

## 1. 聚合操作

聚合操作允许用户处理多个文档并返回计算结果。聚合操作组值来自多个文档，可以对分组数据执行各种操作以返回单个结果。

聚合操作包含三类：单一作用聚合、聚合管道、MapReduce。

- 单一作用聚合：提供了对常见聚合过程的简单访问，操作都从单个集合聚合文档。MongoDB提供 `db.collection.estimatedDocumentCount()`, `db.collection.countDocument()`, `db.collection.distinct()` 这类单一作用的聚合函数。所有这些操作都聚合来自单个集合的文档。虽然这些操作提供了对公共聚合过程的简单访问，但它们缺乏聚合管道和map-Reduce的灵活性和功能。
- 聚合管道是一个数据聚合的框架，模型基于数据处理流水线的概念。文档进入多级管道，将文档转换为聚合结果。
- MapReduce操作具有两个阶段：处理每个文档并向每个输入文档发射一个或多个对象的map阶段，以及reduce组合map操作的输出阶段。从MongoDB 5.0开始，map-reduce操作已被弃用。聚合管道比映射-reduce操作提供更好的性能和可用性。

MongoDB 6.0在原有聚合功能的基础上，推出了如下新特性以及优化项：

- 分片集群实例支持`$lookup`和`$graphLookup`。
- 改进`$lookup`对JOINS的支持。
- 改进`$graphLookup`对图遍历的支持。
- 提升`$lookup`性能，部分场景中性能提升可达百倍。

### 1.1 聚合管道

MongoDB 聚合框架（Aggregation Framework）是一个计算框架，它可以：

- 作用在一个或几个集合上；
- 对集合中的数据进行一系列运算；
- 将这些数据转化为期望的形式；

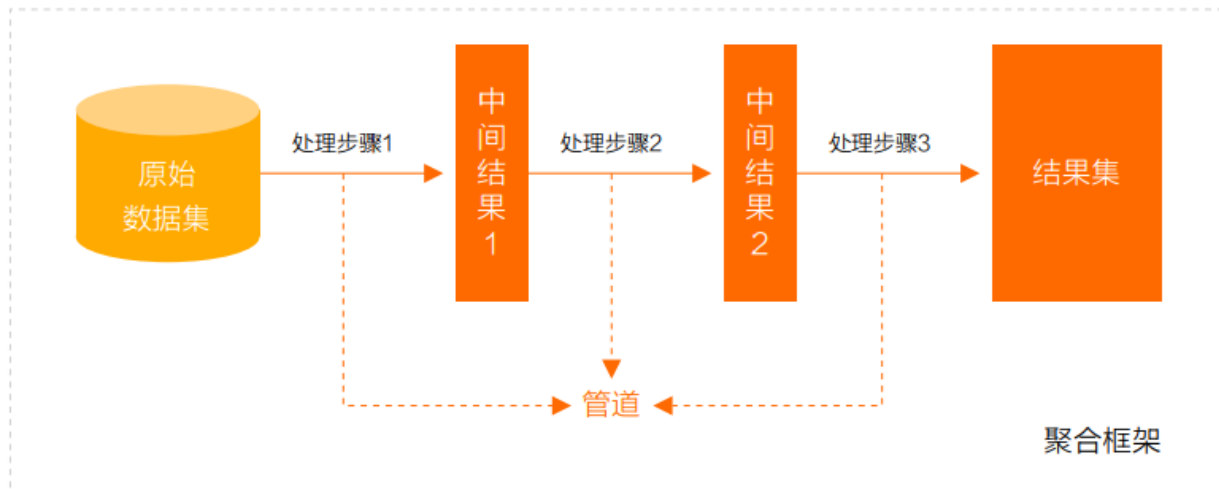
从效果而言，聚合框架相当于 SQL 查询中的GROUP BY、LEFT OUTER JOIN、AS等。

#### 管道（Pipeline）和阶段（Stage）

整个聚合运算过程称为管道（Pipeline），它是由多个阶段（Stage）组成的，每个管道：

- 接受一系列文档（原始数据）；
- 每个阶段对这些文档进行一系列运算；
- 结果文档输出给下一个阶段；

通过将多个操作符组合到聚合管道中，用户可以构建出足够复杂的数据处理管道以提取数据并进行分析。



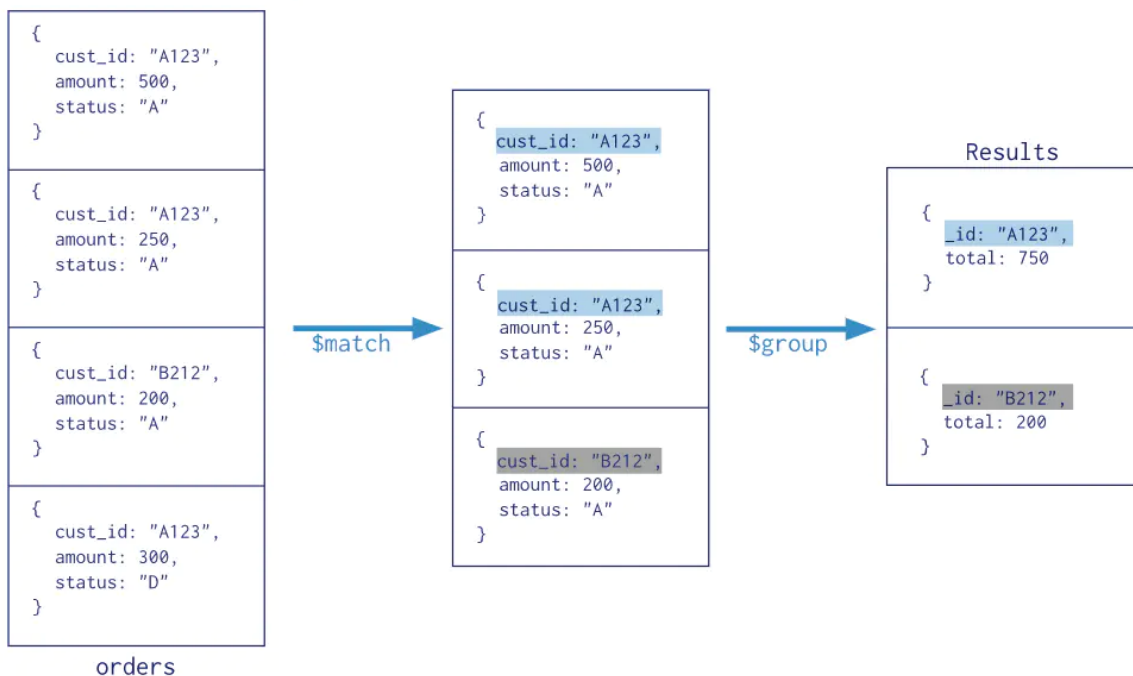
## 聚合管道操作语法

```
1 pipeline = [$stage1, $stage2, ...$stageN];
2 db.collection.aggregate(pipeline, {options})
```

- pipelines 一组数据聚合阶段。除\$out、\$Merge和\$geonear阶段之外，每个阶段都可以在管道中出现多次。
- options 可选，聚合操作的其他参数。包含：查询计划、是否使用临时文件、游标、最大操作时间、读写策略、强制索引等等

Collection

```
db.orders.aggregate( [
  $match stage → { $match: { status: "A" } },
  $group stage → { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }
] )
```



## 常用的管道聚合阶段

聚合管道包含非常丰富的聚合阶段，下面是最常用的聚合阶段

阶段运算符	描述	SQL等价运算符
\$match	筛选条件	WHERE
\$project	投影	AS
\$lookup	左外连接	LEFT OUTER JOIN
\$sort	排序	ORDER BY
\$group	分组	GROUP BY
\$skip/\$limit	分页	
\$unwind	展开数组	
\$graphLookup	图搜索	
\$facet/\$bucket	分面搜索	

文档: [Aggregation Pipeline Stages — MongoDB Manual](#)

## 聚合表达式

获取字段信息

- 1 `$<field>` : 用 `$` 指示字段路径
- 2 `$<field>.<sub field>` : 使用 `$` 和 `.` 来指示内嵌文档的路径

## 常量表达式

- 1 `$literal :<value>` : 指示常量 `<value>`

## 系统变量表达式

- 1 `$$<variable>` 使用 `$$` 指示系统变量

2 **\$\$CURRENT** 指示管道中当前操作的文档

## 数据准备

### 准备数据集，执行脚本

```
1 var tags = ["nosql","mongodb","document","developer","popular"];
2 var types = ["technology","sociality","travel","novel","literature"];
3 var books=[];
4 for(var i=0;i<50;i++){
5     var typeIdx = Math.floor(Math.random()*types.length);
6     var tagIdx = Math.floor(Math.random()*tags.length);
7     var tagIdx2 = Math.floor(Math.random()*tags.length);
8     var favCount = Math.floor(Math.random()*100);
9     var username = "xx00"+Math.floor(Math.random()*10);
10    var age = 20 + Math.floor(Math.random()*15);
11    var book = {
12        title: "book-"+i,
13        type: types[typeIdx],
14        tag: [tags[tagIdx],tags[tagIdx2]],
15        favCount: favCount,
16        author: {name:username,age:age}
17    };
18    books.push(book)
19 }
20 db.books.insertMany(books);
```

