

# MongoDB

Aula 6:

Aggregate Framework

Banco de Dados 2 (BD2)



# Lembrando nosso techBank

Coleções com documentos de Clientes, Contas e Endereços.

```
techBank> show collections
clientes
contas
enderecos
techBank> db.clientes.count()
250
techBank> db.contas.count()
262
techBank> db.enderecos.count()
258
```

# count



Podemos usar o **COUNT** de maneiras diferentes:

```
db.collection.find().count()
```

```
db.collection.countDocuments()  <- Não use so o count()
```

```
db.collection.aggregate({$count:"alias"})
```

```
techBank> db.contas.aggregate({$count:"Quantidade total de contas cadastradas"})  
[ { 'Quantidade total de contas cadastradas': 262 } ]
```

# count



Imagine que agora você precisa contar quantas contas tem de cada tipo, como você faria?

Poderíamos fazer vários comandos, como:

```
techBank> db.contas.distinct("tipo")  
[ 'Conta corrente', 'Conta poupança', 'Conta salário' ]
```

```
techBank> db.contas.find({tipo:"Conta corrente"}).count()  
89
```

```
techBank> db.contas.find({tipo:"Conta poupança"}).count()  
89
```

```
techBank> db.contas.find({tipo:"Conta salário"}).count()  
84
```

# count



Mas isso para um sistema vai ficar ruim, muitas queries, muitas requisições e o mais importante, muito custoso para o banco de dados. Isso pode ser simplificado com o aggregate, vamos ver o exemplo abaixo:

```
techBank> db.contas.aggregate({$group:{_id:"$tipo","Total de Contas":{$count:{}}}})
[
  { _id: 'Conta poupança', 'Total de Contas': 89 },
  { _id: 'Conta corrente', 'Total de Contas': 89 },
  { _id: 'Conta salário', 'Total de Contas': 84 }
]
```

Nele usamos o operador **\$group** mas aplicado em um Framework de Agregação, ou simplesmente o Aggregate Framework.

# Aggregate Framework



## O que é?

- O Aggregation Framework é uma poderosa ferramenta de análise de dados no MongoDB.
- Ele permite a realização de operações avançadas de manipulação e transformação de dados.

## Por que usar?

- É ideal para responder a perguntas complexas sobre seus dados.
- Oferece flexibilidade e poder para realizar análises sofisticadas e cálculos em tempo real.
- Ajuda a evitar a necessidade de transferir grandes volumes de dados para o cliente para análise, reduzindo a sobrecarga de rede.

# Aggregate Framework



Permite criar pipelines de etapas (estágios) de processamento para transformar, filtrar e calcular dados.

Aplicável a diferentes cenários, desde análises simples até análises complexas e relatórios.

Exemplos:

**\$count:** Conta campos ou valores de campos.

**\$match:** Filtra documentos com base em critérios especificados.

**\$group:** Agrupa documentos e realiza cálculos de agregação, como soma, média, etc.

**\$project:** Controla quais campos são incluídos na saída final.

**\$sort:** Ordena documentos com base em um ou mais campos.

**\$limit** e **\$skip:** Permitem limitar e pular resultados para paginação.

E outros estágios que podem ser encadeados para formar um pipeline de agregação.

# Aggregate Framework



<https://www.mongodb.com/docs/v4.4/reference/operator/aggregation-pipeline/>

- No método `db.collection.aggregate()` e no método `db.aggregate()`, os estágios do pipeline aparecem em um array.
- Os documentos passam pelas etapas em sequência.



# Aggregate Framework



## *Exemplo de Uso*

Imagine um cenário de análise de vendas, onde desejamos calcular a receita total por categoria de produto.

Usando o Aggregation Framework, podemos criar um pipeline que filtra, agrupa e calcula essas informações de maneira eficiente.

