range**

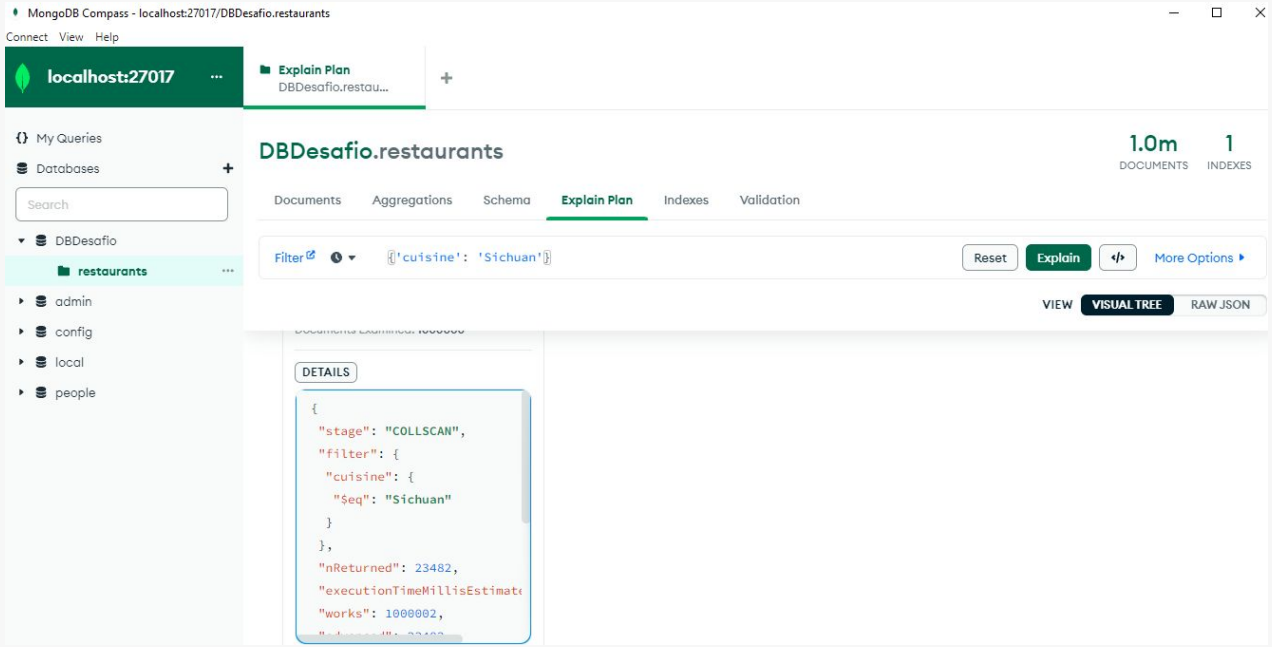
exp.find({ "address.zipcode": { $gt: '50000' }, cuisine: 'Sushi' }).sort({ stars: -1 })