## \$project

**投影操作**，将原始字段投影成指定名称，如将集合中的 title 投影成 name

```
1 db.books.aggregate([{$project:{name:"$title"}}])
```

**\$project** 可以灵活控制输出文档的格式，也可以剔除不需要的字段

```
1 db.books.aggregate([{$project:{name:"$title",_id:0,type:1,author:1}}])
```

## 从嵌套文档中排除字段

```
1 db.books.aggregate([
2     {$project:{name:"$title",_id:0,type:1,"author.name":1}}
3 ])
4 或者
5 db.books.aggregate([
6     {$project:{name:"$title",_id:0,type:1,author:{name:1}}}
7 ])
```

## \$match

**\$match**用于对文档进行筛选，之后可以在得到的文档子集上做聚合，\$match可以使用除了地理空间之外的所有常规查询操作符，在实际应用中尽可能将\$match放在管道的前面位置。这样有两个好处：一是可以快速将不需要的文档过滤掉，以减少管道的工作量；二是如果再投射和分组之前执行\$match，查询可以使用索引。

```
1 db.books.aggregate([{$match:{type:"technology"}}])
```

筛选管道操作和其他管道操作配合时候时，尽量放到开始阶段，这样可以减少后续管道操作符要操作的文档数，提升效率

```
1 db.books.aggregate([
2     {$match:{type:"technology"}},
3     {$project:{name:"$title",_id:0,type:1,author:{name:1}}}
4 ])
```

## \$count

计数并返回与查询匹配的结果数

```

1 db.books.aggregate([
2     {$match:{type:"technology"}},
3     {$count: "type_count"}
4 ])

```

\$match阶段筛选出type匹配technology的文档，并传到下一阶段；

\$count阶段返回聚合管道中剩余文档的计数，并将该值分配给type\_count

## \$group

按指定的表达式对文档进行分组，并将每个不同分组的文档输出到下一个阶段。输出文档包含一个\_id字段，该字段按键包含不同的组。

输出文档还可以包含计算字段，该字段保存由\$group的\_id字段分组的一些accumulator表达式的值。

\$group不会输出具体的文档而只是统计信息。

```

1 { $group: { _id: <expression>, <field1>: { <accumulator1> : <expression1> }, ... } }

```

- \_id字段是必填的;但是，可以指定\_id值为null来为整个输入文档计算累计值。
- 剩余的计算字段是可选的，并使用<accumulator>运算符进行计算。
- \_id和<accumulator>表达式可以接受任何有效的[表达式](#)。

## accumulator操作符

名称	描述	类比sql
\$avg	计算均值	avg
\$first	返回每组第一个文档，如果有排序，按照排序，如果没有按照默认的存储的顺序的第一个文档。	limit 0,1
\$last	返回每组最后一个文档，如果有排序，按照排序，如果没有按照默认的存储的顺序的最后个文档。	-
\$max	根据分组，获取集合中所有文档对应值得最大值。	max
\$min	根据分组，获取集合中所有文档对应值得最小值。	min

<b>\$push</b>	将指定的表达式的值添加到一个数组中。	-
\$addToSet	将表达式的值添加到一个集合中（无重复值，无序）。	-
<b>\$sum</b>	计算总和	sum
\$stdDevPop	返回输入值的总体标准偏差（population standard deviation）	-
\$stdDevSamp	返回输入值的样本标准偏差（the sample standard deviation）	-

**\$group**阶段的内存限制为100M。默认情况下，如果stage超过此限制，**\$group**将产生错误。但是，要允许处理大型数据集，请将allowDiskUse选项设置为true以启用**\$group**操作以写入临时文件。

### book的数量，收藏总数和平均值

```
1 db.books.aggregate([
2     {$group: {_id: null, count: {$sum: 1}, pop: {$sum: "$favCount"}, avg: {$avg: "$favCount"}}}
3 ])
```

### 统计每个作者的book收藏总数

```
1 db.books.aggregate([
2     {$group: {_id: "$author.name", pop: {$sum: "$favCount"}}}
3 ])
```

### 统计每个作者的每本book的收藏数

```
1 db.books.aggregate([
2     {$group: {_id: {name: "$author.name", title: "$title"}, pop: {$sum: "$favCount"}}}
3 ])
```

### 每个作者的book的type合集

```
1 db.books.aggregate([
```

```
2     {$group: {_id: "$author.name", types: {$addToSet: "$type"}}}
3   ])
```

## \$unwind

可以将数组拆分为单独的文档

v3.2+支持如下语法:

```
1  {
2    $unwind:
3    {
4      #要指定字段路径, 在字段名称前加上$符并用引号括起来。
5      path: <field path>,
6      #可选, 一个新字段的名称用于存放元素的数组索引。该名称不能以$开头。
7      includeArrayIndex: <string>,
8      #可选, default :false, 若为true, 如果路径为空, 缺少或为空数组, 则$unwind输出文档
9      preserveNullAndEmptyArrays: <boolean>
10   } }
```

姓名为xx006的作者的book的tag数组拆分为多个文档

```
1  db.books.aggregate([
2    {$match: {"author.name": "xx006"}},
3    {$unwind: "$tag"}
4  ])
5
6  db.books.aggregate([
7    {$match: {"author.name": "xx006"}}
8  ])
```

每个作者的book的tag合集

```
1
2  db.books.aggregate([
3    {$unwind: "$tag"},
```



```
4     {$group: {_id: "$author.name", types: {$addToSet: "$tag"}}}
5   ]})
6
```

## 案例

### 示例数据

```
1 db.books.insert([
2   {
3     "title" : "book-51",
4     "type" : "technology",
5     "favCount" : 11,
6     "tag": [],
7     "author" : {
8       "name" : "fox",
9       "age" : 28
10    }
11  }, {
12    "title" : "book-52",
13    "type" : "technology",
14    "favCount" : 15,
15    "author" : {
16      "name" : "fox",
17      "age" : 28
18    }
19  }, {
20    "title" : "book-53",
21    "type" : "technology",
22    "tag" : [
23      "nosql",
24      "document"
25    ],
26    "favCount" : 20,
27    "author" : {
28      "name" : "fox",
29      "age" : 28
30    }
31  }] )
```

## 测试

```
1 # 使用includeArrayIndex选项来输出数组元素的数组索引
2 db.books.aggregate([
3     {$match:{"author.name":"fox"}},
4     {$unwind:{path:"$tag", includeArrayIndex: "arrayIndex"}}
5 ])
6 # 使用preserveNullAndEmptyArrays选项在输出中包含缺少size字段, null或空数组的文档
7 db.books.aggregate([
8     {$match:{"author.name":"fox"}},
9     {$unwind:{path:"$tag", preserveNullAndEmptyArrays: true}}
10 ])
```

## \$limit

限制传递到管道中下一阶段的文档数

```
1 db.books.aggregate([
2     {$limit : 5 }
3 ])
```

此操作仅返回管道传递给它的前5个文档。 \$limit对其传递的文档内容没有影响。

**注意：**当\$sort在管道中的\$limit之前立即出现时，\$sort操作只会在过程中维持前n个结果，其中n是指定的限制，而MongoDB只需要将n个项存储在内存中。

## \$skip

跳过进入stage的指定数量的文档，并将其余文档传递到管道中的下一个阶段

```
1 db.books.aggregate([
2     {$skip : 5 }
3 ])
```

此操作将跳过管道传递给它的前5个文档。 `$skip`对沿着管道传递的文档的内容没有影响。

## `$sort`

对所有输入文档进行排序，并按排序顺序将它们返回到管道。

语法：

```
1 { $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
```

要对字段进行排序，请将排序顺序设置为1或-1，以分别指定升序或降序排序，如下例所示：

```
1 db.books.aggregate([
2     {$sort : {favCount:-1,"author.age":1}}
3 ])
```

## `$lookup`

Mongodb 3.2版本新增，主要用来实现多表关联查询，相当关系型数据库中多表关联查询。每个输入待处理的文档，经过`$lookup`阶段的处理，输出的新文档中会包含一个新生成的数组（可根据需要命名新key）。数组列存放的数据是来自被Join集合的适配文档，如果没有，集合为空（即为[]）

语法：

```
1 db.collection.aggregate([
2     $lookup: {
3         from: "<collection to join>",
4         localField: "<field from the input documents>",
5         foreignField: "<field from the documents of the from collection>",
6         as: "<output array field>"
7     }
8 ])
```

from

同一个数据库下等待被Join的集合。

localField	源集合中的match值，如果输入的集合中，某文档没有 localField 这个Key (Field)，在处理的过程中，会默认为此文档含有 localField: null的键值对。
foreignField	待Join的集合的match值，如果待Join的集合中，文档没有foreignField 值，在处理的过程中，会默认为此文档含有 foreignField: null的键值对。
as	为输出文档的新增值命名。如果输入的集合中已存在该值，则会覆盖掉

注意：null = null 此为真

其语法功能类似于下面的伪SQL语句：

```

1  SELECT *, <output array field>
2  FROM collection
3  WHERE <output array field> IN (SELECT *
4                                FROM <collection to join>
5                                WHERE <foreignField>= <collection.localField>);

