Mục lục

[I. Tìm hiểu NoSQL 2](#_Toc403669141)

[1. Vì sao chọn NoSQL? 2](#_Toc403669142)

[a) Yếu điểm của RDBMS 2](#_Toc403669143)

[b) Đặc điểm nổi bật của NoSQL 2](#_Toc403669144)

[2. So sánh NoSQL và RDBMS 3](#_Toc403669145)

[a) Mặt tích cực của RDBMS 3](#_Toc403669146)

[b) Mặt tích cực của NoSQL 3](#_Toc403669147)

[c) Sự khác nhau 3](#_Toc403669148)

[3. Ưu nhược điểm khi chuyển từ RDBMS sang NoSQL. 4](#_Toc403669149)

[a) Ưu điểm 4](#_Toc403669150)

[b) Nhược điểm 4](#_Toc403669151)

[II. Tìm hiểu MongoDB 5](#_Toc403669152)

[1. Tóm tắt lịch sử 5](#_Toc403669153)

[2. Các khái niệm cơ bản trong MongoDB 5](#_Toc403669154)

[a) Văn bản (Document) 5](#_Toc403669155)

[b) Bộ sưu tập (Collection) 7](#_Toc403669156)

[c) Chỉ mục (Index) 8](#_Toc403669157)

[3. Cài đặt MongoDB 10](#_Toc403669158)

[4. Các thao tác cơ bản với MongoDB 15](#_Toc403669159)

[a) Thao tác thêm văn bản. 15](#_Toc403669160)

[b) Thao tác xóa document, collection 16](#_Toc403669161)

[c) Thao tác cập nhật 17](#_Toc403669162)

[d) Thao tác truy vấn 32](#_Toc403669163)

[e) Làm việc với chỉ số (Index) 48](#_Toc403669164)

[5. Một số ví dụ câu lệnh được ánh xạ từ SQL 51](#_Toc403669165)

# Tìm hiểu NoSQL

## Vì sao chọn NoSQL?

### Yếu điểm của RDBMS

Ngày nay, Big Data, Big Users và Cloud Computing là các xu hướng đang phát triển cực kì mạnh mẽ đã dẫn tới sự phát triển theo của công nghệ NoSQL là sự tất yếu. NoSQL ngày càng được xem là một sự thay thế cho CSDL quan hệ.

Các RDBMs (Relational Database Management System) hiện tại bộc lộ nhiều yếu điểm như đánh chỉ mục cho một lượng lớn dữ liệu, phân trang, phân phối luồng dữ liệu media, gặp khó khăn khi xử lý một lượng dữ liệu cực lớn và cập nhật liên tục do số lượng người dùng quá nhiều ở một thời điểm, gặp vấn đề về hiệu suất với những bài toán lớn về hệ thống thông tin, phân tán hay lưu trữ dữ liệu.

Ngoài ra, với chi phí triển khai cũng như phát triển các ứng dụng sử dụng CSDL quan hệ tốn kém đặc biệt khi truy vấn một lượng bản ghi lớn trong thời gian dài. Hơn thế nữa, với sự phát triển mạnh của những thiết bị cầm tay như smartphone thì CSDL quan hệ bộc lộ rõ khuyết điểm vì bộ nhớ của thiết bị cũng như khả năng xử lý thấp.

### Đặc điểm nổi bật của NoSQL

* Khả năng mở rộng (Scalability): gần như không có giới hạn cho dữ liệu và người dùng hệ thống.
* Tính sẵn sàng (High Availability): Vì chấp nhận sự trung lặp trong lưu trữ dữ liệu nên nếu một node bị chết sẽ không ảnh hưởng tới toàn bộ hệ thống.
* Atomicity: Độc lập data state trong các thao tác.
* Consistency: Chấp nhận tính nhất quán yếu, cập nhật mới không đảm bảo các truy xuất sau đó thấy được sự thay đổi. Sau một khoảng thời gian lan truyền thì tính nhất quán cuối cùng cua dữ liệu mới được đảm bảo.
* Durability: Dữ liệu có thể tồn tại trong bộ nhớ máy tính, nhưng cũng đồng thời được lưu trữ tại đĩa cứng.
* Deployment Flexibility: Hệ thống sẽ tư động nhận biết việc bổ sung thêm hay loại bỏ các node. Hệ thống không đòi hỏi cấu hình phần cứng mạnh, đồng nhất.
* Modeling flexibility: cặp dữ liệu key-value, dữ liệu cấu trúc (Hierarchical data), Graphs.
* Query Flexibility: Multi-Gets, load một tập giá trị dựa vào tập khóa (Range queries)
* Horizontal scalable: hay còn gọi là khả năng mở rộng chiều ngang. Bình thường, với các hệ quản trị cơ sở dữ liệu quan hệ, khi mà dữ liệu quá lớn phương pháp tăng khả năng lưu trữ là sẽ phải mở rộng (nâng cấp máy chủ), còn đối với NoSQL thì chỉ cần bổ sung thêm máy chủ khác vì hệ thống hỗ trợ lưu trữ phân tán trên nhiều máy.

## So sánh NoSQL và RDBMS

Cả NoSQL và RDBMS đều là những công nghệ tuyệt vời, mỗi công nghệ đều hay ở một lĩnh vực khác, biết sử dụng linh hoạt 2 công nghệ này sẽ giúp ích và phát huy tối đa lợi ích.

### Mặt tích cực của RDBMS

* RDBMS được hỗ trợ tốt với tính ACID đặc trưng.
* Xử lý dễ dàng với độ chính xác cao trong việc giải quyết những thao tác lớn.

### Mặt tích cực của NoSQL

* **Hiệu suất hoạt động cao: NoSQL có hiệu suất hoạt động cao, lưu trữ lượng lớn dữ liệu để đáp ứng nhu cầu lưu trữ ngày càng tăng hiện nay. Tuy nhiên để đạt được điều này cần loại bỏ đi một số thứ như: ràng buộc dữ liệu của mô hình quan hệ, tính nhất quán dữ liệu, ngôn ngữ truy vấn SQL. Đồng thời NoSQL có một số cải tiến mới như sử dụng tốt index, khả năng phân tán dễ dàng đã giúp NoSQL có một hiệu suất hoạt động rất cao.**
* **Khả năng phân trang: phân trang trong cơ sở dữ liệu quan hệ khá khó khăn khi không có một phương pháp chính thống nào để phục vụ cho việc này. Người lập trình phải dùng các phương pháp khác nhau để có thể lấy đúng số item cần lấy. Trong khi NoSQL hổ trợ rất tốt việc này đồng thời hiệu suất khi phân trang không hề giảm.**
* **NoSQL là nguồn mở:** Các sản phẩm nguồn mở đưa ra cho những người phát triển với nhiều lợi ích to lớn, trong đó việc sử dụng miễn phí là một lợi ích lớn. Những lợi ích khác: phần mềm nguồn mở có xu hướng sẽ là tin cậy hơn, an ninh hơn và nhanh hơn để triển khai so với các lựa chọn thay thế sở hữu độc quyền.Ví dụ như các hệ quản trị cơ sở dữ liệu (CSDL) NoSQL: Cassandra, CouchDB, Hbase, RavenDB, MongoDB và Redis.
* **Việc mở rộng phạm vi là mềm dẻo**: NoSQL thay thế câu thần chú cũ của các nhà quản trị CSDL về “mở rộng phạm vi” với một thứ mới: “mở rộng ra ngoài”. Thay vì bổ sung thêm các máy chủ lớn hơn để điều khiển nhiều tải dữ liệu hơn, thì CSDL NoSQL cho phép một công ty phân tán tải qua nhiều máy chủ khi mà tải gia tăng.

### Sự khác nhau

|  |  |  |
| --- | --- | --- |
|  | **RDBMS** | **NoSQL** |
| **Architecture** | Cấu trúc dựa trên các bảng | Cấu trúc phân làm 4 loại chính:   * Document base * Key-value pair base * Graph database * Wide-column Store |
| **Schema** | Shema được định nghĩa để lưu trữ dữ liệu có cấu trúc | Không có một định nghĩa trước nào cho chema, mà shema linh hoạt dựa theo thành phần dữ liệu |
| **Scalability** | Được mở rộng theo chiều dọc, nếu chúng ta muốn mở rộng cơ sở dữ liệu thì pahir nâng cấp phần cứng, điều nay flamf hạn chế khả năng mở rộng của CSDL quạn hệ | Mở rộng theo chiều ngang, nghĩa là nếu muốn mở rộng cơ sở dữ liệu, chúng ta chỉ cần them các node và tạo ra mạng lưới phân phối dựa trên yêu cầu mà ta đưa ra, đây là cách giảm tải trên cơ sở dữ liệu quan hệ. |
| **Data Retrieval** | Sử dụng SQL để định nghĩa và thao tác dữ liệu. | Tập trung vào collection và documents, được gọi là UnSQL tức CSDL ko quan hệ. Do đang trong quá trình phát triển nên vẫn còn tùy thuộc vào nhà cung cấp. |
| **Systems in Market Place** | * [Oracle](http://www.oracle.com/) * [MySql](http://www.mysql.com/) * [Postgres](http://www.postgresql.org/) * [MS-SQL](http://www.microsoft.com/en-us/sqlserver/default.aspx%E2%80%8E) | * [MongoDB](http://www.mongodb.org/) * BigTable * Redis * RavenDb * [Cassendra](http://cassandra.apache.org/) * [Hbase](https://hbase.apache.org/%E2%80%8E) * [Neo4j](http://www.neo4j.org/) * [CouchDb](http://couchdb.apache.org/) |
| **Classification** | Phân làm hai loại chính là CSDL quan hệ mã nguồn mở và CSDL mã nguồn đóng. | Phân chia thành 5 loại dựa trên cách lưu trữ data:   * Key-value pair store database * Graph database * Document Store * Column Store * XML Store |
| **Based Concept** | Dựa trên những quy luật và tính ACID | Dựa trên 3 tính chất: nhất quán, sắn sang và phân mảnh (Consistency, Availability, Partition Tolerance) |