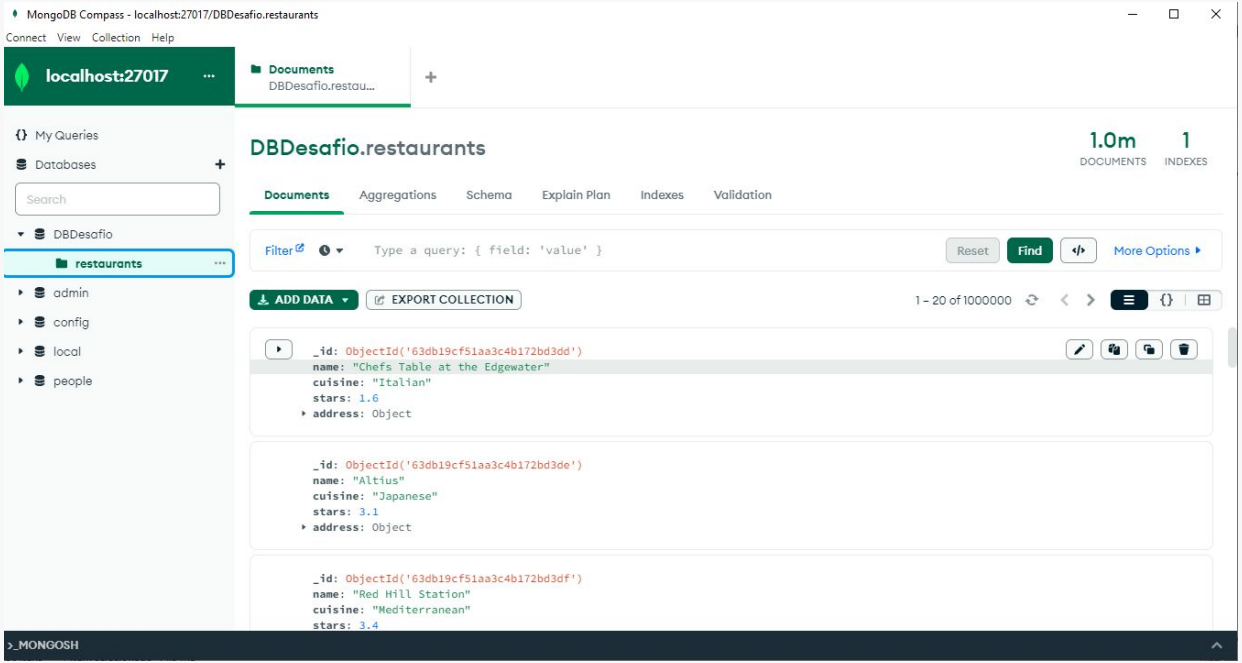
db.restaurants.createIndex({ "cuisine": 1, "stars": 1, "address.zipcode": 1 })

Explain Query (MongoDB Compass)