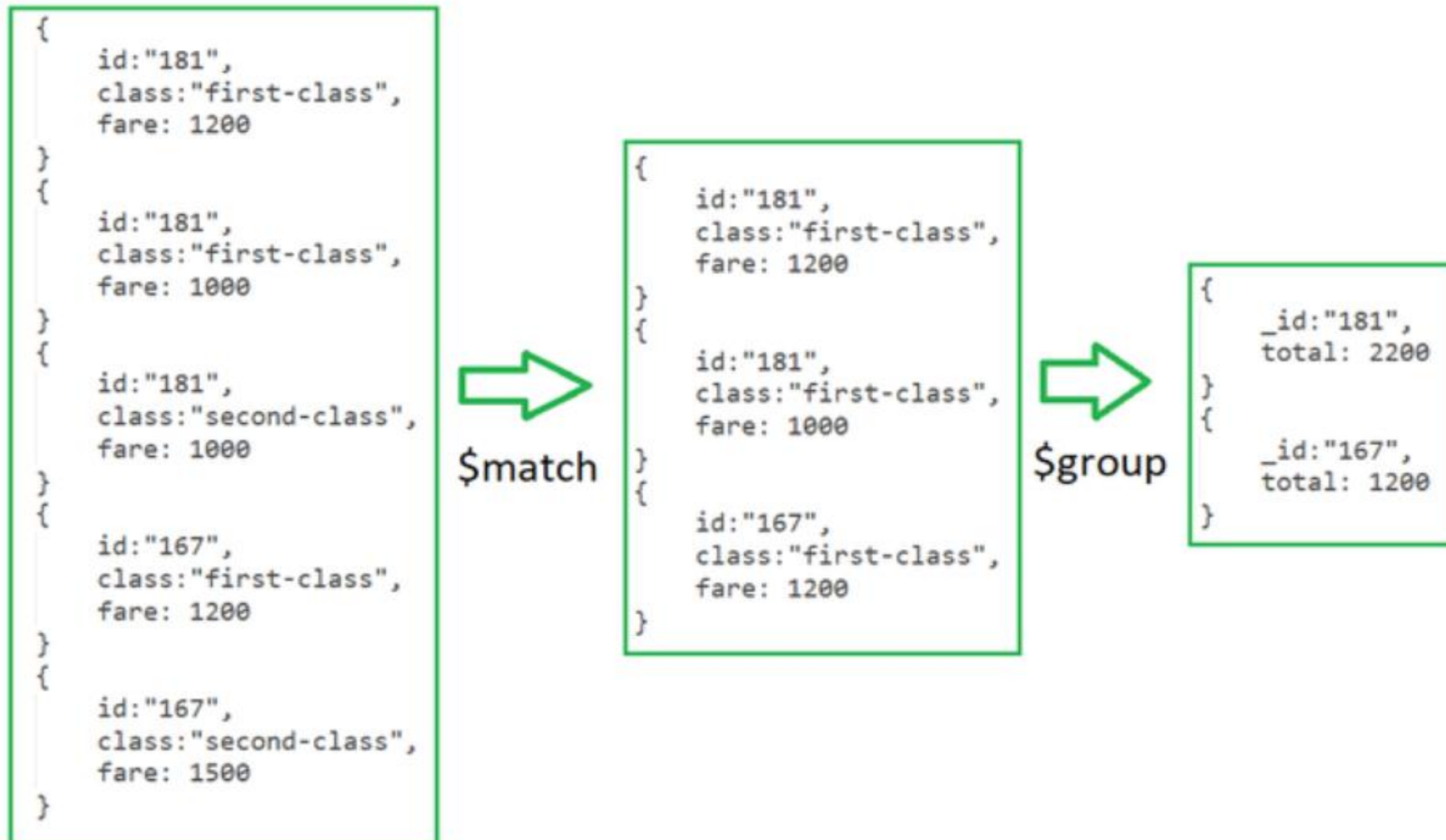
Ao utilizar as etapas de processamento disponíveis, você pode moldar seus dados de acordo com suas necessidades analíticas, abrindo portas para uma análise mais profunda e informada.

# Aggregate Framework



```
db.train.aggregate( [
  {$match:{class:"first-class"}},
  {$group:{_id:"id",total:{$sum:"$fare"}}}
] )
```

} pipeline stages



# \$group



O **\$group** separa os documentos em grupos de acordo com uma "chave de grupo".

A saída é um documento para cada chave de grupo exclusiva.

Uma chave de grupo geralmente é um campo ou grupo de campos.

A chave de grupo também pode ser o resultado de uma expressão.

Use o campo **\_id** no estágio de pipeline **\$group** para definir a chave do grupo.

Query:


```
aggregate( $group: { _id: "$campo" } )
```

# \$group



Vamos usar o aggregate e o **\$group** para agrupar todos os estados civis em clientes:

```
techBank> db.clientes.aggregate({$group:{_id:"$status_civil"}})
[
  { _id: 'Viúvo(a)' },
  { _id: 'Casado(a)' },
  { _id: 'Separado(a)' },
  { _id: 'Solteiro(a)' },
  { _id: 'Divorciado(a)' }
]
```



Isso não tem tanta utilidade, certo? Mas calma, lembra que uma agregação pode receber outras funções?

