

# MongoDB教育訓練-09

## 查詢緩慢

- 沒有使用index

## 解決

- 查看database log
- 開啟 profiling

## Finding slow operations

- The most common source of bad performance is a missing index
- This is definitely not the only cause - bad client code is a close second.
- There are two ways to find inefficient Queries
  - The database log
  - Profiling a database

Profiling requires a feature that is not available on Atlas Shared Instances.

## The getLog Command

The command shown here gets the last 1024 log entries.

Returns a single document - logs are in the **log** array as strings.

In MongoDB 4.4+, these strings are JSON.

Previously they were just text.



```
> var loglines = db.adminCommand({getLog:"global"})
> printjson(loglines)
{
  "totalLinesWritten" : 2625918,
  "log" : [ ... ],
  "ok" : 1,
  "$clusterTime" : {
    "clusterTime" : Timestamp(1594293362, 5),
    "signature" : {
      "hash" : BinData(0,"3+Mc0j...2cUERBHnc3Qs="),
      "keyId" : NumberLong("6835033877094858756")
    }
  },
  "operationTime" : Timestamp(1594293362, 5)
}
```

# Copying the log to a collection

From MongoDB 4.4, all log output is now in JSON format.

You can easily copy the output of getLog to a temporary collection.

And then search or aggregate the data.

```
> use temp
> var x = db.adminCommand( { getLog: "global" } )
> db.log.drop()
>
db.log.insertMany(x.log.map(d=>JSON.parse(d.replace(/\$/g,"_"))))
> db.log.find({"attr.durationMillis":{"$gt:10}}).pretty()
{
  _id : ObjectId("5fcf704fda3f8a47fffdc6e5"),
  t : {
    $date : "2020-12-08T11:52:56.311+00:00"
  },
  s : "I",
  c : "NETWORK",
  ctx : "conn329",
  msg : "Connection ended",
  attr : {
    remote : "192.168.254.30:55800",
    connectionId : 329,
    connectionCount : 36,
    durationMillis: 12
  }
}
...
```

level 2 性能會影響

## The setProfilingLevel command

```
db.setProfilingLevel(
  level, slows
)
```

**level:** 0, 1, or 2

**0** - The Profiler is off and does not collect any data.

**1** - Record slow queries in a collection for analysis

**2** - Record all queries in a collection for analysis

**slows** - the time threshold in milliseconds for slow operations

```
> db.setProfilingLevel(0,5)
{"was" : 0,"slows" : 100,"sampleRate" : 1,"ok" : 1, ... }
> use sample_airbnb
> db.listingsAndReviews.find({amenities:"Snooker"})
> use admin
> var loglines = db.runCommand({getLog:"global"})
> printjson(loglines)
...
"2020-07-09T11:49:00.237+0000 I COMMAND [conn214]
command sample_airbnb.listingsAndReviews appName: \"MongoDB
Shell\" command: find { find: \"listingsAndReviews\", filter: {
amenities: \"Snooker\" }, lsid: { id:
UUID(\"6496eb58-cea3-4c05-b237-0d04e2ffc60b\") }, $clusterTime: {
clusterTime: Timestamp(1594295320, 1), signature: { hash:
BinData(0, E2DB7AAFBFE43F8A424F60EA04033378C1B73B89), keyId:
6847444150736912387 } }, $db: \"sample_airbnb\" } planSummary:
COLLSCAN keysExamined:0 docsExamined:5555 cursorExhausted:1
numYields:43 nreturned:0 queryHash:0AEEE9D2 planCacheKey:0AEEE9D2
reslen:246 locks:{ ReplicationStateTransition: { acquireCount: {
w: 44 } }, Global: { acquireCount: { r: 44 } }, Database: {
acquireCount: { r: 44 } },
...
```

db.setProfilingLevel(1,5) #level 1, 查詢大於5毫秒

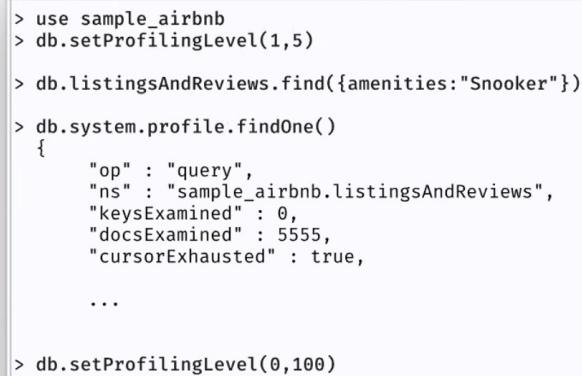
# Recording slow operations

Can be used to capture longer term but means extra writes.

**TURN THIS OFF AFTER USE  
AS IT CAN SLOW  
PRODUCTION!**

Recorded in a capped collection - so only 10MB max.