```

## 案例

### 数据准备

```

1  db.customer.insert({customerCode:1,name:"customer1",phone:"13112345678",address:"test1"
  })
2  db.customer.insert({customerCode:2,name:"customer2",phone:"13112345679",address:"test2"
  })
3
4  db.order.insert({orderId:1,orderCode:"order001",customerCode:1,price:200})
5  db.order.insert({orderId:2,orderCode:"order002",customerCode:2,price:400})
6
7  db.orderItem.insert({itemId:1,productName:"apples",qutity:2,orderId:1})
8  db.orderItem.insert({itemId:2,productName:"oranges",qutity:2,orderId:1})
9  db.orderItem.insert({itemId:3,productName:"mangoes",qutity:2,orderId:1})
10 db.orderItem.insert({itemId:4,productName:"apples",qutity:2,orderId:2})

```

```
11 db.orderItem.insert({itemId:5,productName:"oranges",qutity:2,orderId:2})
12 db.orderItem.insert({itemId:6,productName:"mangoes",qutity:2,orderId:2})
```

## 关联查询

```
1  db.customer.aggregate([
2    {$lookup: {
3      from: "order",
4      localField: "customerCode",
5      foreignField: "customerCode",
6      as: "customerOrder"
7    }
8  }
9  ])
10
11 db.order.aggregate([
12   {$lookup: {
13     from: "customer",
14     localField: "customerCode",
15     foreignField: "customerCode",
16     as: "curstomer"
17   }
18
19 },
20   {$lookup: {
21     from: "orderItem",
22     localField: "orderId",
23     foreignField: "orderId",
24     as: "orderItem"
25   }
26 }
27 ])
```

## 聚合操作案例1

## 统计每个分类的book文档数量

```
1 db.books.aggregate([
2     {$group: {_id: "$type", total: {$sum: 1}}},
3     {$sort: {total: -1}}
4 ])
5
```

## 标签的热度排行，标签的热度则按其关联book文档的收藏数（favCount）来计算

```
1 db.books.aggregate([
2     {$match: {favCount: {$gt: 0}}},
3     {$unwind: "$tag"},
4     {$group: {_id: "$tag", total: {$sum: "$favCount"}}},
5     {$sort: {total: -1}}
6 ])
```


1. \$match阶段：用于过滤favCount=0的文档。
2. \$unwind阶段：用于将标签数组进行展开，这样一个包含3个标签的文档会被拆解为3个条目。
3. \$group阶段：对拆解后的文档进行分组计算，\$sum: "\$favCount"表示按favCount字段进行累加。
4. \$sort阶段：接收分组计算的输出，按total得分进行排序。

## 统计book文档收藏数[0,10),[10,60),[60,80),[80,100),[100,+∞)

```
1 db.books.aggregate([
2     $bucket: {
3         groupBy: "$favCount",
4         boundaries: [0, 10, 60, 80, 100],
5         default: "other",
6         output: {"count": {$sum: 1}}
7     }
8 ])
```

## 聚合操作案例2

导入邮政编码数据集:<https://media.mongodb.org/zipcodes.json>

 zipcodes.json  
3.03MB

### 使用mongoimport工具导入数据

```
1 mongoimport -h 192.168.65.174 -d test -u fox -p fox --authenticationDatabase=admin -c  
  zipcodes --file D:\ProgramData\mongodb\import\zipcodes.json
```

`h,--host` : 代表远程连接的数据库地址, 默认连接本地Mongo数据库;  
`--port`: 代表远程连接的数据库的端口, 默认连接的远程端口27017;  
`-u,--username`: 代表连接远程数据库的账号, 如果设置数据库的认证, 需要指定用户账号;  
`-p,--password`: 代表连接数据库的账号对应的密码;  
`-d,--db`: 代表连接的数据库;  
`-c,--collection`: 代表连接数据库中的集合;  
`-f, --fields`: 代表导入集合中的字段;  
`--type`: 代表导入的文件类型, 包括csv和json,tsv文件, 默认json格式;  
`--file`: 导入的文件名称  
`--headerline`: 导入csv文件时, 指明第一行是列名, 不需要导入;

```
C:\WINDOWS\system32>mongoimport -h 192.168.65.174 -d test -u fox -p fox --authenticationDatabase=admin -c zipcodes --file D:\ProgramData\mongodb\import\zipcodes.json
2021-12-29T15:43:11.315+0800 connected to: 192.168.65.174
2021-12-29T15:43:11.845+0800 imported 29353 documents
```

### 返回人口超过1000万的州

```
1 db.zipcodes.aggregate( [  
2   { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },  
3   { $match: { totalPop: { $gte: 10*1000*1000 } } }  
4 ] )
```

这个聚合操作的等价SQL是:

```
1 SELECT state, SUM(pop) AS totalPop  
2 FROM zipcodes
```

```
3 GROUP BY state
4 HAVING totalPop >= (10*1000*1000)
```

## 返回各州平均城市人口

```
1 db.zips.aggregate( [
2   { $group: { _id: { state: "$state", city: "$city" }, cityPop: { $sum: "$pop" } } },
3   { $group: { _id: "$_id.state", avgCityPop: { $avg: "$cityPop" } } }
4 ] )
5
```

## 按州返回最大和最小的城市

```
1 db.zips.aggregate( [
2   { $group:
3     {
4       _id: { state: "$state", city: "$city" },
5       pop: { $sum: "$pop" }
6     }
7   },
8   { $sort: { pop: 1 } },
9   { $group:
10    {
11      _id : "$_id.state",
12      biggestCity: { $last: "$_id.city" },
13      biggestPop: { $last: "$pop" },
14      smallestCity: { $first: "$_id.city" },
15      smallestPop: { $first: "$pop" }
16    }
17  },
18  { $project:
19    { _id: 0,
20      state: "$_id",
```



```
21     biggestCity: { name: "$biggestCity", pop: "$biggestPop" },
22     smallestCity: { name: "$smallestCity", pop: "$smallestPop" }
23   }
24 }
25 ] )
```

## 1.2 聚合优化

<https://www.mongodb.com/docs/manual/core/aggregation-pipeline-optimization/>

### 聚合优化的三大目标:

- 尽可能利用索引完成搜索和排序
- 尽早尽多减少数据量
- 尽可能减少执行步骤



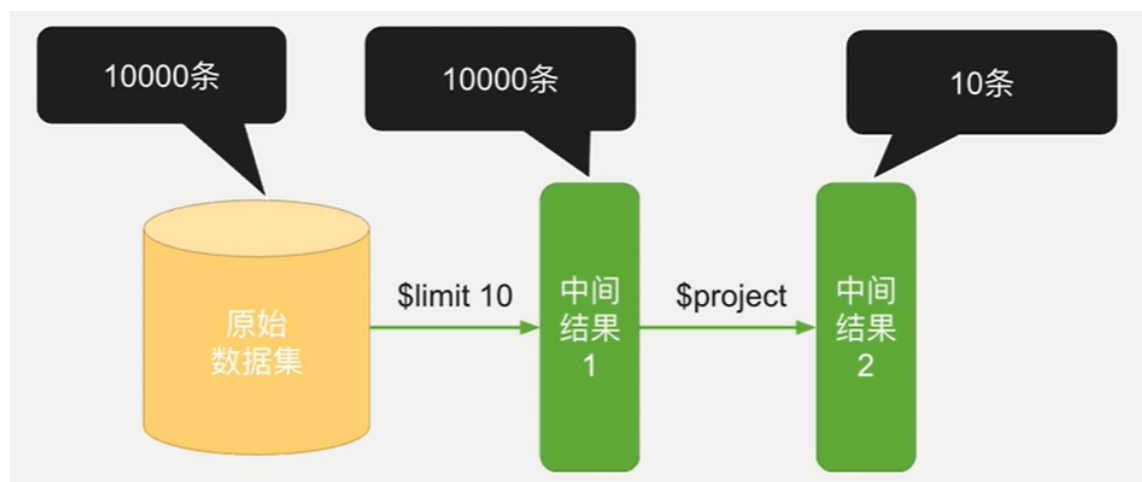
## 执行顺序

### \$match/\$sort vs \$project/\$addFields

为了使查询能够命中索引，\$match/\$sort步骤需要在最前面，该原则适用于MongoDB <= 3.4。  
MongoDB 3.6开始具备一定的自动优化能力。

### \$project + \$skip/\$limit

\$skip/\$limit应该尽可能放在\$project之前，减少\$project的工作量。3.6开始自动完成这个优化。



## 内存排序

在没有索引支持的情况下，MongoDB最多只支持使用100MB内存进行排序。假设总共可用内存为16GB，一个请求最多可以使用100MB内存排序，总共可以有 $16000 / 100 = 160$ 个请求同时执行。