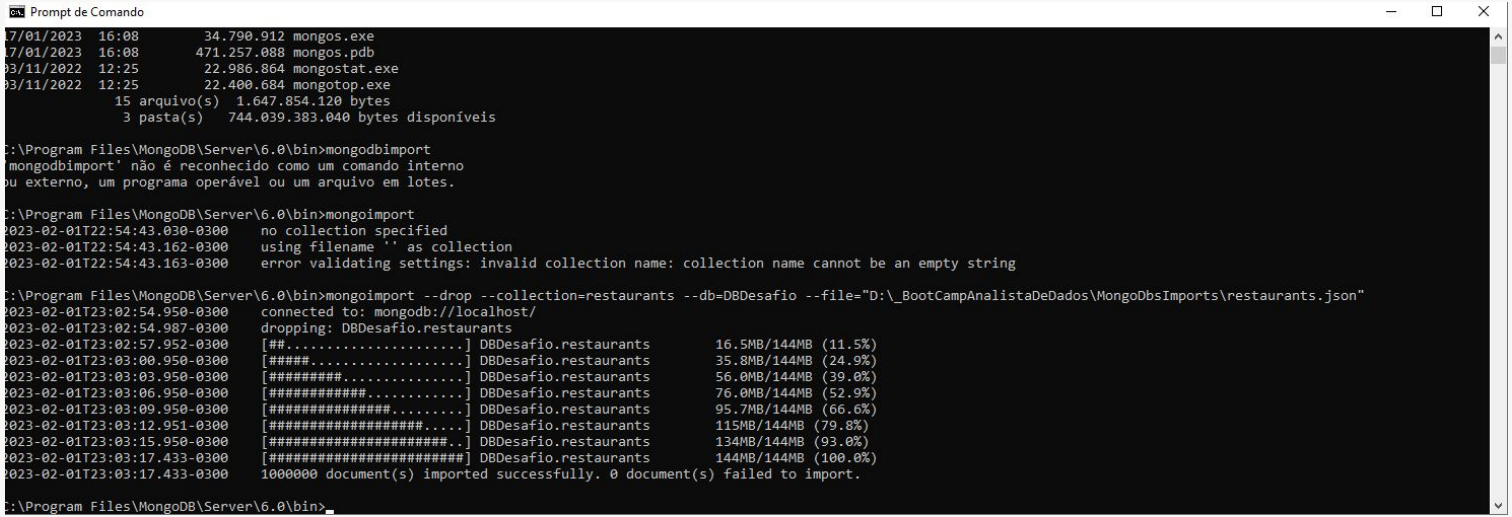


Explain Query (MongoDB Compass)



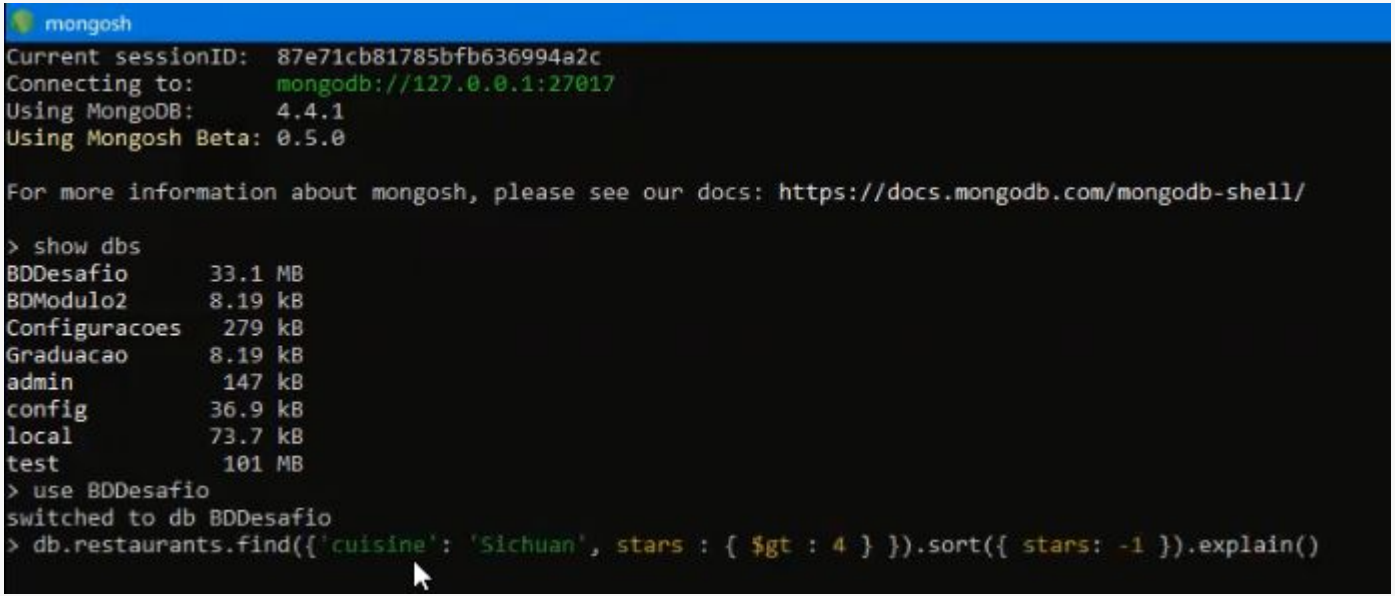

MongoDB Compass - localhost:27017/DBDesafio.restaurants
Connect View Collection Help
localhost:27017
Documents
DBDesafio.restau...
My Queries
Databases
Search
DBDesafio
restaurants
admin
config
local
people
DBDesafio.restaurants
1.0m
DOCUMENTS
1
INDEXES
Documents
Aggregations
Schema
Explain Plan
Indexes
Validation
Filter
Type a query: { field: 'value' }
Reset
Find
More Options
ADD DATA
EXPORT COLLECTION
1 - 20 of 1000000
_id: ObjectId('63db19cf51aa3c4b172bd3dd')
name: "Chefs Table at the Edgewater"
cuisine: "Italian"
stars: 1.6
address: Object
_id: ObjectId('63db19cf51aa3c4b172bd3de')
name: "Altius"
cuisine: "Japanese"
stars: 3.1
address: Object
_id: ObjectId('63db19cf51aa3c4b172bd3df')
name: "Red Hill Station"
cuisine: "Mediterranean"
stars: 3.4
>_MONGOSH