Visualization and aggregations are possible.



```
> use sample_airbnb
> db.setProfilingLevel(1,5)

> db.listingsAndReviews.find({amenities:"Snooker"})

> db.system.profile.findOne()
{
  "op" : "query",
  "ns" : "sample_airbnb.listingsAndReviews",
  "keysExamined" : 0,
  "docsExamined" : 5555,
  "cursorExhausted" : true,
  ...
}

> db.setProfilingLevel(0,100)
```

## 慢查詢的原因

- 缺少index
- 寫入資料超出disk 性能
- 高CPU使用
- 巢狀join

## Causes of slow operations

- Missing indexes leading to lots of Disk I/O
- Writes to cache outstripping disk write capability
  - Waiting for cache to flush
- Locking
  - You can see in logs what locks things take
  - Most things don't block others - but a few admin things do
- Excessive CPU usage
  - Constantly logging in
  - Document lock contention.
  - Inappropriately large arrays
  - Running code in the database
- Nested Joins exploding complexity.

# Compared to SQL

MongoDB find():

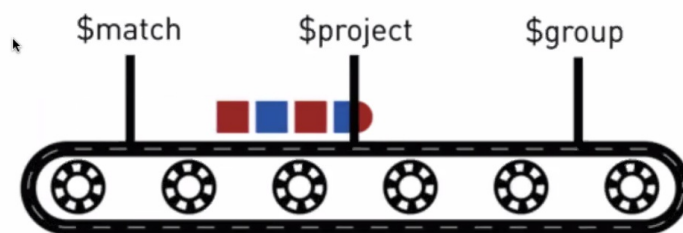
```
SELECT a,c,b FROM database.table WHERE d<100 ORDER BY d ASC
```

MongoDB aggregate():

```
SELECT b+c as a,SUM(e) AS t  
FROM D.T LEFT JOIN D.T2 ON y  
WHERE T2.A=T.B  
GROUP BY a  
HAVING t > 100
```

## Aggregation is a pipeline

- Each transformation is a single step known as a stage.
- Compared to one huge SQL style statement this is
  - Easier to understand
  - Easier to debug
  - Easier for MongoDB to rewrite and optimize



## Stages we already know

---

- `$match`            equivalent to `find(query)`
- `$project`        equivalent to `find({},projection)`
- `$sort`            equivalent to `find().sort(order)`
- `$limit`           equivalent to `find().limit(num)`
- `$skip`            equivalent to `find().skip(num)`
- `$count`           equivalent to `find().count()`

When these are used at the start of a pipeline, they are transformed to a `find()` by the query optimizer.

## Comparing Aggregation syntax

---

Find the name of the host in Canada with the most "total listings":

Using `find()`

```
db.listingsAndReviews.find(
  {"address.country":"Canada"},
  {"_id":0, "host.host_total_listings_count":1, "host.host_name":1}
).sort({"host.host_total_listings_count":-1}).limit(1)
```

Using `aggregate()`

```
db.listingsAndReviews.aggregate([
  {$match:{"address.country":"Canada"}},
  {$sort: {"host.host_total_listings_count":-1 }},
  {$limit:1},
  {$project:{"_id":0, "host.host_total_listings_count":1, "host.host_name":1}}
])
```

# Comparing Aggregation syntax

Find the name of the host in Canada with the most "total listings":

Using find()

```
db.listingsAndReviews.find(
  {"address.country":"Canada"},
  {"_id":0, "host.host_total_listings_count":1, "host.host_name":1}
).sort({"host.host_total_listings_count":-1}).limit(1)
```

Using aggregate()

```
db.listingsAndReviews.aggregate([
  {$match:{"address.country":"Canada"}},
  {$sort: {"host.host_total_listings_count":-1 }},
  {$limit:1},
  {$project:{"_id":0, "host.host_total_listings_count":1, "host.host_name":1}}
])
```

## Dollar Overloading

`{$match: {a: 5}}` - Dollar on left means a stage name - in this case a **\$match** stage

`{$set: {b: "$a"}}` - Dollar on right of colon "**\$a**" refers to the value of field a

`{$set: {area: {$multiply: [5,10]}}` - **\$multiply** is an expression name left of colon

```
{$set: {priceswithtax: {$map: {input: "$prices",
                             as: "p",
                             in: {$multiply :["$$$p",1.08]}}}}}
```

**\$\$\$p** Refers to the temporary loop variable "p" declared in \$map

`{$set: {displayprice: {$literal: "$12"}}` - Use \$literal when you want either a string with a \$ or to \$project an explicit number



# How to write Aggregations

Think as a programmer - not as a database shell.

Define variables. Doing this helps you keep track of brackets.