内存排序消耗的不仅是内存，还有大量CPU

### 方案一：\$sort + \$limit

只排Top N，只要N条记录总和不超过100MB即可

### 方案二：{allowDiskUse: true}

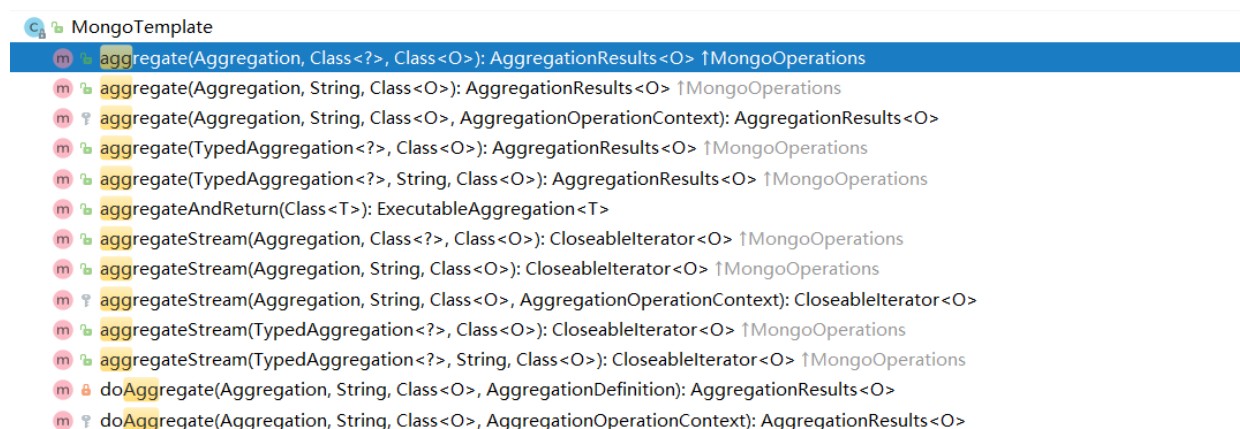
使用磁盘作为交换空间完成全量，超出100MB部分与磁盘交换排序

### 方案三：索引排序

使用索引完成排序，没有内存限制

## 1.3 整合Springboot进行聚合操作

MongoTemplate提供了aggregate方法来实现对数据的聚合操作。



基于聚合管道mongodb提供的可操作的内容：

支持的操作	java接口	说明
\$project	Aggregation.project	修改输入文档的结构。
\$match	Aggregation.match	用于过滤数据
\$limit	Aggregation.limit	用来限制MongoDB聚合管道返回的文档数
\$skip	Aggregation.skip	在聚合管道中跳过指定数量的文档
\$unwind	Aggregation.unwind	将文档中的某一个数组类型字段拆分成多条
\$group	Aggregation.group	将集中的文档分组，可用于统计结果
\$sort	Aggregation.sort	将输入文档排序后输出
\$geoNear	Aggregation.geoNear	输出接近某一地理位置的有序文档

基于聚合操作Aggregation.group，mongodb提供可选的表达式

聚合表达式	java接口	说明
\$sum	Aggregation.group().sum("field").as("sum")	求和
\$avg	Aggregation.group().avg("field").as("avg")	求平均
\$min	Aggregation.group().min("field").as("min")	获取集中所有文档对应值得最小值
\$max	Aggregation.group().max("field").as("max")	获取集中所有文档对应值得最大值
\$push	Aggregation.group().push("field").as("push")	在结果文档中插入值到一个数组中
\$addToSet	Aggregation.group().addToSet("field").as("addToSet")	在结果文档中插入值到一个数组中，但不创建副本
\$first	Aggregation.group().first("field").as("first")	根据资源文档的排序获取第一个文档数据
\$last	Aggregation.group().last("field").as("last")	根据资源文档的排序获取最后一个文档数据

示例：以聚合操作案例2为例

返回人口超过1000万的州

```
1 db.zips.aggregate( [
2   { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
3   { $match: { totalPop: { $gt: 10*1000*1000 } } }
4 ] )
```

java实现

```
1 @Test
2 public void test(){
3     //$group
4     GroupOperation groupOperation =
5     Aggregation.group("state").sum("pop").as("totalPop");
```

```

6    //$match
7    MatchOperation matchOperation = Aggregation.match(
8        Criteria.where("totalPop").gte(10*1000*1000));
9
10   // 按顺序组合每一个聚合步骤
11   TypedAggregation<Zips> typedAggregation = Aggregation.newAggregation(Zips.class,
12       groupOperation, matchOperation);
13
14   //执行聚合操作,如果不使用 Map, 也可以使用自定义的实体类来接收数据
15   AggregationResults<Map> aggregationResults =
16       mongoTemplate.aggregate(typedAggregation, Map.class);
17   // 取出最终结果
18   List<Map> mappedResults = aggregationResults.getMappedResults();
19   for(Map map:mappedResults){
20       System.out.println(map);
21   }
22 }

```

## 返回各州平均城市人口

```

1  db.zips.aggregate( [
2    { $group: { _id: { state: "$state", city: "$city" }, cityPop: { $sum: "$pop" } } },
3    { $group: { _id: "$_id.state", avgCityPop: { $avg: "$cityPop" } } },
4    { $sort:{avgCityPop:-1}}
5  ] )

```

## java实现

```

1  @Test
2  public void test2(){
3      //$group
4      GroupOperation groupOperation =
5          Aggregation.group("state","city").sum("pop").as("cityPop");
6      //$group
7      GroupOperation groupOperation2 =
8          Aggregation.group("_id.state").avg("cityPop").as("avgCityPop");

```

```

7      //$sort
8      SortOperation sortOperation = Aggregation.sort(Sort.Direction.DESC,"avgCityPop");
9
10     // 按顺序组合每一个聚合步骤
11     TypedAggregation<Zips> typedAggregation = Aggregation.newAggregation(Zips.class,
12         groupOperation, groupOperation2,sortOperation);
13
14     //执行聚合操作,如果不使用 Map, 也可以使用自定义的实体类来接收数据
15     AggregationResults<Map> aggregationResults =
16         mongoTemplate.aggregate(typedAggregation, Map.class);
17     // 取出最终结果
18     List<Map> mappedResults = aggregationResults.getMappedResults();
19     for(Map map:mappedResults){
20         System.out.println(map);
21     }

```

## 按州返回最大和最小的城市

```

1  db.zips.aggregate( [
2      { $group:
3          {
4              _id: { state: "$state", city: "$city" },
5              pop: { $sum: "$pop" }
6          }
7      },
8      { $sort: { pop: 1 } },
9      { $group:
10         {
11             _id : "$_id.state",
12             biggestCity: { $last: "$_id.city" },
13             biggestPop: { $last: "$pop" },
14             smallestCity: { $first: "$_id.city" },
15             smallestPop: { $first: "$pop" }
16         }
17     },

```

```

18 { $project:
19   { _id: 0,
20     state: "$_id",
21     biggestCity: { name: "$biggestCity", pop: "$biggestPop" },
22     smallestCity: { name: "$smallestCity", pop: "$smallestPop" }
23   }
24 },
25 { $sort: { state: 1 } }
26 ] )

```

## java实现

```

1 @Test
2 public void test3(){
3     //$group
4     GroupOperation groupOperation = Aggregation
5         .group("state","city").sum("pop").as("pop");
6
7     //$sort
8     SortOperation sortOperation = Aggregation
9         .sort(Sort.Direction.ASC,"pop");
10
11     //$group
12     GroupOperation groupOperation2 = Aggregation
13         .group("_id.state")
14         .last("_id.city").as("biggestCity")
15         .last("pop").as("biggestPop")
16         .first("_id.city").as("smallestCity")
17         .first("pop").as("smallestPop");
18
19     //$project
20     ProjectionOperation projectionOperation = Aggregation
21         .project("state","biggestCity","smallestCity")
22         .and("_id").as("state")
23         .andExpression(
24             "{ name: \"\$biggestCity\", pop: \"\$biggestPop\" }")
25         .as("biggestCity")
26         .andExpression(

```

```

27         "{ name: \"\$smallestCity\", pop: \"\$smallestPop\" }"
28     ).as("smallestCity")
29     .andExclude("_id");
30
31     //$sort
32     SortOperation sortOperation2 = Aggregation
33         .sort(Sort.Direction.ASC, "state");
34
35
36     // 按顺序组合每一个聚合步骤
37     TypedAggregation<Zips> typedAggregation = Aggregation.newAggregation(
38         Zips.class, groupOperation, sortOperation, groupOperation2,
39         projectionOperation, sortOperation2);
40
41     //执行聚合操作,如果不使用 Map, 也可以使用自定义的实体类来接收数据
42     AggregationResults<Map> aggregationResults = mongoTemplate
43         .aggregate(typedAggregation, Map.class);
44     // 取出最终结果
45     List<Map> mappedResults = aggregationResults.getMappedResults();
46     for(Map map:mappedResults){
47         System.out.println(map);
48     }
49
50 }

```

### 3. MongoDB索引详解

索引是一种用来快速查询数据的数据结构。B+Tree就是一种常用的数据库索引数据结构，MongoDB采用B+Tree做索引，索引创建在collections上。MongoDB不使用索引的查询，先扫描所有的文档，再匹配符合条件的文档。使用索引的查询，通过索引找到文档，使用索引能够极大的提升查询效率。

思考：MongoDB索引数据结构是B-Tree还是B+Tree？

#### 3.1 索引数据结构

B-Tree说法来源于官方文档，然后就导致了分歧：有人说MongoDB索引数据结构使用的是B-Tree,有的人又说是B+ Tree。

MongoDB官方文档: <https://docs.mongodb.com/manual/indexes/>

MongoDB indexes use a B-tree data structure.

WiredTiger官方文档: [https://source.wiredtiger.com/3.0.0/tune\\_page\\_size\\_and\\_comp.html](https://source.wiredtiger.com/3.0.0/tune_page_size_and_comp.html)

WiredTiger maintains a table's data in memory using a data structure called a B-Tree ( B+ Tree to be specific), referring to the nodes of a B-Tree as pages. Internal pages carry only keys. The leaf pages store both keys and values.