## Ưu nhược điểm khi chuyển từ RDBMS sang NoSQL.

### Ưu điểm

Khi chuyển từ RDBMS sang NoSQL ta sẽ thừa hưởng được tất cả những gì là điểm mạnh của NoSQL đã được trình bày ở trên

### Nhược điểm

Mỗi một record trong CSDL quan hệ phù hợp với một lược đồ quan hệ, với một số cố định các cột với mục đích và kiểu dữ liệu cụ thể, các dòng là giống nhau về định nghĩa. Dữ liệu không được chuẩn hóa giữa nhiều bảng, ngược lại có sự trùng lặp ít trong dữ liệu. Nhược điểm nữa là nếu chỉnh sửa 1 bảng nào đó ta sẽ phải kiểm tra ràng buộc của các bảng liên quan để đảm bảo giữ sự ổn định cho cơ sở dữ liệu.

Ngoài ra, khi chuyển từ RDBMS sang NoSQL ta còn gặp phải vấn đề là phân tích, thiết kế lại database sao cho phù hợp với cách tiếp cận mới.

Để tối ưu nhất, tùy theo từng ứng dụng cụ thể mà ta kết hợp sử dụng cả RDBMS và NoSQL.

# Tìm hiểu MongoDB

MongoDB là một cơ sở dữ liệu (CSDL) mã nguồn mở, nó có cơ chế hoạt động với hiệu suất cao, dễ dàng cấu hình, và dễ dàng mở rộng vì được xây dựng hoàn toàn trên ngôn ngữ C/C++.

MongoDB không có các lược đồ quan hệ giống như hệ cơ sở dữ liệu quan hệ (CSDLQH). Đặc biệt, MongoDB là một “Cơ sở dữ liệu hướng tài liệu”.

MongoDB quản lý các dữ liệu thành các tập dạng BSON (Binary JSON), BSON là kiểu dữ đã được mã hóa nhị phân của dạng dữ liệu JSON. Các dữ liệu này có thể được lồng vào nhau, hình thành nên một cấu trúc phức tạp, nhưng vẫn dễ dàng truy suất được nhờ được đánh các chỉ mục. Điều này cho phép cho phép dữ liệu được thiết kế tự nhiên và dễ dàng phù hợp cho ứng dụng.

## Tóm tắt lịch sử

* Năm 2007, MongoDB được thành lập vào bởi 10gen, tại New York, Mỹ.
* Năm 2009, MongoDB đã chính thức trở thành mã nguồn mở.
* Trong tháng 3 năm 2011, từ phiên bản 1.4, MongoDB đã được hoàn thiện và sẵn sàng cho các ứng dụng.
* Tới tháng 9 năm 2014, phiên bản 2.6.4 (8/11/2014) là phiên bản mới nhất.

## Các khái niệm cơ bản trong MongoDB

### Văn bản (Document)

MongoDB lưu trữ tất cả dữ liệu ở dạng văn bản, theo cấu trúc lưu trữ JSON có các trường và giá trị tương ứng.

Nó được xem tương đương với một dòng dữ liệu trong cơ sở dữ liệu quan hệ.

{“item”: “pencil”, “qty”: 500, “type”: “no.2”}

#### Định đạng văn bản (Document Format)

MongoDB lưu trữ các văn bản ở dạng BSON theo một cách tuần tự. BSON là cách biễu diễn nhị phân của các cấu trúc JSON.

#### Cấu trúc văn bản (Document Structure)

Văn bản MongoDB có cấu trúc bao gồm các trường (field) và các giá trị (value) tương ứng, theo cấu trúc sau:

{

field1: value1,

field2: value2,

field3: value3,

...

fieldN: valueN

}

Ví dụ:

{“greeting”: “Hello, world!”}

Văn bản trên gồm 1 trường là “greeting” với giá trị tương ứng là “Hello, world!”

Ví dụ:

{“greeting”: “Hello, world!”, “foo”: 3}

Các giá trị có thể ở bất cứ kiểu dữ liệu nào, có thể bao gồm các văn bản khác, dạng mảng, hoặc là các mảng văn bản.

var mydoc = {

\_id: ObjectId(“5099803df3f4948bd2f98391”),

name: {first: “Alan”, last: “Turing”},

birth: new Date(“Jun 23, 1912”),

death: new Date(“Jun 07, 1954”),

contribs: [ “Turing machine”, “Turing test”, “Turingery” ],

views: NumberLong(1250000)

}

Ví dụ văn bản trên có:

* \_id: đối tượng ObjectId.
* name: có một văn bản con gồm các trường là “first” và “last” cùng các giá trị tương ứng.
* birth và death: giá trị có kiểu dữ liệu dạng Date.
* Contribs: giá trị có kiểu dữ liệu mảng của chuỗi.
* Views: giá trị có kiểu dữ liệu Long interger.

#### Trường có một số quy luật phải được tuân thủ:

* Tên trường là một chuỗi ký tự.
* Trường có nội dung “\_id” được dành riêng cho các khóa chính của văn bản, và cần được tuân thủ các điều kiện của khóa chính.
* Trường không thể được bắt đầu bằng ký tự “$”.
* Trường không thể chứa dấu chấm “.”.
* Trường phải là một chuỗi kí tự khác rỗng “null”
* Trong một văn bản, không thể chứa các trường giống nhau, ví dụ trường hợp sau là không hợp lệ:

{“greeting”: “Hello, world!”, “greeting”: “Hello, MongoDB!”}

#### Trường khóa (\_id) có một số ràng buộc sau:

* Mặc định, MongoDB sẽ tự động tạo ra trường “\_id” mỗi khi có một văn bản được tạo.
* Trường khóa luôn luôn là trường đầu tiên của văn bản. Nếu như hệ thống nhận một văn bản có trường “\_id” không phải đứng đầu, thì hệ thống sẽ tự động chuyển trường “\_id” lên đầu.

### Bộ sưu tập (Collection)

Bộ sưu tập là một nhóm các văn bản, nó được xem là tương đương với các dòng dữ liệu của một bảng trong cơ sở dữ liệu quan hệ.

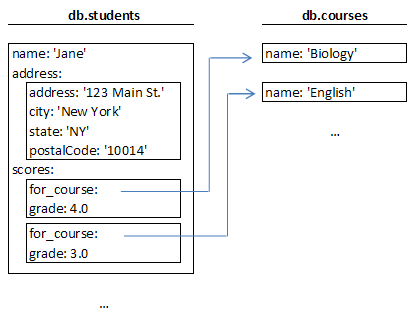
Một bộ sưu tập được xem là một Schema-Free, nghĩa là các văn bản phải có cấu trúc giống nhau mới có thể lưu vào chung trong một bộ sưu tập.

Ví dụ các văn bản sau có thể được dùng để lưu trong một bộ sưu tập:

{“greeting”: “Hello, world!”}

{“foo”: 5}

Bộ sưu tập được xác định bởi tên của nó là một chuỗi UTF-8



Hình : Hình minh họa có 2 bộ sưu tập: students và courses.

Các văn bản student được nhúng văn bản address và văn bản score. Trong đó, văn bản Score được tham chiếu đến Courses. So sánh với lược đồ quan hệ: ta cần lưu Score vào bảng riêng và dùng khóa ngoài liên kết với Student.

### Chỉ mục (Index)

Chỉ mục trong MongoDB hỗ trợ thực hiện các truy vấn một các hiệu quả. Nếu không có chỉ mục,

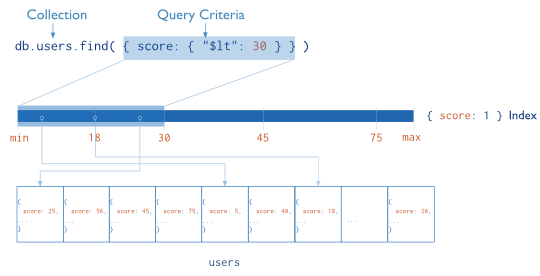
MongoDB sẽ phải duyệt tất cả các tài liệu để lưa chọn ra những văn bản phù hợp nhất cho lệnh truy vấn. Cách này thường không hiệu quả vì nó xử lý một khối dữ liệu lớn mà không cần thiết.

Chỉ số là cầu trúc đặc biệt dùng để lưu trữ một phần nhỏ dữ liệu của bộ sưu tập. Chỉ số lưu trữ giá trị của một trường cụ thể hoặc được thiết lập, sắp xếp theo các giá trị của các trường.