It also helps you really understand the Object concept.

```
//Do it THIS way for ease of testing and debugging

> no_celebs = {$match:{"user.followers_count":{$lt:200000}}}

> name_only = {$project:{"user.name":1,
"user.followers_count":1,_id:0}}

> most_popular = {$sort: {"user.followers_count":-1}}

> first_in_list = {$limit:1}

> pipeline = [no_celebs,name_only,most_popular,first_in_list]

> db.twitter.aggregate(pipeline)
```

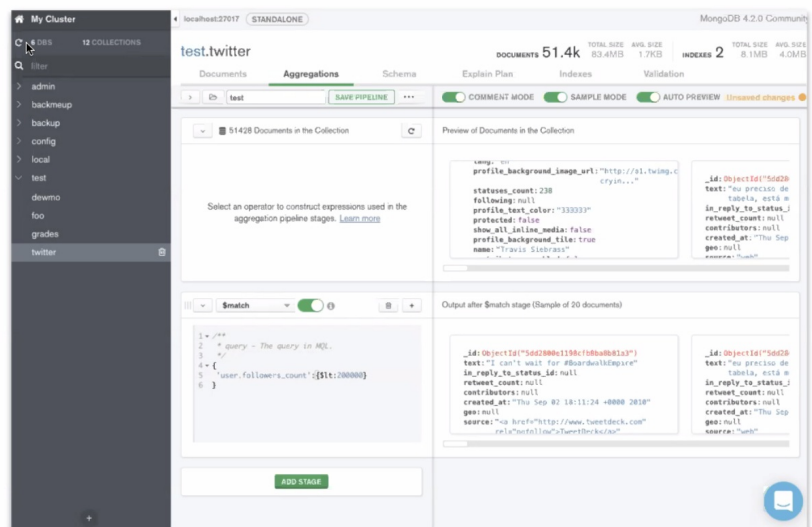
[MongoDB Compass Download](#) | [MongoDB](#)

## GUI Aggregation Builder

Compass is MongoDB's GUI query tool.

It has an Aggregation Builder and visualizer.

It is helpful when learning but limits your thinking - be careful.

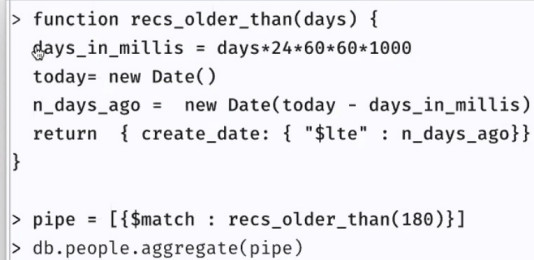


# Aggregation stages from code

Write functions that generate aggregation stages.

Allow them to be parameterised and call each other.

This is a **very** simple example.



```
> function recs_older_than(days) {
  days_in_millis = days*24*60*60*1000
  today= new Date()
  n_days_ago = new Date(today - days_in_millis)
  return { create_date: { "$lte" : n_days_ago}}
}

> pipe = [{ $match : recs_older_than(180)}]
> db.people.aggregate(pipe)
```

## Aggregation Expressions

- Aggregations have stages like `$match` and `$project` and `$sort`
- Stages often referred to **expressions**
- Expressions can be simple like "\$name" - the value of the field "name"
- Or more complicated like this for RMS\* of an array of values.  
`{{ $sqrt:{{ $avg:{{ $map:{{ input: "$a", in:{{ $multiply: [ "$$this", "$$this" ] } } } } } } } } }`
- Or 100s of KB long performing a complex calculation.

## Some examples of arithmetic expressions

```
{ $subtract : [ "$cost", "$price" ] }
```

```
{ $add : [ "$price", "$shipping" ] }
```

```
{ $add : [ "$price", "$shipping", "$tax" ] }
```

```
{ $multiply : [ "$price", "$taxrate" ] }
```

```
{ $divide : [ "$quantity", "$price" ] }
```

```
{ $add : [ "$price" , { $multiply : [ "$price", "$taxrate" ] } ] }
```



# Expression Categories

---

- [Arithmetic Expression Operators](#)
- [Array Expression Operators](#)
- [Boolean Expression Operators](#)
- [Comparison Expression Operators](#)
- [Conditional Expression Operators](#)
- [Date Expression Operators](#)
- [Literal Expression Operator](#)
- [Object Expression Operators](#)
- [Set Expression Operators](#)
- [String Expression Operators](#)
- [Text Expression Operator](#)
- [Trigonometry Expression Operators](#)
- [Type Expression Operators](#)
- [Accumulators \(\\$group\)](#)

## Using \$project & \$set

---

- \$project specifies the output document shape, fields are defined by expressions.

```
{
  $project: {
    name: "$fullname",
    average_speed: { $divide : ["$distance", "$time"] }
  }
}
```

- To add in additional fields rather than specify the whole output use \$set, not \$project
  - \$set also lets you replace existing values

```
{ $set : { average_speed: { $divide : ["$distance", "$time"] } }
```

# Exercise

---

Using the Shell or Compass aggregation builder and the airbnb listings data (`sample_airbnb.listingsAndReviews`) read the following question and output the answer. You will need to use a `$set` stage and a `$project` stage with expressions

1. The price for the basic rental is in the `$price` field; for that price, you can have the number of guests provided in `$guests_included` field.
  2. The property may take more guests in total (maximum guests is in `$accommodates` property).
  3. You have to pay `$extra_people` dollars per person for every person more than `$guests_included`
- Calculate how many extra guests you can have for each property and add that as a field using `$set`
  - Calculate how much it would cost with these extra guests. `$project` the basic price and the price if fully occupied with `$accommodates` people.