参考数据结构网站: <https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

### WiredTiger数据文件在磁盘的存储结构

B+ Tree中的leaf page包含一个页头（page header）、块头（block header）和真正的数据（key/value），其中页头定义了页的类型、页中实际载荷数据的大小、页中记录条数等信息；块头定义了此页的checksum、块在磁盘上的寻址位置等信息。

WiredTiger有一个块设备管理的模块，用来为page分配block。如果要定位某一行数据（key/value）的位置，可以先通过block的位置找到此page（相对于文件起始位置的偏移量），再通过page找到行数据的相对位置，最后可以得到行数据相对于文件起始位置的偏移量offsets。

## 3.2 索引操作

### 创建索引

创建索引语法格式

```
1 db.collection.createIndex(keys, options)
```

- Key 值为你要创建的索引字段，1 按升序创建索引， -1 按降序创建索引
- 可选参数列表如下：

Parameter	Type	Description
background	Boolean	建索引过程会阻塞其它数据库操作， background可指定以后台方式创建索引，即增加 "background" 可选参数。 "background" 默认值为false。



unique	Boolean	建立的索引是否唯一。指定为 true 创建唯一索引。默认值为 false.
name	string	索引的名称。如果未指定, MongoDB 的通过连接索引的字段名和排序顺序生成一个索引名称。
dropDups	Boolean	3.0+ 版本已废弃。在建立唯一索引时是否删除重复记录,指定 true 创建唯一索引。默认值为 false.
sparse	Boolean	对文档中不存在的字段数据不启用索引; 这个参数需要特别注意, 如果设置为 true 的话, 在索引字段中不会查询出不包含对应字段的文档。默认值为 false.
expireAfterSeconds	integer	指定一个以秒为单位的数值, 完成 TTL 设定, 设定集合的生存时间。
v	index version	索引的版本号。默认的索引版本取决于 mongod 创建索引时运行的版本。
weights	document	索引权重值, 数值在 1 到 99,999 之间, 表示该索引相对于其他索引字段的得分权重。
default_language	string	对于文本索引, 该参数决定了停用词及词干和词器的规则的列表。默认为英语
language_override	string	对于文本索引, 该参数指定了包含在文档中的字段名, 语言覆盖默认的 language, 默认值为 language.

注意: 3.0.0 版本前创建索引方法为 `db.collection.ensureIndex()`

```

1 # 创建索引后台执行
2 db.values.createIndex({open: 1, close: 1}, {background: true})
3 # 创建唯一索引
4 db.values.createIndex({title:1},{unique:true})

```

## 查看索引

```
1 #查看索引信息
2 db.books.getIndexes()
3 #查看索引键
4 db.books.getIndexKeys()
```

## 删除索引

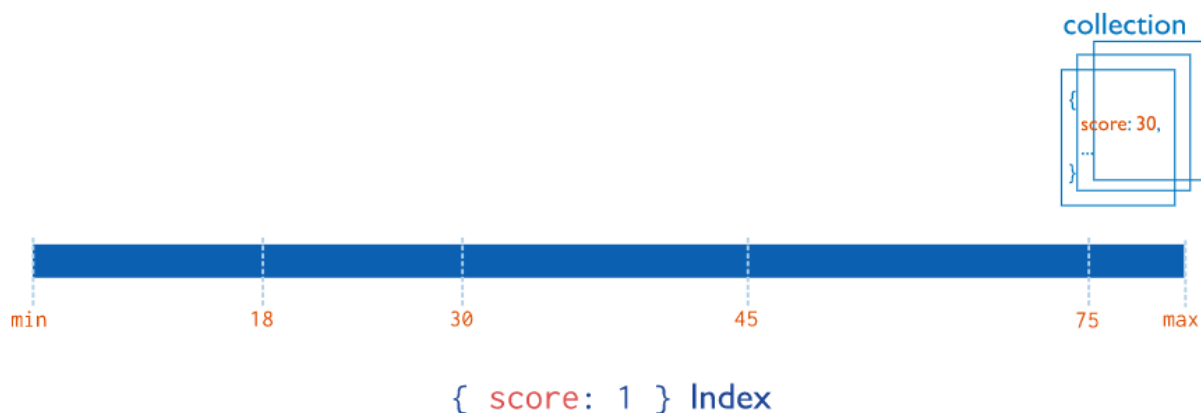
```
1 #删除集合指定索引
2 db.col.dropIndex("索引名称")
3 #删除集合所有索引    不能删除主键索引
4 db.col.dropIndexes()
```

### 3.3 索引类型

与大多数数据库一样，MongoDB支持各种丰富的索引类型，包括单键索引、复合索引，唯一索引等一些常用的结构。由于采用了灵活可变的文档类型，因此它也同样支持对嵌套字段、数组进行索引。通过建立合适的索引，我们可以极大地提升数据的检索速度。在一些特殊应用场景，MongoDB还支持地理空间索引、文本检索索引、TTL索引等不同的特性。

#### 单键索引 (Single Field Indexes)

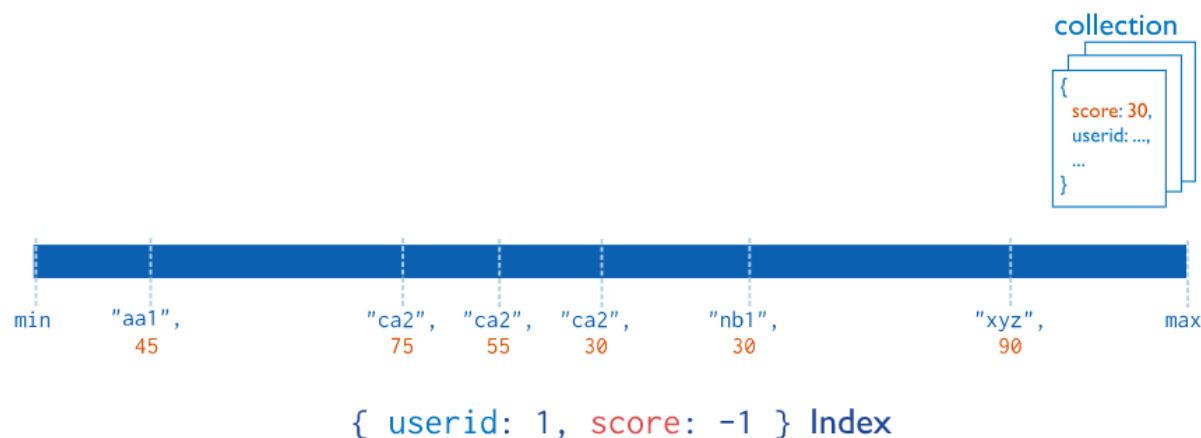
在某一个特定的字段上建立索引| mongoDB在ID上建立了唯一的单键索引,所以经常会使用id来进行查询； 在索引字段上进行精确匹配、排序以及范围查找都会使用此索引



```
1 db.books.createIndex({title:1})
2 # 对内嵌文档字段创建索引:
3 db.books.createIndex({"author.name":1})
```

## 复合索引 (Compound Index)

复合索引是多个字段组合而成的索引，其性质和单字段索引类似。但不同的是，**复合索引中字段的顺序、字段的升降序对查询性能有直接的影响**，因此在设计复合索引时则需要考虑不同的查询场景。



```
1 db.books.createIndex({type:1,favCount:1})
2 #查看执行计划
3 db.books.find({type:"novel",favCount:{$gt:50}}).explain()
```

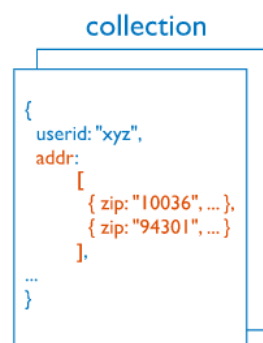
```

maxScansToExplodeReached: false,
winningPlan: {
  stage: 'FETCH',
  inputStage: {
    stage: 'IXSCAN',
    keyPattern: { type: 1, favCount: 1 },
    indexName: 'type_1_favCount_1',
    isMultiKey: false,
    multiKeyPaths: { type: [], favCount: [] },
    isUnique: false,
    isSparse: false,
    isPartial: false,
    indexVersion: 2,
    direction: 'forward',
    indexBounds: { type: [ '['novel', 'novel'] ], favCount: [ '(50, inf.0]' ] }
  }
},
rejectedPlans: []

```

## 多键索引 (Multikey Index)

在数组的属性上建立索引。针对这个数组的任意值的查询都会定位到这个文档,既多个索引入口或者键值引用同一个文档



{ "addr.zip": 1 } Index

准备inventory集合:

```

1 db.inventory.insertMany([
2   { _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] },
3   { _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] },
4   { _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] },
5   { _id: 8, type: "food", item: "ddd", ratings: [ 9, 5 ] },
6   { _id: 9, type: "food", item: "eee", ratings: [ 5, 9, 5 ] }
7 ])

```

## 创建多键索引

```
1 db.inventory.createIndex( { ratings: 1 } )
```

多键索引很容易与复合索引产生混淆，复合索引是多个字段的组合，而多键索引则仅仅是在一个字段上出现了多键（multi key）。而实质上，多键索引也可以出现在复合字段上

```
1 # 创建复合多值索引
2 db.inventory.createIndex( { item:1,ratings: 1 } )
```