Cơ bản, chỉ số trong MongoDB tương tự như trong các hệ thống dữ liệu khác. MongoDB định nghĩa các chỉ mục vào một bộ sưu tập cấp độ và hỗ trợ các trường hoặc các trường con trong văn bản trong bộ sưu tập dữ liệu MongoDB.

Nếu các chỉ số được sử dụng một cách thích hợp trong truy vấn, MongoDB có thể dùng các chỉ số để giới hạn lại số lượng các văn bản cần được kiểm tra. Trong một số trường hợp, MongoDB có thể dùng dữ liệu từ các chỉ số để xác định văn bản nào phù hợp với lệnh truy vấn.

Sơ đồ dưới đây minh họa một truy vấn chọn tài liệu sử dụng một chỉ số.

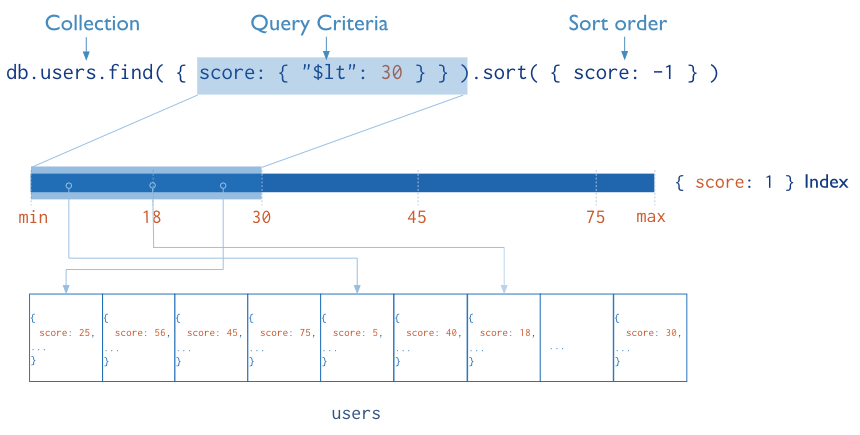


Hình : Sơ đồ của một truy vấn văn bản sử dụng một chỉ số.

MongoDB thu hẹp phạm vi tìm kiếm văn bản bằng các duyệt tất cả văn bản có giá trị score nhỏ hơn 30.

#### Sắp xếp kết quả

MongoDB có thể dùng các chỉ số để trả về các văn bản đã được sắp xếp bằng các chỉ mục một các trực tiếp mà không cần phải thêm một bước sắp xếp trung gian.

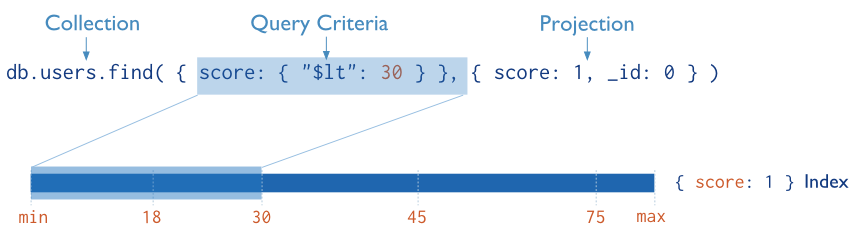


Hình : Sơ đồ sử dụng chỉ số để truy vấn và sắp xếp tăng dần của “score”.

MongoDB có thể dùng các chỉ số để duyệt theo thứ tự tăng dần hoặc giảm dần và trả về kết quả đã được sắp xếp.

#### Tối ưu truy kết quả truy suất

Khi câu truy vấn hợp lệ với yêu cầu một chỉ số cụ thể, MongoDB sẽ trả về trực tiếp kết quả tương ứng với chỉ số mà không cần phải duyệt hay ghi các văn bản vào bộ nhớ. Điều này giúp cho hệ thống hoạt động hiệu quả hơn.



Hình : Sơ đồ của truy vấn chỉ sử dụng chỉ số để truy vấn và trả về kết quả trực tiếp.

MongoDB không cần phải kiểm tra dữ liệu bên ngoài của các chỉ số để thực hiện các truy vấn.

## Cài đặt MongoDB

#### Cài đặt

MongoDB không hỗ trợ trên Windows XP, vì vậy cần dùng các hệ điều hành cao hơn để sử dụng cài đặt MongoDB

##### Bước 1:

Tải phiên bản mới nhất từ trang chủ MongoDB: <http://www.mongodb.org/>

Đảm bảo lựa chọn đúng phiên bản thích hợp nhất cho hệ điều hành hiện tại của máy. (có thể lựa chọn định đạng là msi hoặc zip, khuyên mọi người nên dùng định dạng zip)

##### Bước 2:

###### Đối với định dạng zip:

Giải nén, và di chuyển thư mục đó vào một đường dẫn mong muốn và đổi tên cho dễ dàng thao tác.

Ở đây, ví dụ như D:\mongodb:

###### Đối với định dạng msi:

Tiến hành cài đặt bình thường với các chế độ mặc định (Typical):

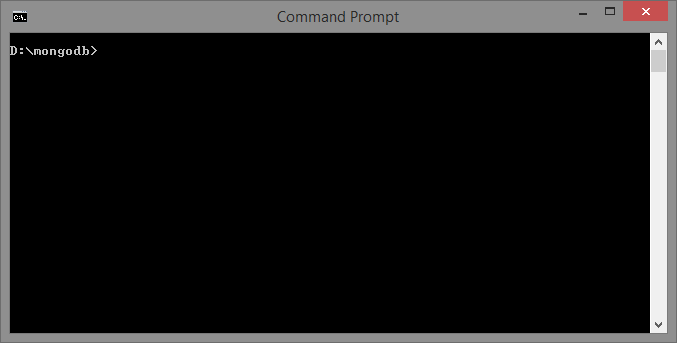
Hệ thống sẽ tự tạo ra 1 thư mục có đường dẫn “C:\Program Files\MongoDB 2.x.x”, có thể sử dụng đường dẫn mặc định này để cài đặt, nhưng khuyên nên di chuyển thư mục này và đổi tên cho dễ dàng cấu hình cài đặt: D:\mongodb

Hoặc cũng có thể tùy chỉnh đường dẫn này qua chế độ cài đặt tùy chỉnh (Customer) trong các bước tiến hành cài đặt.

#### Cấu hình MongoDB

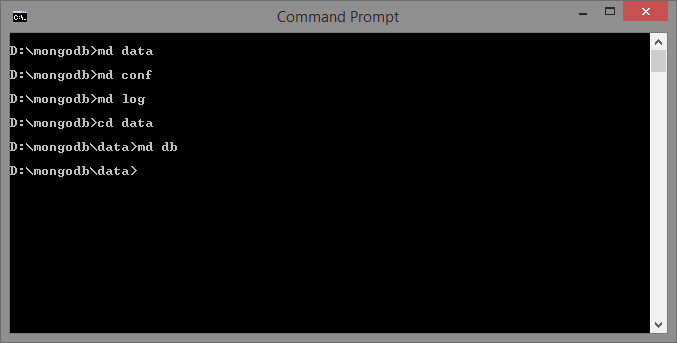
##### Bước 1: Cài đặt biến môi trường cho MongoDB

(Với đường dẫn chính của MongoDB là: “D:\mongodb”)



Cần tạo

* 1 thư mục data: Để chứa dữ liệu chính của MongoDB (“D:\mongodb\data”)
  + 1 thư muc con db trong thư mục data. (“D:\mongodb\data\db”)
* 1 thư mục log: Để ghi lại quá trình hoạt động của MongoDB (“D:\mongodb\log”)
* 1 thự mục conf: Để lưu lại các thông tin cấu hình của MongoDB (“D:\mongodb\conf”)



Tạo thư mục data,conf, log bằng câu lệnh

Md data

Md conf

Md log

Tạo tiếp thư mục db trong thư mục data

Cd data  
md db

Tiếp theo ta tiến hành tạo và ghi cấu hình cho file “conf/mongo.cfg”, ta thực hiện câu lệnh sau:

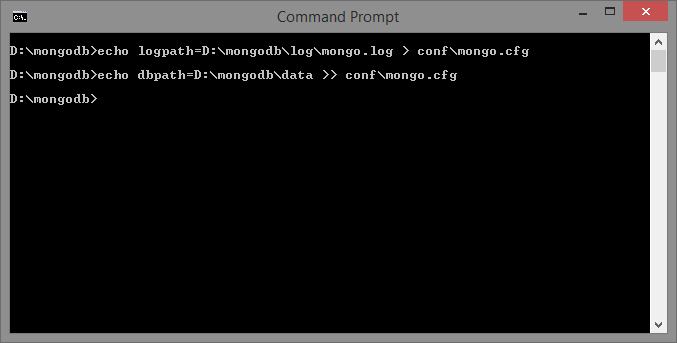
Ghi đường dẫn file log:

Cd..