# \$group + \$count



Agora, vamos adicionar o operador **\$count** para contar quantos cada um aparece.

```
techBank> db.clientes.aggregate({$group:{_id:"$status_civil","Quantidade":{$count:{}}}})
[
  { _id: 'Solteiro(a)', Quantidade: 47 },
  { _id: 'Separado(a)', Quantidade: 39 },
  { _id: 'Viúvo(a)', Quantidade: 60 },
  { _id: 'Divorciado(a)', Quantidade: 51 },
  { _id: 'Casado(a)', Quantidade: 53 }
]
```

Query:

```
aggregate( {$group: {_id: "$campo", "alias":{$count:{}}}} )
```

# \$group + \$count

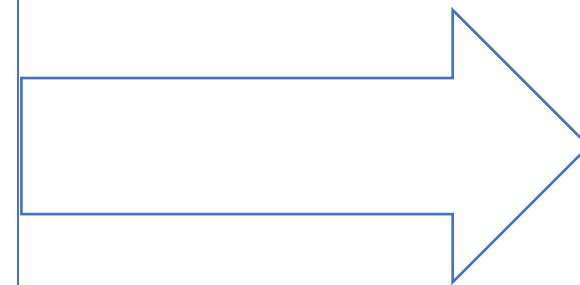


Vamos entender como foi que o mongo fez essa operação de agregação com um agrupamento de dados.

1

Agrupar os valores do campo status\_civil, o que vai corresponder ao nosso \_id

```
{ $group: { _id: $status_civil }
```



```
[  
  { _id: 'Viúvo(a)' },  
  { _id: 'Casado(a)' },  
  { _id: 'Separado(a)' },  
  { _id: 'Solteiro(a)' },  
  { _id: 'Divorciado(a)' }  
]
```

Query:

```
aggregate( { $group: { _id: "$campo", "alias": { $count: {} } } } )
```

# \$group + \$count

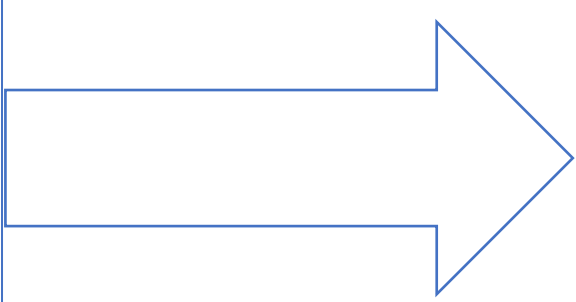


Vamos entender como foi que o mongo fez essa operação de agregação com um agrupamento de dados.

2

Contar quantos cada `_id` aparece na collection e colocar o resultado em um novo campo chamado “Quantidade”

“Quantidade”:{`$count`:{}}



```
[
  { _id: 'Solteiro(a)', Quantidade: 47 },
  { _id: 'Separado(a)', Quantidade: 39 },
  { _id: 'Viúvo(a)', Quantidade: 60 },
  { _id: 'Divorciado(a)', Quantidade: 51 },
  { _id: 'Casado(a)', Quantidade: 53 }
]
```

Query:

```
aggregate( {$group: {_id: “$campo”, “alias”:{$count:{}}}} )
```



# Como seria no SQL?

**MongoDB:**

```
db.clientes.aggregate({$group:{_id:"$status_civil","Quantidade":{$count:{}}}})
```

**SQL:**

```
SELECT status_civil, COUNT(*) AS Quantidade FROM clientes GROUP BY status_civil;
```





# Invertendo o processo!

Observe a query abaixo e diga o que ela está investigando?

```
db.vendas.aggregate( [ {  
  $group: {  
    _id : "$id_cliente",  
    quantidade: { $count:{} }  } } ] )
```

Cada documento da coleção contém informações como o ID do cliente, o valor da compra e a data da transação

**Resposta: A quantidade de vendas por clientes!**

# \$sum



Quanto cada tipo de contas tem de saldo no banco?  
Use o operador **\$sum** no campo **valor** em um **\$group**.

```
techBank> db.contas.aggregate({ $group: { _id: "$tipo", "Total em dinheiro": { $sum: "$valor" } } })  
[  
  { _id: 'Conta poupança', 'Total em dinheiro': 499625.37 },  
  { _id: 'Conta salário', 'Total em dinheiro': 391491.52999999997 },  
  { _id: 'Conta corrente', 'Total em dinheiro': 449840.01 }  
]
```

Mas e se for pedido para mostrar em ordem do tipo de conta que tem menos para a que tem mais dinheiro?