注意：MongoDB并不支持一个复合索引中同时出现多个数组字段

## 嵌入文档的索引数组

```
1 db.inventory.insertMany([
2 {
3   _id: 1,
4   item: "abc",
5   stock: [
6     { size: "S", color: "red", quantity: 25 },
7     { size: "S", color: "blue", quantity: 10 },
8     { size: "M", color: "blue", quantity: 50 }
9   ]
10 },
11 {
12   _id: 2,
13   item: "def",
14   stock: [
15     { size: "S", color: "blue", quantity: 20 },
16     { size: "M", color: "blue", quantity: 5 },
17     { size: "M", color: "black", quantity: 10 },
18     { size: "L", color: "red", quantity: 2 }
19   ]
20 },
21 {
```

```

22   _id: 3,
23   item: "ijk",
24   stock: [
25     { size: "M", color: "blue", quantity: 15 },
26     { size: "L", color: "blue", quantity: 100 },
27     { size: "L", color: "red", quantity: 25 }
28   ]
29 }
30 ])

```

在包含嵌套对象的数组字段上创建多键索引

```

1  db.inventory.createIndex( { "stock.size": 1, "stock.quantity": 1 } )
2
3  db.inventory.find({"stock.size":"S","stock.quantity":{"$gt:20}}).explain()

```

```

maxScansToExplodeReached: false,
winningPlan: {
  stage: 'FETCH',
  filter: { 'stock.quantity': { '$gt': 20 } },
  inputStage: {
    stage: 'IXSCAN',
    keyPattern: { 'stock.size': 1, 'stock.quantity': 1 },
    indexName: 'stock.size_1_stock.quantity_1',
    isMultiKey: true,
    multiKeyPaths: { 'stock.size': [ 'stock' ], 'stock.quantity': [ 'stock' ] },
    isUnique: false,
    isSparse: false,
    isPartial: false,
    indexVersion: 2,
    direction: 'forward',
    indexBounds: {
      'stock.size': [ ['S', "S"] ],
      'stock.quantity': [ '[MinKey, MaxKey]' ]
    }
  }
}

```

## Hash索引 (Hashed Indexes)

不同于传统的B-Tree索引,哈希索引使用hash函数来创建索引。在索引字段上进行精确匹配,但不支持范围查询,不支持多键hash; Hash索引上的入口是均匀分布的,在分片集合中非常有用;

```

1  db.users.createIndex({username : 'hashed'})

```

## 地理空间索引 (Geospatial Index)

在移动互联网时代，基于地理位置的检索（LBS）功能几乎是所有应用系统的标配。MongoDB为地理空间检索提供了非常方便的功能。**地理空间索引 (2dsphereindex)** 就是专门用于实现位置检索的一种特殊索引。

案例：MongoDB如何实现 “查询附近商家”？

假设商家的数据模型如下：

```
1 db.restaurant.insert({
2   restaurantId: 0,
3   restaurantName:"兰州牛肉面",
4   location : {
5     type: "Point",
6     coordinates: [ -73.97, 40.77 ]
7   }
8 })
```

### 创建一个2dsphere索引

```
1 db.restaurant.createIndex({location : "2dsphere"})
```

### 查询附近10000米商家信息

```
1 db.restaurant.find( {
2   location:{
3     $near :{
4       $geometry :{
5         type : "Point" ,
6         coordinates : [ -73.88, 40.78 ]
7       } ,
8       $maxDistance:10000
9     }
10  }
11 } )
```

- \$near查询操作符，用于实现附近商家的检索，返回数据结果会按距离排序。
- \$geometry操作符用于指定一个GeoJSON格式的地理空间对象，type=Point表示地理坐标点，coordinates则是用户当前所在的经纬度位置；\$maxDistance限定了最大距离，单位是米。

## 全文索引 (Text Indexes)

MongoDB支持全文检索功能，可通过建立文本索引来实现简易的分词检索。

```
1 db.reviews.createIndex( { comments: "text" } )
```

**\$text操作符可以在有text index的集合上执行文本检索。** \$text将会使用空格和标点符号作为分隔符对检索字符串进行分词，并且对检索字符串中所有的分词结果进行一个逻辑上的 OR 操作。

全文索引能解决快速文本查找的需求，比如有一个博客文章集合，需要根据博客的内容来快速查找，则可以针对博客内容建立文本索引。

## 案例

### 数据准备

```
1 db.stores.insert(  
2   [  
3     { _id: 1, name: "Java Hut", description: "Coffee and cakes" },  
4     { _id: 2, name: "Burger Buns", description: "Gourmet hamburgers" },  
5     { _id: 3, name: "Coffee Shop", description: "Just coffee" },  
6     { _id: 4, name: "Clothes Clothes Clothes", description: "Discount clothing" },  
7     { _id: 5, name: "Java Shopping", description: "Indonesian goods" }  
8   ]  
9 )
```

### 创建name和description的全文索引

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

## 测试



通过\$text操作符来查寻数据中所有包含“coffee”、“shop”、“java”列表中任何词语的商店

```
1 db.stores.find({$text: {$search: "java coffee shop"}})
```

MongoDB的文本索引功能存在诸多限制，而官方并未提供中文分词的功能，这使得该功能的应用场景十分受限。

### 通配符索引 (Wildcard Indexes)

MongoDB的文档模式是动态变化的，而通配符索引可以建立在一些不可预知的字段上，以此实现查询的加速。MongoDB 4.2 引入了通配符索引来支持对未知或任意字段的查询。

### 案例

准备商品数据，不同商品属性不一样

```
1 db.products.insert([
2   {
3     "product_name" : "Spy Coat",
4     "product_attributes" : {
5       "material" : [ "Tweed", "Wool", "Leather" ],
6       "size" : {
7         "length" : 72,
8         "units" : "inches"
9       }
10    }
11  },
12  {
13    "product_name" : "Spy Pen",
14    "product_attributes" : {
15      "colors" : [ "Blue", "Black" ],
16      "secret_feature" : {
17        "name" : "laser",
18        "power" : "1000",
```

```

19         "units" : "watts",
20     }
21 }
22 },
23 {
24     "product_name" : "Spy Book"
25 }
26 ])

```

## 创建通配符索引

```

1 db.products.createIndex( { "product_attributes.$**" : 1 } )

```

## 测试

通配符索引可以支持任意单字段查询 product\_attributes 或其嵌入字段：

```

1 db.products.find( { "product_attributes.size.length" : { $gt : 60 } } )
2 db.products.find( { "product_attributes.material" : "Leather" } )
3 db.products.find( { "product_attributes.secret_feature.name" : "laser" } )

```

## 注意事项

- 通配符索引不兼容的索引类型或属性
- 通配符索引是稀疏的，不索引空字段。因此，通配符索引不能支持查询字段不存在的文档。

```

1 # 通配符索引不能支持以下查询
2 db.products.find( {"product_attributes" : { $exists : false } } )
3 db.products.aggregate([
4     { $match : { "product_attributes" : { $exists : false } } }
5 ])

```

- 通配符索引为文档或数组的内容生成条目，而不是文档/数组本身。因此通配符索引不能支持精确的文档/数组相

**等匹配。**通配符索引可以支持查询字段等于空文档{}的情况。

```
1 #通配符索引不能支持以下查询：
2 db.products.find({ "product_attributes.colors" : [ "Blue", "Black" ] } )
3
4 db.products.aggregate([ {
5   $match : { "product_attributes.colors" : [ "Blue", "Black" ] }
6 } ] )
```

## 3.4 索引属性

### 唯一索引 (Unique Indexes)

在现实场景中，唯一性是很常见的一种索引约束需求，重复的数据记录会带来许多处理上的麻烦，比如订单的编号、用户的登录名等。通过建立唯一性索引，可以保证集合中文档的指定字段拥有唯一值。

```
1 # 创建唯一索引
2 db.values.createIndex({title:1},{unique:true})
3 # 复合索引支持唯一性约束
4 db.values.createIndex({title:1, type:1},{unique:true})
5 #多键索引支持唯一性约束
6 db.inventory.createIndex( { ratings: 1 }, {unique:true} )
```

- **唯一性索引**对于文档中缺失的字段，会使用null值代替，因此不允许存在多个文档缺失索引字段的情况。
- 对于分片的集合，**唯一性约束必须匹配分片规则**。换句话说，为了保证全局的唯一性，分片键必须作为唯一性索引的前缀字段。

### 部分索引 (Partial Indexes)

**部分索引**仅对满足指定过滤器表达式的文档进行索引。通过在一个集合中为文档的一个子集建立索引，**部分索引具有更低的存储需求和更低的索引创建和维护的性能成本**。3.2新版功能。

部分索引提供了稀疏索引功能的超集，应该优先于稀疏索引。

```
1 db.restaurants.createIndex(  
2   { cuisine: 1, name: 1 },  
3   { partialFilterExpression: { rating: { $gt: 5 } } }  
4 )
```

partialFilterExpression选项接受指定过滤条件的文档:

- 等式表达式(例如:field: value或使用\$eq操作符)
- \$exists: true
- \$gt, \$gte, \$lt, \$lte
- \$type
- 顶层的\$and

```
1 # 符合条件, 使用索引  
2 db.restaurants.find( { cuisine: "Italian", rating: { $gte: 8 } } )  
3 # 不符合条件, 不能使用索引  
4 db.restaurants.find( { cuisine: "Italian" } )
```