Echo logpath=D:\mongodb\log\mongo.log > conf\mong.cfg

Ghi đường dẫn file db

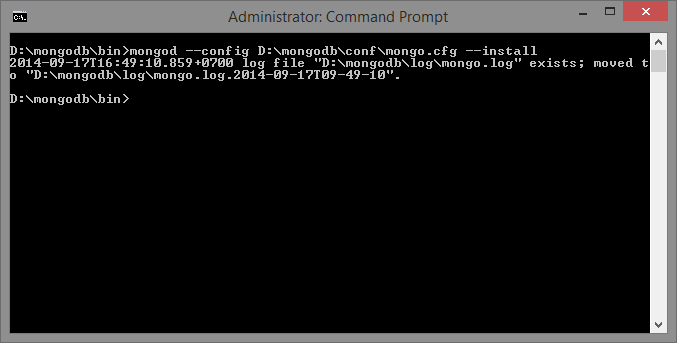
Echo dbpath=D:\mongodb\data >> conf\mongo.cfg



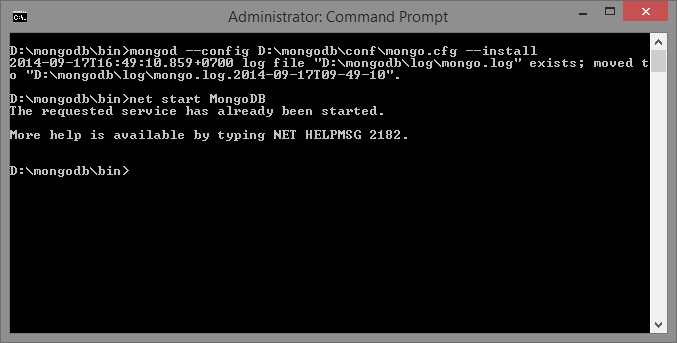
Tiếp theo, ta tiến cài đặt MongoDB service với file cấu hình mongo.cfg mới được thiết lập, di chuyển vào thư mục bin và chạy câu lệnh bằng cách:

Mongod.exe --config D:\mongodb\conf\mongo.cfg --install

Chú ý: ở bước này ta cần chạy câu lệnh với quyền Administrator

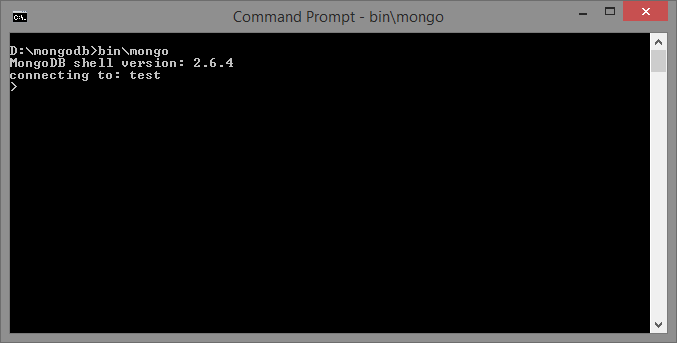


Tiếp theo ta khởi động MongoDB services



Tiếp theo, thử kết nối đến MongoDB bằng đường dẫn.

Bin/mongo



Kết quả như trên thể hiện việc cài đặt và kết nối đã thành công.

Chú ý: Nếu trong trong đường dẫn có dấu khoảng trắng thì cần để đường dẫn trong dấu ngoặc kép, ví dụ

cd “D:/x y z/mongodb/”

## Các thao tác cơ bản với MongoDB

Các câu lệnh được chạy trong thư mục hiện hành bin của MongoDB

Để khởi động MongoDB service (cần chạy ở quyền admin)

Net start MongoDB

###### Để dùng MongoDB service (cần chạy ở quyền admin):

Net stop MongoDB

Khi đã khởi động MongoDB service, để kết nối đền MongoDB:

Mongo

MongoDB shell version: <whatever>

url: test

connecting to: test

type “help” for help

>

connecting to: cho biết rằng CSDL đang được kết nối, để thay đổi:

use mydb

switched to db mydb

Sử dụng câu lệnh use để chuyển CSDL sử dụng, tuy nhiên nếu CSDL được sử dụng lần đầu thì thực chất CSDL vẫn chưa được tạo, MongoDB sẽ tạo ngầm khi thêm dữ liệu vào, sử dụng câu lệnh:

[tên\_CSDL].[tên\_collection].save(x);

Hoặc

[tên\_CSDL].[tên\_collection].insert(x);

### Thao tác thêm văn bản.

Phương thức thêm document vào trong collection là thao tác cơ bản trong MongoDB.

Để thực hiện thêm, ta thực hiện theo cú pháp sau:

[tên\_CSDL].[tên\_collection].save(x);

Hoặc

[tên\_CSDL].[tên\_collection].insert(x);

Trong đó: x là một document hay là một mảng bao gồm nhiều document

Ví dụ:

db.foo.insert({“bar”: “baz”})

Hệ thống sẽ tự thêm trường “\_id” vào document (nếu như nó không được khai báo).

#### Thêm đồng loạt (Batch insert):

Nếu như bạn gặp trường hợp cần thêm nhiều document vào collection, có thể sử dụng batch insert để có được tốc độ nhanh hơn. Batch insert cho phép bạn sử dụng các mảng chứa các document làm tham số. Nó có thể chứa một vài, hay hàng trăm, hàng ngàn document hoặc nhiều hơn trong một mảng, và thời gian thực hiện nhanh hơn đáng kể so với thực hiện lần lượt insert từng document.

Batch insert được sử dụng trong chủ yếu trong việc thêm một lượng dữ liệu lớn trong một lần, như các ứng dụng về phân tích dữ liệu, hoặc sao lưu, phục hồi một khối lượng dữ liệu lớn nào đó.

Batch insert chỉ có thể thực hiện được 1 lần trên 1 collection, nó không thể thực hiện trên nhiều collection thông qua 1 câu lệnh.

Ví dụ:

db.products.insert(

[

{\_id: 11, item: “pencil”, qty: 50, type: “no.2”},

{item: “pen”, qty: 20},

{item: “eraser”, qty: 25}

]

)

##### Toán

### Thao tác xóa document, collection

Để xóa tất cả các document trong 1 collection

Ta sử dụng cú pháp sau để xóa:

[tên\_CSDL].[tên\_collection].remove(x);

Trong đó, x là tham số của nó là một document hay là một cú pháp được trả về là document, thì phương thức sẽ xóa các document đó trong collection.

Nếu x = {}: (Rỗng), thì collection sẽ xóa hết tất cả các document bên trong nó.

Ví dụ:

db.mailing.remove({“opt-out”: true})

Khi dữ liệu đã được xóa, sẽ không có cách nào để phục hồi lại dữ liệu đã bị xóa.

#### Xóa nhanh

Phương thức xóa document là phương thức được thực hiện khá nhanh chóng.

Với một collection với nhiều document, để xóa tất cả, nên dùng phương thực collection.drop(), và sau đó tạo lại những index nếu cần thiết. Như vậy sẽ nhanh hơn rất nhiều so với việc duyệt lần lượt các document là xóa tuần tự.

Ví dụ: Ta có một collection với 1 000 000 document:

for i in range(1000000):

collection.insert({“foo”: “bar”, “baz”: i, “z”: 10 - i})

Và sau đó, ta dùng một đoạn mã giả để duyệt tuần tự và xóa tất cả các document vừa thêm vào như sau:

db = Connection().foo

collection = db.bar

DBCursor cursor = collection.find();

while(cursor.hasNext())

{

collection.remove(cursor.next());

}

Đối với cách dùng này, hệ thống sẽ mất khoảng 3 phút đề hoàn thành công việc.

Thay vào đó sẽ tiết kiệm thời gian hơn nếu như sử dụng phương thức collection.drop() và sau đó tạo lại các Index cần thiết (nếu có).

### Thao tác cập nhật

Một document đã được lưu trong collection, nó vẫn có thể thay đổi giá trị thông qua phương thức

[tên\_CSDL].[tên\_collection].update(x,y);

Phương thức gồm có 2 tham số:

x là tham số để xác định document cần thay đổi

y là tham số cập nhật là document đó.

Nếu như 2 phương thức cùng đến trong khoảng thời gian tương tự nhau. Cái nào đến trước sẽ được thực thi trước, đến sau sẽ được thực thi sau.

#### Sử dụng để thay thế document

Cách sử dụng:

Giả sử, ta có một document đơn giản cần được cập nhật lại. Giá trị của document như sau:

{

“\_id”: ObjectId(“4b2b9f67a1f631733d917a7a”),

“name”: “joe”,

“friends”: 32,

“enemies”: 2

}

Và ta muốn sau khi cập nhật, giá trị của nó sẽ thay đổi lại theo giá trị sau:

{

“\_id”: ObjectId(“4b2b9f67a1f631733d917a7a”),

“username”: “joe”,

“relationships”:

{

“friends”: 32,

“enemies”: 2

}

}

Ta có thể thực hiện sự thay đổi này bằng cách sử dụng phương thức update như sau:

> var joe = db.users.findOne({“name”: “joe”});

> joe.relationships = {“friends”: joe.friends, “enemies”: joe.enemies};

{

“friends”: 32,

“enemies”: 2

}

> joe.username = joe.name;

“joe”

> delete joe.friends;

true

> delete joe.enemies;

true

> delete joe.name;

true

> db.users.update({“name”: “joe”}, joe);

Sau đó, hãy dùng phương thức find() hoặc findOne() để kiểm tra document đã được cập nhật.

Trong trường hợp do trong collection có nhiều hơn một document thỏa điều kiện x.