# \$sort



Mas e se for pedido para mostrar em ordem da conta que tem menos para a que tem mais dinheiro?

```
techBank> db.contas.aggregate([
...   { $group: { _id: "$tipo", "Total em dinheiro": { $sum: "$valor" } } },
...   { $sort: { "Total em dinheiro": 1 } }
... ])
[
  { _id: 'Conta salário', 'Total em dinheiro': 391491.52999999997 },
  { _id: 'Conta corrente', 'Total em dinheiro': 449840.01 },
  { _id: 'Conta poupança', 'Total em dinheiro': 499625.37 }
]
```

Percebeu que o **\$sort** é feito no ALIAS e não no campo, certo? Mas funciona colocar qualquer um! Ainda assim, por boas práticas e para evitar confusões, é recomendado usar o nome do campo resultante da agregação na etapa, que neste caso é "Total em dinheiro".

# Desafio



Qual agencia tem mais contas no banco?

```
techBank> db.contas.aggregate([{$group:{_id:"$agencia", "qnt":{$count:{}}}}, {$sort:{"qnt":-1}}])
[
  { _id: 1545, qnt: 66 }, { _id: 5819, qnt: 2 },
  { _id: 2140, qnt: 2 }, { _id: 498, qnt: 2 },
  { _id: 6383, qnt: 2 }, { _id: 936, qnt: 2 },
  { _id: 5294, qnt: 2 }, { _id: 1066, qnt: 2 },
  { _id: 6967, qnt: 2 }, { _id: 5568, qnt: 2 },
  { _id: 1198, qnt: 1 }, { _id: 4713, qnt: 1 },
  { _id: 3441, qnt: 1 }, { _id: 1684, qnt: 1 },
  { _id: 1760, qnt: 1 }, { _id: 2863, qnt: 1 },
  { _id: 2229, qnt: 1 }, { _id: 2239, qnt: 1 },
  { _id: 966, qnt: 1 }, { _id: 2151, qnt: 1 }
]
```

# \$match



Um operador de filtragem, ou seja, ele filtra os documentos para passar apenas os documentos que correspondem à(s) condição(ões) especificada(s) para o próximo estágio do pipeline.

Por exemplo, quais são os clientes da agencia **1066**?

```
techBank> db.contas.aggregate([{$match:{agencia:1066}}])
[
  {
    _id: 154,
    id_cliente: 145,
    numero_conta: '261090-6',
    agencia: 1066,
    tipo: 'Conta corrente',
    cpf: '170.029.883-60',
    valor: ''
  },
  {
    _id: 153,
    id_cliente: 145,
    numero_conta: '261090-6',
    agencia: 1066,
    tipo: 'Conta salário',
    cpf: '170.029.883-60',
    valor: 7846.66
  }
]
```

# \$match



Ok, a query com o **\$match** é igual ao find() então qual a vantagem?

```
techBank> db.contas.aggregate([{$match:{agencia:1066}}])
[
  {
    _id: 154,
    id_cliente: 145,
    numero_conta: '261090-6',
    agencia: 1066,
    tipo: 'Conta corrente',
    cpf: '170.029.883-60',
    valor: ''
  },
  {
    _id: 153,
    id_cliente: 145,
    numero_conta: '261090-6',
    agencia: 1066,
    tipo: 'Conta salário',
    cpf: '170.029.883-60',
    valor: 7846.66
  }
]
```

```
techBank> db.contas.find({agencia:1066})
[
  {
    _id: 154,
    id_cliente: 145,
    numero_conta: '261090-6',
    agencia: 1066,
    tipo: 'Conta corrente',
    cpf: '170.029.883-60',
    valor: ''
  },
  {
    _id: 153,
    id_cliente: 145,
    numero_conta: '261090-6',
    agencia: 1066,
    tipo: 'Conta salário',
    cpf: '170.029.883-60',
    valor: 7846.66
  }
]
```

# \$match



Ok, a query com o **\$match** é igual ao `find()` então qual a vantagem?