Export Collections



mongoimport --drop --collection=restaurants --db=DBDesafio --file="D:_BootCampAnalistaDeDados\MongoDbsImports\restaurants.json"
mongoimport --drop --collection=people --db=people --file="D:_BootCampAnalistaDeDados\MongoDbsImports\people.json"

Verificando indexação DBDesafio criado:



Verificando indexação DBDesafio: Questão 11:

```
> use BDDesafio
switched to db BDDesafio
> db.restaurants.find({'cuisine': 'Sichuan', stars : { $gt : 4 } }).sort({ stars: -1 }).explain()
{
  queryPlanner: {
    plannerVersion: 1,
    namespace: 'BDDesafio.restaurants',
    indexFilterSet: false,
    parsedQuery: {
      '$and': [ { cuisine: { '$eq': 'Sichuan' } }, { stars: { '$gt': 4 } } ]
    },
    winningPlan: {
      stage: 'SORT',
      sortPattern: { stars: -1 },
      memLimit: 104857600,
      type: 'simple',
      inputStage: {
        stage: 'COLLSCAN',
        filter: { '$and': [ { cuisine: [Object] }, { stars: [Object] } ] },
        direction: 'forward'
      }
    },
    rejectedPlans: []
  },
  serverInfo: {
    host: 'SERVER-GU',
    port: 27017,
    version: '4.4.1',
    gitVersion: 'ad91a93a5a31e175f5cbf8c69561e788bbc55ce1'
  },
  ok: 1
}
>
```

Verificando indexação DBDesafio: Questão 11: Índices

```
> db.restaurants.createIndex({ "cuisine": 1 , "stars" : -1 })
```

índice com ‘cousine’ e ‘stars’ -1, para tratar um ‘CollScan’ (‘Collection Scan’).

Verificando indexação DBDesafio: Questão 11: Índices

```
> db.restaurants.find({'cuisine': 'Sichuan', stars : { $gt : 4 } }).sort({ stars: -1 }).explain()
{
  queryPlanner: {
    plannerVersion: 1,
    namespace: 'BDDesafio.restaurants',
    indexFilterSet: false,
    parsedQuery: {
      '$and': [ { cuisine: { '$eq': 'Sichuan' } }, { stars: { '$gt': 4 } } ]
    },
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { cuisine: 1, stars: -1 },
        indexName: 'cuisine_1_stars_-1',
        isMultiKey: false,
        multiKeyPaths: { cuisine: [], stars: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: {
          cuisine: [ '["Sichuan", "Sichuan"]' ],
          stars: [ '[inf.0, 4)' ]
        }
      }
    },
    rejectedPlans: []
  },
  serverInfo: {
    host: 'SERVER-GU',
    port: 27017,
    version: '4.4.1',
    gitVersion: 'ad91a93a5a31e175f5cbf8c69561e788bbc55ce1'
  },
  ok: 1
}
>
```

Com o índice, tenho agora um ‘IXSCAN’, que é situação melhor que ‘CollScan’.
O índice trata primeiro a condição de igualdade, depois o ‘sort’ e a situação de Range.