Ví dụ như:

> db.people.find({“name”: “joe”})

{“\_id”: ObjectId(“4b2b9f67a1f631733d917a7b”), “name”: “joe”, “age”: 65},

{“\_id”: ObjectId(“4b2b9f67a1f631733d917a7c”), “name”: “joe”, “age”: 20},

{“\_id”: ObjectId(“4b2b9f67a1f631733d917a7d”), “name”: “joe”, “age”: 49},

Thì khi thực hiện câu update:

> joe = db.people.findOne({“name”: “joe”, “age”: 20});

{

“\_id”: ObjectId(“4b2b9f67a1f631733d917a7c”),

“name”: “joe”,

“age”: 20

}

> joe.age++;

> db.people.update({“name”: “joe”}, joe);

MongoDB sẽ thông báo lỗi “E11001 duplicate key on update”.

Có nghĩa là khi xác định document là {“name”: “joe”} thì MongoDB sẽ trả về 3 document thỏa mãn điều kiện này.

Theo thứ tự, document đầu tiên cập nhật đó là document

{“\_id”: ObjectId(“4b2b9f67a1f631733d917a7b”), “name”: “joe”, “age”: 65},

MongoDB sẽ cố gắng thử thay đổi nội dung của document này thành document joe được khai báo trước đó. Nhưng collection đã tồn tại document có trường \_id tương với \_id của biến joe. Cho nên, MongoDB sẽ báo lỗi vì trường \_id là trường không được phép trùng giá trị.

Để khác phục vấn đề này, thay vì sử dụng {“name”: “joe”} là điều kiện để xác định document, ta nên dùng \_id hoặc các trường nào không có trùng lắp về giá trị để làm điều kiện xác định document. Như vậy, sẽ đảm bảo được document được chọn và cập nhật là duy nhất và chính xác.

#### Sử dụng để thay đổi giá trị trường

Thông thường, các yêu cầu cập nhật dữ liệu thường chỉ là cập nhật lại một phần nào đó của một document.

Ví dụ:

Giả sử ta có một website phân tích dữ liệu người dùng, và trong một document của sở sở dữ liệu có cấu trúc như sau:

{

“\_id”: ObjectId(“4b253b067525f35f94b60a31”),

“url”: “www.example.com”,

“pageviews”: 52

}

Mục đích ta muốn tăng số lượng pageviews lên 1 mỗi khi có ai đó đăng nhập vào website có đường dẫn là giá trị của trường url.

Để làm được như trên, cứ mỗi khi có lượt truy cập vào website có địa chỉ là giá trị url. Ta sẽ thiết đặt hàm để gọi đến phương thức update, và dùng biến thay đổi là “$inc” để tăng giá trị cho trường pageviews:

> db.analytics.update({“url”: “www.example.com”},

...{“$inc”: {“pageviews”: 1}})

Bây giờ hãy xuất giá trị của document ra để xem lại sự thay đổi.

> db.analytics.find()

{

“\_id”: ObjectId(“4b253b067525f35f94b60a31”),

“url”: “www.example.com”,

“pageviews”: 53

}

Lưu ý: Sử dụng các thay đổi giá trị của trường không thể thay đổi được giá trị của trường “\_id” (thay vào đó, sử dụng cách thay thế document để làm việc này), những trường còn lại, kể các các trường index đều có thể thay thế được bằng phương pháp này.

#### Các toán tử phổ biến

##### Cấu trúc toán tử “$set”

Biến thay đổi $set có tác dụng thay đổi giá trị của trường. Nếu trường đó chưa tồn tại, nó sẽ được tạo mới, vì thế, nó có thể được dùng với mục đích thiết đặt lại giá trị cho trường hoặc tạo 1 thêm 1 trường mới.

Ví dụ:

Giả sự có một document đơn giản lưu trữ dữ liệu của một người dùng như sau:

> db.users.findOne()

{

“\_id”: ObjectId(“4b253b067525f35f94b60a31”),

“name”: “joe”,

“age”: 30,

“sex”: “male”,

“location”: “Wisconsin”

}

Đây là một document gồm những thông tin căn bản nhất cho một người dùng. Nếu như người dùng muốn lưu trữ thêm mục “favorite book”. Có thể sử dụng biến thay đổi “set”:

> db.users.update({“\_id”: ObjectId(“4b253b067525f35f94b60a31”)},

... {“$set”: {“favorite book”: “war and peace”}})

Bây giờ, dùng find() hoặc findOne() để xem lại kết quả sau khi cập nhật.

> db.users.findOne()

{

“\_id”: ObjectId(“4b253b067525f35f94b60a31”),

“name”: “joe”,

“age”: 30,

“sex”: “male”,

“location”: “Wisconsin”,

“favorite book”: “war and peace”

}

Nếu như người dùng quyết định thiết đặt lại giá trị cho trường “favorite book” với cuốn sách khác, “$set” có thể được sử dụng lần nữa để làm công việc này.

> db.users.update({“name”: “joe”},

... {“$set”: {“favorite book”: “green eggs and ham”}})

“$set” còn có thể thay đổi kiểu dữ liệu của trường.

Ví dụ: Tiếp theo trường hợp trên. Người dùng lại thay đổi giá trị của “favorite book”, nhưng lần này, người dùng muốn sử dụng nhiều quyển sách hơn trong trường này, anh ta có thể thay đổi giá trị của “favorite book” thành một mảng (array) với mỗi giá trị trong mảng là một quyển sách:

> db.users.update({“name”: “joe”},

... {“$set”: {“favorite book”:

... [“cat”s cradle”, “foundation trilogy”, “ender”s game”]}})

Tuy nhiên, nếu như sau đó, người dùng lại nhận ra anh ta không thích đọc sách nữa, anh ta có loại bỏ trường này bằng cách sử dụng biến thay đổi “$unset”

Ví dụ

> db.users.update({“name”: “joe”},

... {“$unset”: {“favorite book”: 1}})

Bây giờ trường favorite book đã được loại bỏ, và document trở lại trạng thái ban đầu.

Ngoài ra, biến thay đổi “$set” còn được dùng để cập nhật, thêm mới các trường và giá trị của các document con trong document cha.

Ví dụ:

> db.blog.posts.findOne()

{

“\_id”: ObjectId(“4b253b067525f35f94b60a31”),

“title”: “A Blog Post”,

“content”: “...”,

“author”: {

“name”: “joe”,

“email”: “joe@example.com”

}

}

Ta có document sở hưu một document “author” là một document con. Ta có thể dùng “$set” để truy cập và cập nhật giá trị trường trong document con này. Trong trường hợp này, ta sẽ cập nhật lại giá trị cho author.name = “joe schmoe”, ta thực hiện như sau:

> db.blog.posts.update({“author.name”: “joe”},

... {“$set”: {“author.name”: “joe schmoe”}})

Và kiểm tra lại kết quả:

> db.blog.posts.findOne()

{

“\_id”: ObjectId(“4b253b067525f35f94b60a31”),

“title”: “A Blog Post”,

“content”: “...”,

“author”: {

“name”: “joe schmoe”,

“email”: “joe@example.com”

}

}

Lưu ý:

Cần phải luôn luôn sử dụng biến thay đổi “$...” để thực hiện thêm, thay đổi, hay xóa các trường. Một sai lầm thường gặp của mọi người khi muốn cập nhật giá trị của trường. Ví dụ: thay đổi giá trị của trường “foo” thành giá trị “bar”, mọi người thường dùng theo cấu trúc như sau:

> db.coll.update(x, {“foo”: “bar”})

Trong đó x là biểu thức xác định document cần update.

Cấu trúc của phương thức trên thực chất không phải kiểu thay đổi giá trị, mà là thay thế document, ý nghĩa của câu trên là thay đổi document được xác định bởi x thành một document mới với cấu trúc của document là {“foo”: “bar”}, những trường cũ trong document x đã bị loại bỏ hoàn toàn.

Để thay thế giá trị của trường “foo” thành “bar” mà không ảnh hưởng đến các trường khác, thực hiện câu lệnh như sau:

> db.coll.update(x,{“$set”: {“foo”: “bar”}})

Cần luôn luôn sử dụng biến thay đổi “$...” để thay đổi các trường một cách riêng lẻ.

##### Tăng và giảm giá trị tự động “$inc”

Biến thay đổi “$inc” được dùng để thay đổi giá trị đã tồn tại hoặc tạo mới nếu trường đó chưa được tồn tại. Nó rất hữu dụng cho để cập nhật lại các giá trị thống kê, bình chọn, hoặc bất cứ thông tin nào có liên quan đến các con số.

Giả sử, chúng ta tạo ra một collection để lưu trữ lại các thông tin của một trò chơi, bao gồm thông tin về trò chơi và người dùng, mà mỗi khi thực hiện lưu lại trò chơi, thì collection sẽ cập nhật lại điểm số mà người dùng đạt được. Khi trò chơi bắt đầu, collection sẽ thêm thông tin trò chơi và tên người chơi tương ứng:

> db.games.insert({“game”: “pinball”, “user”: “joe”})

Mỗi khi trái banh va chạm, trò chơi sẽ tăng điểm của người chơi, giả sử số điểm cơ bản của một lần va chạm là 50 điểm. Ta có thể thiết lập biến thay đổi “$inc” để cập nhật điểm cho người chơi như sau:

> db.games.update({“game”: “pinball”, “user”: “joe”},

... {“$inc”: {“score”: 50}})

Kiểm tra lại kết quả sau khi cập nhật:

> db.games.findOne()

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“game”: “pinball”,

“name”: “joe”,

“score”: 50

}

Trường “Score” chưa được tạo trước đó, nên nó được tạo mới bởi biến thay đổi “$inc” và thiết lập nó tăng lên 50

Giả sử, ta thiết lập chức năng nhận điểm thưởng, sẽ thêm 10000 điểm vào số điểm hiện tại, ta thực hiện như sau:

> db.games.update({“game”: “pinball”, “user”: “joe”},

... {“$inc”: {“score”: 10000}})

Hãy xem kết quả đạt được sau khi thực hiện câu lệnh trên

> db.games.find()

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“game”: “pinball”,

“name”: “joe”,

“score”: 10050

}

Trường “Score” đã tồn tại và được thiết lập ở dạng chữ số, cho nên server đã thêm 10000 điểm vào số điểm hiện tại, và ta có tổng cộng là 10050 điểm.