## 案例1

### restaurants集合数据

```
1 db.restaurants.insert({  
2   "_id" : ObjectId("5641f6a7522545bc535b5dc9"),  
3   "address" : {  
4     "building" : "1007",  
5     "coord" : [  
6       -73.856077,  
7       40.848447  
8     ],  
9     "street" : "Morris Park Ave",  
10    "zipcode" : "10462"  
11  },  
12  "borough" : "Bronx",  
13  "cuisine" : "Bakery",  
14  "rating" : { "date" : ISODate("2014-03-03T00:00:00Z"),
```

```
15         "grade" : "A",
16         "score" : 2
17     },
18     "name" : "Morris Park Bake Shop",
19     "restaurant_id" : "30075445"
20 })
```

## 创建索引

```
1 db.restaurants.createIndex(
2   { borough: 1, cuisine: 1 },
3   { partialFilterExpression: { 'rating.grade': { $eq: "A" } } }
4 )
```

## 测试

```
1 db.restaurants.find( { borough: "Bronx", 'rating.grade': "A" } )
2 db.restaurants.find( { borough: "Bronx", cuisine: "Bakery" } )
```

## 唯一约束结合部分索引使用导致唯一约束失效的问题

注意：如果同时指定了`partialFilterExpression`和唯一约束，那么唯一约束只适用于满足筛选器表达式的文档。如果文档不满足筛选条件，那么带有唯一约束的部分索引不会阻止插入不满足唯一约束的文档。

## 案例2

### users集合数据准备

```
1 db.users.insertMany( [
2   { username: "david", age: 29 },
3   { username: "amanda", age: 35 },
4   { username: "rajiv", age: 57 }
5 ] )
```

创建索引，指定`username`字段和部分过滤器表达式`age: {$gte: 21}`的唯一约束。

```
1 db.users.createIndex(  
2   { username: 1 },  
3   { unique: true, partialFilterExpression: { age: { $gte: 21 } } }  
4 )
```

## 测试

索引防止了以下文档的插入，因为文档已经存在，且指定的用户名和年龄字段大于21:

```
1 db.users.insertMany( [  
2   { username: "david", age: 27 },  
3   { username: "amanda", age: 25 },  
4   { username: "rajiv", age: 32 }  
5 ] )
```

但是，以下具有重复用户名的文档是允许的，因为唯一约束只适用于年龄大于或等于21岁的文档。

```
1 db.users.insertMany( [  
2   { username: "david", age: 20 },  
3   { username: "amanda" },  
4   { username: "rajiv", age: null }  
5 ] )
```

## 稀疏索引 (Sparse Indexes)

索引的稀疏属性确保索引只包含具有索引字段的文档的条目，索引将跳过没有索引字段的文档。

特性： 只对存在字段的文档进行索引（包括字段值为null的文档）

```
1 #不索引不包含xmpp_id字段的文档  
2 db.addresses.createIndex( { "xmpp_id": 1 }, { sparse: true } )
```

如果稀疏索引会导致查询和排序操作的结果集不完整，MongoDB将不会使用该索引，除非hint()明确指定索引。

## 案例

### 数据准备

```
1 db.scores.insertMany([
2     {"userid" : "newbie"},
3     {"userid" : "abby", "score" : 82},
4     {"userid" : "nina", "score" : 90}
5 ])
```

### 创建稀疏索引

```
1 db.scores.createIndex( { score: 1 } , { sparse: true } )
```

### 测试

```
1 # 使用稀疏索引
2 db.scores.find( { score: { $lt: 90 } } )
3
4 # 即使排序是通过索引字段，MongoDB也不会选择稀疏索引来完成查询，以返回完整的结果
5 db.scores.find().sort( { score: -1 } )
6
7 # 要使用稀疏索引，使用hint()显式指定索引
8 db.scores.find().sort( { score: -1 } ).hint( { score: 1 } )
```

同时具有稀疏性和唯一性的索引可以防止集合中存在字段值重复的文档，但允许不包含此索引字段的文档插入。

## 案例

```
1 db.scores.dropIndex({score:1})
2 # 创建具有唯一约束的稀疏索引
3 db.scores.createIndex( { score: 1 } , { sparse: true, unique: true } )
```

### 测试

这个索引将允许插入具有唯一的分数字段值或不包含分数字段的文档。因此，给定scores集合中的现有文档，索引允许以下插入操作：

```
1 db.scores.insertMany( [  
2   { "userid": "AAAAAAA", "score": 50 },  
3   { "userid": "BBBBBBB", "score": 64 },  
4   { "userid": "CCCCCCC" },  
5   { "userid": "CCCCCCC" }  
6 ] )
```

索引不允许添加下列文件，因为已经存在评分为82和90的文件：

```
1 db.scores.insertMany( [  
2   { "userid": "AAAAAAA", "score": 82 },  
3   { "userid": "BBBBBBB", "score": 90 }  
4 ] )
```

## TTL索引 (TTL Indexes)

在一般的应用系统中，并非所有的数据都需要永久存储。例如一些系统事件、用户消息等，这些数据随着时间的推移，其重要程度逐渐降低。更重要的是，存储这些大量的历史数据需要花费较高的成本，因此项目中通常会对过期且不再使用的数据进行老化处理。

通常的做法如下：

方案一：为每个数据记录一个时间戳，应用侧开启一个定时器，按时间戳定期删除过期的数据。

方案二：数据按日期进行分表，同一天的数据归档到同一张表，同样使用定时器删除过期的表。

对于数据老化，MongoDB提供了一种更加便捷的做法：TTL (Time To Live) 索引。TTL索引需要声明在一个日期类型的字段中，TTL 索引是特殊的单字段索引，MongoDB 可以使用它在一定时间或特定时钟时间后自动从集合中删除文档。

```
1 # 创建 TTL 索引，TTL 值为3600秒  
2 db.eventlog.createIndex( { "lastModifiedDate": 1 }, { expireAfterSeconds: 3600 } )
```



对集合创建TTL索引之后，MongoDB会在周期性运行的后台线程中对该集合进行检查及数据清理工作。除了数据老化功能，TTL索引具有普通索引的功能，同样可以用于加速数据的查询。

TTL 索引不保证过期数据会在过期后立即被删除。文档过期和 MongoDB 从数据库中删除文档的时间之间可能存在延迟。删除过期文档的后台任务每 60 秒运行一次。因此，在文档到期和后台任务运行之间的时间段内，文档可能会保留在集合中。

## 案例

### 数据准备

```
1 db.log_events.insertOne( {  
2   "createdAt": new Date(),  
3   "logEvent": 2,  
4   "logMessage": "Success!"  
5 } )
```

### 创建TTL索引

```
1 db.log_events.createIndex( { "createdAt": 1 }, { expireAfterSeconds: 20 } )
```

### 可变的过期时间

TTL索引在创建之后，仍然可以对过期时间进行修改。这需要使用collMod命令对索引的定义进行变更

```
1 db.runCommand({collMod:"log_events",index:{keyPattern:  
  {createdAt:1},expireAfterSeconds:600}})
```

```
> db.runCommand({collMod:"log_events",index:{keyPattern:{createdAt:1},expireAfterSeconds:600}})
{ "expireAfterSeconds_old" : 20, "expireAfterSeconds_new" : 600, "ok" : 1 }
> db.log_events.getIndexes()
[
  {
    "v" : 2,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_"
  },
  {
    "v" : 2,
    "key" : {
      "createdAt" : 1
    },
    "name" : "createdAt_1"
    "expireAfterSeconds" : NumberLong(600)
  }
]
```

## 使用约束

TTL索引的确可以减少开发的工作量，而且通过数据库自动清理的方式会更加高效、可靠，但是在使用TTL索引时需要注意以下的限制：

- TTL索引只能支持单个字段，并且必须是非\_id字段。
- TTL索引不能用于固定集合。
- TTL索引无法保证及时的数据老化，MongoDB会通过后台的TTLMonitor定时器来清理老化数据，默认的间隔时间是1分钟。当然如果在数据库负载过高的情况下，TTL的行为则会进一步受到影响。
- TTL索引对于数据的清理仅仅使用了remove命令，这种方式并不是很高效。因此TTL Monitor在运行期间对系统CPU、磁盘都会造成一定的压力。相比之下，按日期分表的方式操作会更加高效。

## 隐藏索引 (Hidden Indexes)

隐藏索引对查询规划器不可见，不能用于支持查询。通过对规划器隐藏索引，用户可以在不实际删除索引的情况下评估删除索引的潜在影响。如果影响是负面的，用户可以取消隐藏索引，而不必重新创建已删除的索引。[4.4新版功能](#)。

```
1 创建隐藏索引
2 db.restaurants.createIndex({ borough: 1 },{ hidden: true });
3 # 隐藏现有索引
4 db.restaurants.hideIndex( { borough: 1} );
5 db.restaurants.hideIndex( "索引名称" )
6 # 取消隐藏索引
```

```
7 db.restaurants.unhideIndex( { borough: 1 } );
8 db.restaurants.unhideIndex( "索引名称" );
```

## 案例

```
1 db.scores.insertMany([
2     { "userid" : "newbie" },
3     { "userid" : "abby", "score" : 82 },
4     { "userid" : "nina", "score" : 90 }
5 ])
```

## 创建隐藏索引

```
1 db.scores.createIndex(
2     { userid: 1 },
3     { hidden: true }
4 )
```

## 查看索引信息

```
1 db.scores.getIndexes()
```

索引属性hidden只在值为true时返回

```
> db.scores.getIndexes()
[
  {
    "v" : 2,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_"
  },
  {
    "v" : 2,
    "hidden" : true,
    "key" : {
      "userid" : 1
    },
    "name" : "userid_1"
  }
]
```