Se você só precisa filtrar e recuperar documentos sem transformações → use **find()**, pois é mais simples e direto.

Se você vai processar dados dentro do aggregation framework → use **\$match** para reduzir o número de documentos logo no início e melhorar a performance.



# \$match e \$count

Se quiser saber quais são as contas do tipo “Conta salário” que tem mais que 8500 no campo valor?

```
techBank> db.contas.find({tipo:"Conta salário", valor:{$gt:8500}}).count()  
12
```

Tente pensar em ter esse resultado, porem agora usando o operador **\$match** e o **\$count**.

```
techBank> db.contas.aggregate({$match: {tipo:"Conta salário", valor:{$gt:8500}}  
... ,{$count:"Total de contas salários com mais de R$8500 de saldo"}  
... })  
[ { 'Total de contas salários com mais de R$8500 de saldo': 12 } ]
```

Nesse caso o find() é mais indicado. Busca direta!





# \$match e \$group

Tem até uma maneira mais correta e mais difícil, que é. Imagine que você quer na resposta já o que é buscado, então devemos usar o **\$group** também.

```
techBank> db.contas.aggregate([
...  {$match:{tipo:"Conta salário", valor:{$gt:8500}}},
...  {$group:{_id:"$tipo", "Quantidade":{$count:{}}}}
... ])
[ { _id: 'Conta salário', Quantidade: 12 } ]
```

Legal né?





# \$match e \$group

Mas se a gente quiser saber de cada conta, quantas tem mais do que 8000 no valor?

```
techBank> db.contas.aggregate([
...   {$match:{valor:{$gt:8000}}},
...   {$group:{_id:"$tipo", "Quantidade":{$count:{}}}}
... ])
[
  { _id: 'Conta salário', Quantidade: 16 },
  { _id: 'Conta poupança', Quantidade: 26 },
  { _id: 'Conta corrente', Quantidade: 19 }
]
```

Agora sim, a coisa começou a fazer sentido!



# Aggregate (\$match, \$group, \$sort)

Mas voltando.... e se agora for pedido para mostrar em ordem da conta que tem menos para a que tem mais dinheiro, mas apenas da agencia 1545?

```
techBank> db.contas.aggregate([
...  {$match:{agencia:1545}},
...  {$group:{_id:"$tipo", "Total":{$sum:"$valor"}}},
...  {$sort:{"Total":1}}
... ])
[
  { _id: 'Conta salário', Total: 92693.91 },
  { _id: 'Conta poupança', Total: 111758.59 },
  { _id: 'Conta corrente', Total: 149299.81 }
]
```

# \$limit

O `.limit()` é usado com uma estrutura `find()`, mas também existe o operador de agregação **\$limit**.

Observe que os retornos ao lado são iguais.

```
techBank> db.contas.find().limit(1)
[
  {
    _id: 120,
    id_cliente: 112,
    numero_conta: '27984233-1',
    agencia: 2140,
    tipo: 'Conta salário',
    cpf: '997.041.851-38',
    valor: ''
  }
]
techBank> db.contas.aggregate({$limit:1})
[
  {
    _id: 120,
    id_cliente: 112,
    numero_conta: '27984233-1',
    agencia: 2140,
    tipo: 'Conta salário',
    cpf: '997.041.851-38',
    valor: ''
  }
]
```



# \$sort e \$limit



Lembra que teve uma query onde você pegou o cliente mais velho (idade)? Você consegue pensar em uma agregação usando os operadores **\$sort** e **\$limit** para extrair esse mesmo resultado?

```
techBank> db.clientes.aggregate({$sort:{data_nascimento:1}},{$limit:1})
[
  {
    _id: 45,
    nome: 'Alice Márcia Pereira',
    cpf: '686.742.959-91',
    data_nascimento: ISODate("1944-05-13T21:00:00.000Z"),
    genero: 'Feminino',
    profissao: 'Mecânico de automóveis',
    status_civil: 'Viúvo(a)',
    seguros: [ 'seguro de vida', 'seguro para carro', 'seguro para casa' ]
  }
]
```



# Aggregate (\$group, \$sort, \$limit)

Se a pergunta for, qual a agência que tem mais saldo em suas contas?

```
techBank> db.contas.aggregate(  
...  {$group: {_id: "$agencia", "Total": {$sum: "$valor"}}},  
...  {$sort: {"Total": -1}},  
...  {$limit: 1} )  
[ { _id: 1545, Total: 353752.31 } ]
```