“$inc” tương tự với “$set”, nhưng nó được thiết đặt để tăng hoặc giảm một giá trị số. “$inc” được dùng với các dạng số integer, long, hoặc double, nếu như trường được sử dụng là bất cứ loại nào khác, server sẽ báo lỗi vì không thể thực hiện được. Có rất nhiều dạng dữ liệu không phải số, như nulls, booleans, strings hay số ở dạng ký tự, ví dụ như:

> db.foo.insert({“count”: “1”})

> db.foo.update({}, {$inc: {count: 1}})

Cannot apply $inc modifier to non-number

Giá trị của biến thay đổi “$inc” buộc phải là số. Sẽ không thể tăng một ký tự hay một chuỗi ký tự, hay bất kỳ dạng dữ liệu không phải chữ số nào. Bạn sẽ nhận được thông báo lỗi từ server cho vấn đề này.

Để thay đổi kiểu dữ liệu cho trường này, cần dùng biến thay đổi “$set” như đã đề cập phía trên.

#### Thay đổi giá trị cho mảng

##### Toán tử “$push”

Đây là một cấu trúc mở rộng của của cấu trúc “$set”, cấu trúc này chỉ được dùng cho kiểu dữ liệu mảng. Ví dụ, bạn không thể thêm vào một đối tượng cho một chữ số, hoặc lấy đi các giá trị trong một chuỗi kí tự, thay vào đó, hãy dùng “$set” và “$inc” để làm việc đó.

“$push” có tác dụng thêm các giá trị vào vị trí chỉ số cuối cùng trong mảng của trường nếu như trường đó đã được tồn tại, hoặc tạo mới nếu nó chưa tồn tại. Ví dụ, thay vì lưu trữ các bài viết của một trang blog, và thêm các giá trị vào trường comments dưới dạng mảng. Ta có thể tiến hành tạo và thêm “comment” bằng câu lệnh sau:

> db.blog.posts.findOne()

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“title”: “A blog post”,

“content”: “...”

}

> db.blog.posts.update({“title”: “A blog post”}, {$push: {“comments”:

... {“name”: “joe”, “email”: “joe@example.com”, “content”: “nice post.”}}})

> db.blog.posts.findOne()

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“title”: “A blog post”,

“content”: “...”,

“comments”: [

{

“name”: “joe”,

“email”: “joe@example.com”,

“content”: “nice post.”

}

]

}

Tiếp theo, nếu như ta muốn thêm một comment khác, ta có thể sử dụng lại “$push”:

> db.blog.posts.update({“title”:”A blog post”}, {$push:{“comments”:

... {“name”: “bob”, “email”: “bob@example.com”, “content”: “good post.”}}})

> db.blog.posts.findOne()

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“title”: “A blog post”,

“content”: “...”,

“comments”: [

{

“name”: “joe”,

“email”: “joe@example.com”,

“content”: “nice post.”

},

{

“name”: “bob”,

“email”: “bob@example.com”,

“content”: “good post.”

}

]

}

##### Toán tử “$ne”

Trong trường hợp bạn muốn thêm giá trị cho mảng, nếu như giá trị đó chưa tồn tại trong mảng, có thể sử dụng toán tử “$ne” (not exist) cho trường hợp này. Ví dụ, thêm một “author” vào một danh sách “citations” chỉ khi “author” này chưa được thêm vào trước đó:

> db.papers.update({“authors cited”: {“$ne”: “Richie”}},

... {$push: {“authors cited”: “Richie”}})

##### Toán tử “$addToSet”

Ta cũng có thể dùng “$addToSet”, với công dụng cũng tương đương với sử dụng “$ne”, nhưng các sử dụng của nó đơn giản hơn. Ví dụ, giả sử ta có 1 document với cấu trúc như sau, ta sẽ thêm địa chỉ emails cho document này.

> db.users.findOne({“\_id”: ObjectId(“4b2d75476cc613d5ee930164”)})

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“username”: “joe”,

“emails”: [

“joe@example.com”,

“joe@gmail.com”,

“joe@yahoo.com”

]

}

Khi ta thử thêm 1 email đã tồn tại và dùng “$addToSet” với việc đó:

> db.users.update({“\_id”: ObjectId(“4b2d75476cc613d5ee930164”)},

... {“$addToSet”: {“emails”: “joe@gmail.com”}})

Kiểm tra lại kết quả

> db.users.findOne({“\_id”: ObjectId(“4b2d75476cc613d5ee930164”)})

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“username”: “joe”,

“emails”: [

“joe@example.com”,

“joe@gmail.com”,

“joe@yahoo.com”,

]

}

Vì email joe@gmail.com đã tồn tại trong trường emails của document, nên “$addToSet” sẽ không thêm nữa, và không có gì thay đổi trong trường hợp này.

Tiếp theo, ta thử 1 trường hợp khác, thêm 1 email mà chưa có trong trường emails, tiếp tục sử dụng “$addToSet”

> db.users.update({“\_id”: ObjectId(“4b2d75476cc613d5ee930164”)},

... {“$addToSet”: {“emails”: “joe@hotmail.com”}})

Kiểm tra lại kết quả

> db.users.findOne({“\_id”: ObjectId(“4b2d75476cc613d5ee930164”)})

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“username”: “joe”,

“emails”: [

“joe@example.com”,

“joe@gmail.com”,

“joe@yahoo.com”,

“joe@hotmail.com”

]

}

Ta thấy, joe@hotmail.com đã được thêm vào trường emails trong document.

Ngoài ra, “$addToSet” còn có thể kết hợp với toán tử “$each” để thêm nhiều giá trị vào một trường đơn trị, điều này không thể thực hiện được khi sử dụng toán tử “$ne/$push”. Ví dụ, chúng ta muốn thêm nhiều emails cùng 1 lúc vào trường emails, tuy nhiên đảm bảo rằng các emails này không được trùng nhau.

> db.users.update({“\_id”: ObjectId(“4b2d75476cc613d5ee930164”)}, {“$addToSet”:

... {“emails”: {“$each”: [“joe@php.net”, “joe@example.com”, “joe@python.org”]}}})

> db.users.findOne({“\_id”: ObjectId(“4b2d75476cc613d5ee930164”)})

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“username”: “joe”,

“emails”: [

“joe@example.com”,

“joe@gmail.com”,

“joe@yahoo.com”,

“joe@hotmail.com”

“joe@php.net”

“joe@python.org”

]

}

##### Toán tử “$pop”

Có một vài cách để loại bỏ document trong mảng. Nếu như muốn loại bỏ các document theo kiểu hàng đợi (queue) hay theo kiểu stack, có thể dùng toán tử “$pop”, các sử dụng như sau:

{$pop: {key: 1}}: dùng để loại bỏ document ở cuối mảng.

{$pop: {key: -1}}: dùng để loại bỏ document ở đầu mảng.

##### Toán tử “$pull”

Đôi khi, ta muốn thực hiện loại bỏ một document dựa vào một mô tả cụ thể, mà không chỉ dựa vào vị trí của chúng. Toán tử “$pull” sẽ giúp cho ta làm việc này, nó sẽ loại bỏ một document trong một mảng, dựa vào một mô tả về document đó. Ví dụ, chúng ta có một danh sách những công việc cần phải hoàn thành được liệt kê sau đây:

> db.lists.insert({“todo”: [“dishes”, “laundry”, “dry cleaning”]})

Nếu như chúng ta làm công việc “laundry” trước, giả sử chúng ta hoàn thành nó, và bây giờ muốn loại bỏ nó ra danh sách vì nó đã được hoàn thành, ta làm như sau:

> db.lists.update({}, {“$pull”: {“todo”: “laundry”}})

Bây giờ hãy kiểm tra lại danh sách “todo”:

> db.lists.find()

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“todo”: [

“dishes”,

“dry cleaning”

]

}

Ta đã không còn công việc “laundry” trong danh sách công việc.

Toán tử “$pull” không chỉ loại 1 document, mà nó sẽ loại tất cả các document phù hợp với điều kiện xác định. Ví dụ ta có một danh sách như sau [1,1,2,1], nếu ta sử dụng “$pull” để loại document 1 ra khỏi danh sách, ta thu được kết quả như sau: [2]

##### Thay đổi thông qua vị trí document

Thay đổi giá trị trong mạng sẽ trở nên rối rắm hơn khi trong mảng có nhiều giá trị document tương tự nhau và chúng ta muốn thay đổi một vài trong những document đó.

Có 2 cách để thay đổi giá trị trong mảng là: truy cập vào vị trí của nó, hoặc sử dụng toán tử vị trí “$”. Chúng ta cũng có thể kết hợp 2 cách này với nhau để tạo ra hiệu quả cao nhất.