测试

```
1 # 不使用索引
2 db.scores.find({userid:"abby"}).explain()
3
4 #取消隐藏索引
5 db.scores.unhideIndex( { userid: 1} )
6 #使用索引
7 db.scores.find({userid:"abby"}).explain()
```

## 3.5 索引使用建议

### 1) 为每一个查询建立合适的索引

这个是针对数据量较大比如说超过几十上百万（文档数目）数量级的集合。如果没有索引 MongoDB 需要把所有的 Document 从盘上读到内存，这会对 MongoDB 服务器造成较大的压力并影响到其他请求的执行。

### 2) 创建合适的复合索引，不要依赖于交叉索引

如果你的查询会使用到多个字段，MongoDB 有两个索引技术可以使用：交叉索引和复合索引。交叉索引就是针对每个字段单独建立一个单字段索引，然后在查询执行时候使用相应的单字段索引进行索引交叉而得到查询结果。交叉索引目前触发率较低，所以如果你有一个多字段查询的时候，建议使用复合索引能够保证索引正常的使用。

```
1 #查找所有年龄小于30岁的深圳市马拉松运动员
2 db.athletics.find({sport: "marathon", location: "sz", age: {$lt: 30}}})
3 #创建复合索引
4 db.athletics.createIndex({sport:1, location:1, age:1})
```

### 3) 复合索引字段顺序：匹配条件在前，范围条件在后 (Equality First, Range After)

前面的例子，在创建复合索引时如果条件有匹配和范围之分，那么匹配条件 (sport: "marathon" ) 应该在复合索引的前面。范围条件(age: <30)字段应该放在复合索引的后面。

### 4) 尽可能使用覆盖索引 (Covered Index)

建议只返回需要的字段，同时，利用覆盖索引来提升性能。

### 5) 建索引要在后台运行

在对一个集合创建索引时，该集合所在的数据库将不接受其他读写操作。对大数据量的集合建索引，建议使用后台运行选项 {background: true}

### 6) 避免设计过长的数组索引

数组索引是多值的，在存储时需要使用更多的空间。如果索引的数组长度特别长，或者数组的增长不受控制，则可能导致索引空间急剧膨胀。

## 3.6 explain执行计划

通常我们需要关心的问题：

- 查询是否使用了索引
- 索引是否减少了扫描的记录数量
- 是否存在低效的内存排序

MongoDB提供了explain命令，它可以帮助我们评估指定查询模型 (querymodel) 的执行计划，根据实际情况进行调整，然后提高查询效率。

explain()方法的形式如下：

```
1 db.collection.find().explain(<verbose>)
```

- verbose 可选参数，表示执行计划的输出模式，默认queryPlanner

模式名字	描述

queryPlanner	执行计划的详细信息，包括查询计划、集合信息、查询条件、最佳执行计划、查询方式和 MongoDB 服务信息等
exectionStats	最佳执行计划的执行情况和被拒绝的计划等信息
allPlansExecution	选择并执行最佳执行计划，并返回最佳执行计划和其他执行计划的执行情况

queryPlanner

```
1 # 未创建title的索引
2 db.books.find({title:"book-1"}).explain("queryPlanner")
```

```
> db.books.find({title:"book-1"}).explain("queryPlanner")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "appdb.books",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "title" : {
        "$eq" : "book-1"
      }
    },
    "queryHash" : "6E0D6672",
    "planCacheKey" : "6E0D6672",
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "title" : {
          "$eq" : "book-1"
        }
      }
    },
    "direction" : "forward"
  },
  "rejectedPlans" : [ ]
},
"serverInfo" : {
  "host" : "hadoop02",
  "port" : 27017,
  "version" : "4.4.9",
  "gitVersion" : "b4048e19814bfefbac717cf5a880076aa69aba481"
},
"ok" : 1
}
```

全表扫描

字段名称	描述
plannerVersion	执行计划的版本
namespace	查询的集合

indexFilterSet	是否使用索引
parsedQuery	查询条件
winningPlan	最佳执行计划
stage	查询方式
filter	过滤条件
direction	查询顺序
rejectedPlans	拒绝的执行计划
serverInfo	mongodb服务器信息

## executionStats

executionStats 模式的返回信息中包含了 queryPlanner 模式的所有字段，并且还包含了最佳执行计划的执行情况

```

1 #创建索引
2 db.books.createIndex({title:1})
3
4 db.books.find({title:"book-1"}).explain("executionStats")

```

```

> db.books.find({title:"book-1"}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "appdb.books",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "title" : {
        "$eq" : "book-1"
      }
    },
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "title" : 1
        },
        "indexName" : "title_1",
        "isMultiKey" : false,
        "multiKeyPaths" : {
          "title" : [ ]
        },

```

```

        "isUnique" : false,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 2,
        "direction" : "forward",
        "indexBounds" : {
            "title" : [
                ["book-1\\", \\book-1\\"]
            ]
        }
    },
    "rejectedPlans" : [ ]
},
"executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 1,
    "executionTimeMillis" : 2,
    "totalKeysExamined" : 1,
    "totalDocsExamined" : 1,
    "executionStages" : {
        "stage" : "FETCH",
        "nReturned" : 1,
        "executionTimeMillisEstimate" : 0,
        "works" : 2,
        "advanced" : 1,
        "needTime" : 0,
        "needYield" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 1,
        "docsExamined" : 1,
        "alreadyHasObj" : 0,
        "inputStage" : {
            "stage" : "IXSCAN",
            "nReturned" : 1,
            "executionTimeMillisEstimate" : 0,
            "works" : 2,
            "advanced" : 1,
            "needTime" : 0,
            "needYield" : 0,
            "saveState" : 0,
            "restoreState" : 0,
            "isEOF" : 1,
            "keyPattern" : {
                "title" : 1
            },
            "indexName" : "title_1",
            "isMultiKey" : false,
            "multiKeyPaths" : {
                "title" : [ ]
            },
            "isUnique" : false,
            "isSparse" : false,
            "isPartial" : false,
            "indexVersion" : 2,
            "direction" : "forward",
            "indexBounds" : {
                "title" : [

```



```

        "keysExamined" : 1,
        "seeks" : 1,
        "dupsTested" : 0,
        "dupsDropped" : 0
    }
}
},
"serverInfo" : {
    "host" : "hadoop02",
    "port" : 27017,
    "version" : "4.4.9",
    "gitVersion" : "b4048e19814bfebac717cf5a880076aa69aba481"
},
"ok" : 1
}
>

```

字段名称	描述
winningPlan.inputStage	用来描述子stage，并且为其父stage提供文档和索引关键字
<b>winningPlan.inputStage.stage</b>	子查询方式
winningPlan.inputStage.keyPattern	所扫描的index内容
winningPlan.inputStage.indexName	索引名
winningPlan.inputStage.isMultiKey	是否是Multikey。如果索引建立在array上，将是true
executionStats.executionSuccess	是否执行成功
executionStats.nReturned	返回的个数
executionStats.executionTimeMillis	这条语句执行时间
executionStats.executionStages.executionTimeMillisEstimate	检索文档获取数据的时间
executionStats.executionStages.inputStage.executionTimeMillisEstimate	扫描获取数据的时间
executionStats.totalKeysExamined	索引扫描次数
executionStats.totalDocsExamined	文档扫描次数
executionStats.executionStages.isEOF	是否到达 stream 结尾，1 或者 true 代表已到达结尾
executionStats.executionStages.works	工作单元数，一个查询会分解成小的工作单元
executionStats.executionStages.advanced	优先返回的结果数

executionStats.executionStages.docsExamined	文档检查数
---------------------------------------------	-------

allPlansExecution

allPlansExecution返回的信息包含 executionStats 模式的内容，且包含allPlansExecution:[]块

```
1  "allPlansExecution" : [  
2      {  
3          "nReturned" : <int>,  
4          "executionTimeMillisEstimate" : <int>,  
5          "totalKeysExamined" : <int>,  
6          "totalDocsExamined" : <int>,  
7          "executionStages" : {  
8              "stage" : <STAGEA>,  
9              "nReturned" : <int>,  
10             "executionTimeMillisEstimate" : <int>,  
11             ...  
12         }  
13     }  
14 },  
15 ...  
16 ]  
17  
18
```

stage状态

状态	描述
COLLSCAN	全表扫描
IXSCAN	索引扫描
FETCH	根据索引检索指定文档
SHARD_MERGE	将各个分片返回数据进行合并
SORT	在内存中进行了排序
LIMIT	使用limit限制返回数

SKIP	使用skip进行跳过
IDHACK	对_id进行查询
SHARDING_FILTER	通过mongos对分片数据进行查询
COUNTSCAN	count不使用Index进行count时的stage返回
COUNT_SCAN	count使用了Index进行count时的stage返回
SUBPLA	未使用到索引的\$or查询的stage返回
TEXT	使用全文索引进行查询时候的stage返回
PROJECTION	限定返回字段时候stage的返回

执行计划的返回结果中尽量不要出现以下stage:

- COLLSCAN(全表扫描)
- SORT(使用sort但是无index)
- 不合理的SKIP
- SUBPLA(未用到index的\$or)
- COUNTSCAN(不使用index进行count)