## The \$group stage

---

Take the incoming document stream and reduce it to a smaller set of documents by combining (GROUP BY in SQL is the closest equivalent)

`_id` is what MongoDB uses as a unique field. Each unique value represents one 'group.'

```
{ $group: { _id : <expression>,
            field1 : { <$accum>: <expression> },
            ... } }
```

```
{ $group : { _id: "$country",
              population : { $sum: "$city_population" } } }
```

# The \$unwind stage

- The opposite of \$group
- Applied to any array field
- Converts one document to many
- One per value in the array

```
{ a: 1, b: [2,3,4] }
```

Unwind on b ({ \$unwind: "\$b" }) gives 3 documents in the pipeline.

```
{a:1,b:2}
```

```
{a:1,b:3}
```

```
{a:1,b:4}
```

## "Join" Stages

- \$lookup
  - Query or Run a pipeline and embed results
  - Like Left Outer Join or Nested Select
  - Needs Indexing and tuning
  - 'From' collection cannot be sharded
  - Intended to lookup rapidly changing dimensions like stock prices
  - NOT an excuse for relational design
  - Two forms Query and Sub-pipeline.
  - Returns an array of results

```
#Get Bobs stock records
db.stocks.aggregate([
  {$match: { customer: "bob"}},
  #For each on $lookup the current value of that stock
  {$lookup: {
    from: "currentprices",
    localField: "symbol",
    foreignField: "tkr",
    as: "currentPrice" }},
  #For each record multiply how many bob has by the latest price
  {$set: {
    holdings: {
      $multiply: [ "$numheld",
        {$arrayElemAt: ["$currentPrice", 0]
      }
    }
  }}
  #Add them up by stock
  {$group: { _id: "symbol",
    value: {$sum: "$holdings"}
  }}
])

{ _id: "MSFT", value: 20124 }
{ _id: "ORCL", value: 650 }
{ _id: "MDB", value: 987521 }
```

## "Join" Stages

- \$graphLookup
  - Recursively lookup on same collection
  - A way to traverse trees or graphs
  - Rarely useful in practice as both limited in functionality and relatively slow.
  - There are better schema design patterns for most use cases.

```
db.landmarks.aggregate([
  { $match: { _id: "Brooklyn Bridge" } },
  { $graphLookup: {
    from: "places",
    startWith: "$location",
    connectFromField: "isIn",
    connectToField: "_id",
    as: "address"
  }}}])
```

Returns:

```
{ _id: "Brooklyn Bridge",
  location: "Brooklyn",
  address : [ { _id: "Brooklyn", isIn: "NYC"},
    { _id: "NYC", isIn: "New York" },
    { _id: "New York", isIn: "USA" } ]
}
```

## More grouping

**\$bucket:** Group by defined Ranges of values

**\$bucketAuto:** Group into N similar sized groups

**\$facet:** Combine sub pipelines in one document.

Copyright 2020-2021 MongoDB, Inc. All rights reserved.



## Grouping: \$sortByCount

Shortcut for one of the most popular groupings to do

See what the most common values are of a field.

```
[{
  $group: {
    id: "$city" ,
    count: {
      $sum: 1
    }
  },
  {
    $sort: {
      count: -1
    }
  }
}]

{ $sortByCount: "$city" }
```

# Structural stages

---

<code>\$set</code>	Add extra fields without \$projecting all of them
<code>\$out</code>	Write results to a new collection
<code>\$merge</code>	Update an existing collection
<code>\$replaceRoot</code>	Create a whole new shape of top-level document
<code>\$sample</code>	Choose a random set of docs from the input

## Recap

---

- Aggregation is a way to manipulate records inside the server before returning them.
- It is a full programming language in a mostly functional paradigm.
- It can filter, summarise and calculate almost anything.
- It does take time to learn and master - but it's worth it.

## Expression Variables

---

- Use a double dollar, `$$`
- Internal variables `$$NOW`, `$$CLUSTER_TIME`, `$$ROOT`, `$$REMOVE`
- Used in `$let`, `$map`, `$reduce` expressions
  - `$map` and `$reduce` are list comprehension expressions
  - `$let` allows you to optimise by evaluating something only once