Verificando indexação DBDesafio: Questão 12: Índices

```
> db.restaurants.createIndex({ "cuisine": 1 })
{
  createdCollectionAutomatically: false,
  numIndexesBefore: 1,
  numIndexesAfter: 2,
  ok: 1
}
> db.restaurants.createIndex({ "stars" : -1 })
{
  createdCollectionAutomatically: false,
  numIndexesBefore: 2,
  numIndexesAfter: 3,
  ok: 1
}
>
```

A criação de dois índices para tratar primeiro o filtro: ‘cuisine: ‘Sushi’ e depois o sort: ‘stars: 1’. Tratando duas operações com índices diferentes e não a partir de um índice composto.

Verificando indexação DBDesafio: Questão 12: Índices

- Os dois índices criados resolveram a situação, agora com o ‘IXScan’.
- Importante testar e validar sempre.
- Fica mais barato fazer o ‘fetch’ do que fazer ‘sort’ em memória, não precisando de um índice composto para isto.
- Precisa da análise em conjunto das ‘querys’ envolvidas.

Verificando indexação DBDesafio: Sort: name

```
> db.restaurants.find({ "stars": { $gte: 4 }, cuisine: 'Italian' }).sort({ name: 1 }).explain()
```

‘find’ por cozinha ‘cuisine’ e ordenação ‘sort’ por nome ‘name’.

Verificando indexação DBDesafio: Sort: name

```
> db.restaurants.find({ "stars": { $gte: 4 }, cuisine: 'Italian' }).sort({ name: 1 }).explain()
{
  queryPlanner: {
    plannerVersion: 1,
    namespace: 'BDDesafio.restaurants',
    indexFilterSet: false,
    parsedQuery: {
      '$and': [ { cuisine: { '$eq': 'Italian' } }, { stars: { '$gte': 4 } } ]
    },
    winningPlan: {
      stage: 'SORT',
      sortPattern: { name: 1 },
      memLimit: 104857600,
      type: 'simple',
      inputStage: {
        stage: 'FETCH',
        filter: { stars: { '$gte': 4 } },
        inputStage: {
          stage: 'IXSCAN',
          keyPattern: { cuisine: 1 },
          indexName: 'cuisine_1',
          isMultiKey: false,
          multiKeyPaths: { cuisine: [] },
          isUnique: false,
          isSparse: false,
          isPartial: false,
          indexVersion: 2,
          direction: 'forward',
          indexBounds: { cuisine: [ '["Italian", "Italian"]' ] }
        }
      }
    },
    rejectedPlans: [
      {
        stage: 'SORT',
        sortPattern: { name: 1 },
        memLimit: 104857600,
        type: 'simple',
        inputStage: {
          stage: 'FETCH',
          filter: { cuisine: { '$eq': 'Italian' } },
          inputStage: {
            stage: 'IXSCAN',
            keyPattern: { stars: -1 },
            indexName: 'stars_-1',
            isMultiKey: false,
            multiKeyPaths: { stars: [] },
```

Índice criado anteriormente ajudando ‘query’ com o ‘IxScan’.

Listando índices criados no MongoDB:

```
> db.restaurants.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { cuisine: 1 }, name: 'cuisine_1' },
  { v: 2, key: { stars: -1 }, name: 'stars_-1' }
]
>
```

Comando para verificar índices criados.
Um pelo campo id e dois utilizados para filtrar pelo campo ‘name’.

Sobre o Desafio

Objetivos

Exercitar os seguintes conceitos trabalhados no Módulo:

- Análise e otimização de plano de queries no SQL Server e MongoDB
- Cargas e expurgos massivos de dados no SQL Server e MongoDB

Enunciado

O projeto para o qual você foi contratado no Módulo 2 entra agora em sua segunda fase, onde será feita a otimização das ‘queries’ que foram implementadas no ‘SQL Server’.

Atividades

1. Criar o banco **BDProducoes** e seu schema físico usando o script abaixo;

...

Obrigado pela participação!

Bons estudos!

Sucesso!

Linked-in: https://www.linkedin.com/in/adrianogdiasferreira

Lattes: http://lattes.cnpq.br/3393239069676217

Email: adrianogdf@gmail.com