Mảng trong MongoDB bắt đầu từ chỉ số 0, và các giá trị trong nó có thể được truy cập bằng chỉ số thông qua thuộc tính trường của nó. Ví dụ, giả sử ta có một document chứa một số document con, ở đây document lưu trữ các thông tin phản hồi của một trang blog cá nhân:

> db.blog.posts.findOne()

{

“\_id”: ObjectId(“4b329a216cc613d5ee930192”),

“content”: “...”,

“comments”: [

{

“comment”: “good post”,

“author”: “John”,

“votes”: 0

},

{

“comment”: “i thought it was too short”,

“author”: “Claire”,

“votes”: 3

},

{

“comment”: “free watches”,

“author”: “Alice”,

“votes”: -1

}

]

}

Nếu có một người “Vote” cho “comment” đầu tiên, ta cần tăng số lượng của trường “votes” của nó lên 1 đơn vị, để thực hiện điều này, ta thực hiện như sau:

> db.blog.update({“post”: post\_id},

... {“$inc”: {“comments.0.votes”: 1}})

Trong nhiều trường hợp, chúng ta không biết được chỉ số document cần thay đổi trong mảng. Để giải quyết vấn đề này, MongoDB hỗ trợ toán tử vị trí “$”, nó sẽ trả về chỉ số vị trí của giá trị tương ứng với điều kiện xác định, và cập nhật lại giá trị đó. Ví dụ, chúng ta có người dùng “author” tên là “John” và ta muốn cập nhật lại với tên là Jim, ta sẽ cập nhật trong trường “comment” của collection “blog” bằng cách sử dụng toán tử vị trí “$”.

> db.blog.update({“comments.author”: “John”},

... {“$set”: {“comments.$.author”: “Jim”}})

Toán tử vị trí “$” cập nhật giá trị tương ứng với điều kiện xác định.

#### Các tham số của phương thức “update”

##### Tham số “upsert”

Tham số “upsert” là một tham số mở rộng của phương thức update, nếu như không có document nào thỏa điều kiện tìm kiếm của phương thức update, một document mới sẽ được tạo ra với các trường được kết hợp từ điều kiện tìm kiếm và giá trị update của phương thức. Ngược lại, nếu như có tồn tại document thỏa điều kiện update, phương thức update sẽ được thực thi như mình thường. “Upsert” tỏ ra rất hữu ích vì nó sẽ loại được những dòng lệnh không cần thiết: làm cho các dòng code trở nên dài dòng và thực thi lâu.

Trở lại một ví dụ trước đó, về việc ghi nhận số lượt truy cập của các trang web của người dùng.

Nếu như không sử dụng upsert, người dùng có thể phải cố gằng tìm đến document có chứa URL của trang web đó và tăng số lượng views lên, tuy nhiên nếu chưa tồn tại document có chứa URL trang web đó, cần tạo mới nó, để thực hiện công việc trên ta cần viết 1 đoạn mã riêng, ví dụ ở đây là chương trình Javascript với tên được lưu là scriptname.js (để thực thi đoạn mã này bằng cách chạy lệnh mongo scriptname.js).

Đoạn mã được viết như sau:

*// Bước 1: kiểm tra tồn tại và lấy kết quả trả về của document*

blog = db.analytics.findOne({url: “/blog”})

*// Bước 2: nếu document tồn tại, thực hiện tăng lượng views*

if (blog) {

blog.pageviews++;

db.analytics.save(blog);

}

*// Bước 3: Ngược lại, nếu không tồn tại, cần tạo document mới*

else {

db.analytics.save({url: “/blog”, pageviews: 1})

}

Nó có nghĩa là, chúng ta đã tạo một đường khi khá rắc rối bao gồm 2 phương thức save() và update(), mà đường đi này được thực thi mỗi khi có lượt truy cập. Nếu như đoạn mã này được thực thi trên nhiều process với một tốc độ nhanh, có thể sẽ xảy ra quá trình xung đột dữ liệu nếu như không được xử lý tốt.

Chúng ta có loại bỏ những rắc rối này bằng cách sử dụng upsert trên 1 hàm duy nhất, đây chính là tham số thứ 3 của phương thức update, đây là một tham số mở rộng thứ 1.

db.analytics.update({“url”: “/blog”}, {“$inc”: {“visits”: 1}}, true)

hoặc rõ ràng hơn

db.analytics.update({“url”: “/blog”}, {“$inc”: {“visits”: 1}}, {upsert:true})

Một trong 2 dòng lệnh trên hoàn toàn tương đương với đoạn mã “scriptname.js” trên, tuy nhiên, đặc biệt là nó thực thi nhanh hơn và là đơn lệnh.

Một document mới được tạo ra dựa trên tham số thứ 1 của phương thức update (điều kiện xác định document) làm nền tảng, và sau đó thay đổi nó theo tham số thứ 2 (giá trị cần thay đổi của document). Ví dụ, nếu như dùng upsert để tạo ra 1 document, và document đó sẽ được cập nhật 1 lần nữa thông qua giá trị cập nhật như sau:

> db.math.remove({})

> db.math.update({“count”: 25}, {“$inc”: {“count”: 3}}, true)

> db.math.findOne()

{

“\_id”: ObjectId(“4b3295f26cc613d5ee93018f”),

“count”: 28

}

Đầu tiên, để đảm bảo collection rỗng, ta thực hiên xóa các phần tử trong collection. “Upsert” sẽ tạo document mới với giá trị “count” là 25 và sau đó tăng nó lên 3, cuối cùng ta được “count là 28”.

Nếu ta không dùng upsert trong trường hợp này, có không có document nào thỏa điều kiện “count” bằng 25, và sẽ không có gì xảy ra sau đó.

Nếu như ta chạy lại câu lệnh:

> db.math.update({“count”: 25}, {“$inc”: {“count”: 3}}, true)

Vì không có document nào phù hợp với điều kiện {count: 25}, nên sẽ có 1 document mới được tạo theo 1 quy trình tương tự trên (“count” là 28).

###### Phương thức “save – upsert”

Phương thức save được dùng để thêm 1 document nếu nó chưa tồn tại trong collection (tương đương phương thức insert), hoặc chỉnh sửa 1 document nếu nó đã tồn tại trong collection (tương đương phương thức update).

Cấu trúc sử dụng của phương thức save như sau

> db.collection.save(x)

Trong đó x là document cần thêm hoặc cập nhật.

Nếu như x có giá trị trường “\_id” đã tồn tại trong collection, phương thức save(x) sẽ hoạt động tương tự phương thức update-upsert, nếu không, nó sẽ hoạt động tương tự hàm insert. Phương thức này hỗ trợ rất tốt cho người lập trình viên giảm được đáng kể thời gian viết mã lệnh, ví dụ ở trường hợp sau:

> var x = db.foo.findOne()

> x.num = 42

42

> db.foo.save(x)

Nếu như không sử dụng phương thức save, ta có thể sử dụng update-upsert với tính năng tương tự nhưng có phần phức tạp hơn:

> db.foo.update({“\_id”: x.\_id}, x)

##### Tham số “Update-Multi” – (Update nhiều document 1 lúc)

Theo mặc định, phương thức update chỉ thực hiện cập nhật lại document đầu tiên mà nó tìm được. Trong trường hợp có nhiều document thỏa điều kiện update, các document sau document đầu tiên thỏa mãn sẽ không thay đổi theo phương thức update. Để thay đổi tất cả các document thỏa điều kiện của phương thức update, ta cần phải gán giá trị cho “multi” là “true”

Phương thức update-multi là một cách tốt về hiệu suất hoạt động và tránh việc phải dùng vòng lặp để update từng document. Ví dụ, giả sữ chúng ta muốn gửi 1 tin nhắm đến tất cả người có ngày sinh vào “10/13/1978”, ta thực hiện như sau:

> db.users.update({birthday: “10/13/1978”},

... {$set: {gift: “Happy Birthday!”}}, false, true)

Tham số multi là tham số mở rộng thứ 2, đứng ở trị trí thứ 4 trong các tham số của phương thức update, theo mặc định của nó sẽ là false.

Nó sẽ thêm trường gift: “Happy Birhday!” vào tất cả các document có ngày sinh nhật là “10/13/1978”

Để xem số document update trong câu lệnh trên, ta có thể chạy dòng lệnh sau

db.runCommand({getLastError: 1})

Ta có, “n” là số lượng các document vừa được cập nhật trong phương thức update gần nhất.

> db.count.update({x: 1}, {$inc: {x: 1}}, false, true)

> db.runCommand({getLastError: 1})

{

“err”: null,

“updatedExisting”: true,

“n”: 5,

“ok”: true

}

“n” bằng 5, tương đương có 5 document đã được cập nhật bằng phương thức “update”.

### Thao tác truy vấn

#### Phương thức truy vấn find()

Phương thức find() là loại truy vấn phổ biến nhất trong MongoDB. Nó trả về tập hợp document trong collection. Cách sử dụng như sau:

> db.collection.find(x)

X: là chuỗi các biểu thức xác định phương thức trả về document.

Những document được xác định trả về phụ thuộc vào tham số của phương thức.

Nếu như x được để trống, thì nó sẽ trả về tất cả những gì có trong collection, mặc định của tham số x sẽ là {} (nghĩa là rỗng), ví dụ như:

> db.c.find({})

Hoặc

> db.c.find()

Hai cách biểu diễn phương thức find() trên hoàn toàn tương tự nhau, nó sẽ trả về mọi thứ trong collection c.

Khi muốn truy vấn dữ liệu với một giá trị cần tìm kiếm. Ví dụ, khi cần tìm một user có giá trị “age” là 27, chúng ta có thể điền điều kiện này vào trong tham số của câu truy vấn.

> db.users.find({“age”: 27})

Nếu như muốn truy vấn dựa trên một giá trị dạng chuỗi. Ví dụ, tìm một user có trường “username” với già trị là “joe”, ta làm như sau:

> db.users.find({“username”: “joe”})

Nếu muốn truy vấn document dựa trên nhiều giá trị trên các trường khác nhau, ta chỉ việc đặt các giá trị thành thứ tự các tham số của phương thức. Ví dụ, ta cần tìm user có trường “age” là 27 và “username” là “joe”, ta làm như sau:

> db.users.find({“username”: “joe”, “age”: 27})

##### Phương thức findone();

Với cách sử dụng và chức năng giống với phương thức find(), phương thức findOne() chỉ có 1 sự khác biệt duy nhất đó là nó sẽ chỉ trả về document đầu tiên mà nó so khớp được, còn cách sử dụng hoàn toàn tương đương.

##### Phương thức printjson, pretty()

Để kết quả trả về được thân thiện phương thức printjson và pretty chuyển đổi đối tượng trả về thành định dạng json thân thiện. Ví dụ

> db.people.find().find(printjson);

> db.people.find().pretty()

#### Truy vấn có điều kiện

##### Truy vấn với những trường trả về cụ thể

Đôi khi, không cần thiết phải trả về tất cả giá trị của nó. Trong những trường hợp đó, bạn cần truyền vào tham số thứ 2 của phương thức find (hoặc findOne), tham số này sẽ xác định những trường cần trả về.

Công dụng của nó sẽ tiết kiệm được lượng dữ liệu trao đổi giữa cilent và server, đồng thời cắt giảm chi phí thời gian cũng như dung lượng bộ nhớ.

Ví dụ, nếu bạn có một collection về user và bạn chỉ quan tâm đến trường “username” và “email”, những trường khác không cần thiết để trả về, trong trường hợp này, bạn có thể trả về những giá tị mong muốn theo các sau đây:

> db.users.find({}, {“username”: 1, “email”: 1})

{

“\_id”: ObjectId(“4ba0f0dfd22aa494fd523620”),

“username”: “joe”,

“email”: “joe@example.com”

}

Có thể thấy, trường “\_id” luôn được mặc định trả về, kể cả khi nó không được liệt kê để trả về,

Bạn hoàn toàn có thể sử dụng các này để loại bỏ những trường được chỉ định trong kết quả trả về của câu truy vấn. Tiếp theo ví dụ trên, để không hiện trường “\_id” ta cần truyền giá trị 0 cho “\_id”

> db.users.find({}, {“\_id”: 0,”username”: 1, “email”: 1})

{

“username”: “joe”,

“email”: “joe@example.com”

}

Hoặc chỉ hiện duy nhất trường “username”:

> db.users.find({}, {“username”: 1, “\_id”: 0})

{

“username”: “joe”,

}

##### Giới hạn khoảng phạm vi giá trị của trường.

“$lt”, “$lte”, “$gt”, và “$gte” là những toán tử so sánh.

“$lt” (<): nhỏ hơn.

“$lte” (<=): nhỏ hơn hoặc bằng.

“$gt” (>): lớn hơn.

“$gte” (<): lớn hơn hoặc bằng.

Những toán tử này có thể được kết hợp với các hàm truy vấn... để giới hạn được phạm vi giá trị tìm kiếm. Ví dụ, để tìm những “users” có độ tuổi nằm trong đoạn [18, 30], ta sử dụng như sau:

> db.users.find({“age”: {“$gte”: 18, “$lte”: 30}})

Những toán tử này cũng có thể áp dụng với kiểu dữ liệu ngày tháng. Ví dụ, ta cần tìm những “users” có ngày đăng ký “registered” nhỏ hơn ngày 01/01/2007, ta làm như sau:

> start = new Date(“01/01/2007”)

> db.users.find({“registered”: {“$lt”: start}})

Tuy nhiên, sẽ dễ bị sót nếu sử dụng những toán tử này cho kiểu dữ liệu này, nếu sử lý không tốt, vì kiểu dữ liệu ngày được lưu với đơn vị chính xác là millisecond, vì vậy cần lưu trữ kiểu dữ liệu hợp lý.

Để có một câu lệnh cho các document có giá trị trường khác với giá trị xác định, cần xử dụng một toán tử khác, “$ne” (“not equal” – không bằng).Ví dụ, nếu như bạn muốn tìm tất cả “users” có tên không phải là “joe”, bạn có thể sử dụng như sau:

> db.users.find({“username”: {“$ne”: “joe”}})

“$ne” có thể sử dụng được cho tất cả kiểu dữ liệu khác nhau.

##### Toán tử “$in - $or”

Trong mongoDB, có 2 cách để dử dụng toán tử với ý nghĩa “Hoặc” trong câu lệnh. Toàn tử “$in” có thể được dùng cho nhiều giá trị trong 1 trường. Toán tử “$or” thì khái quát hơn, nó có thể được dùng cho nhiều giá trị trong nhiều trường khác nhau.

Nếu bạn có nhiều giá trị đề tìm giá trị trong 1 trường, có thể dùng mảng liệt kê các giá trị đó, và dùng toán tử “$in” để so khớp. Ví dụ, trong 1 cuộc sổ số, và ta có trước 3 số là số chiến thắng là 725, 542, và 390. Để so khớp giá trị trong trường “ticket\_no” của các document, ta thực hiện như sau:

> db.raffle.find({“ticket\_no”: {“$in”: [725, 542, 390]}})

Toán tử “$in” rất mềm dẻo, nó có thể sử dụng để so khớp giữa các giá trị khác loại với nhau. Ví dụ, nếu như ta chuyển lược đồ từ lưu trữ số ID sang lưu trữ tên, có rất nhiều dữ liệu bị sót, để tìm “user” với trường “user\_id”:

> db.users.find({“user\_id”: {“$in”: [12345, “joe”]})

Document khớp với câu truy vấn sẽ có giá trị trường “user\_id” bằng 12345, và những document có “user\_id” là “joe”

Nếu toán tử “$in” chỉ sử dụng 1 giá trị duy nhất, nó sẽ tương đương với cách so khớp thông thường. Ví dụ, {ticket\_no: {$in: [725]}} sẽ so khớp tương đương hoàn toàn với{ticket\_no: 725}.

Trái ngược với toán tử “$in” là “$nin”, nó sẽ trả về các document có trường so khớp không có trong sanh sách giá trị so khớp. Ví dụ, tiếp tục ví dụ sổ số, nếu như bạn muốn trả về tất cả người không trúng giải, có thể sử dụng như sau:

> db.raffle.find({“ticket\_no”: {“$nin”: [725, 542, 390]}})

Câu truy vấn sẽ trả về tất cả những người có vé không trúng thưởng.

Ta đã biết cách hoạt động của toàn tử “$in” đối với 1 giá trị so khớp, tuy nhiên, nếu như chúng ta cần tìm tất cả documents có “ticket\_no” là 725 và “winner” là true, trong trường hợp này ta cần sử dụng toán tử “$or”, “$or” sử dụng mảng các sự kiện so khớp, trong trường hợp này, ta dủng như sau:

> db.raffle.find({“$or”: [{“ticket\_no”: 725}, {“winner”: true}]})

“$or” có thể sử dụng bất kỳ điểu kiện so khớp nào, ví dụ, chúng ta muốn so khớp “ticket\_no” với 3 giá trị và “winner” là true, trường hợp này ta dùng toán tử “$in” để làm điểu kiện so khớp:

> db.raffle.find({“$or”: [{“ticket\_no”: {“$in”: [725, 542, 390]}}, {“winner”: true}]})

Đối với toán tử “$and”, cấu trúc sử dụng tương tự như toán tử “$or”. Tuy nhiên nó hơi khác về mặt ý nghĩa.

Toán tử “$or”, sẽ trả về true khi chỉ cần 1 điều kiện so khớp trả về true.

Khi đó, toán tử “$and”, sẽ trả về true khi toàn bộ điều kiện so khớp trả về true, còn lại nó sẽ trả về false.

##### Toán tử “$not”

“$not” có thể áp dụng cho bất kỳ điều kiện so khớp nào. “$mod” có 2 tham số, tham số thứ nhất là số chia, tham số thứ 2 là phần dư, và kết quả trả về phải thỏa mãn là số bị chia của phép tính. Ví dụ, giả sử với phép kiểm chia lấy phần dư “$mod” như sau:

> db.users.find({“id\_num”: {“$mod”: [5, 1]}})

Kết quả sẽ trả về danh sách những users có “id\_num” là 1, 6, 11, 16,…. Tuy nhiên nếu đảo ngược lại câu truy vấn trên bằng cách sử dụng toán tử “$not”, kết quả trả về sẽ là danh sách những “users” có “id\_num” là 2, 3, 4, 5, 7, 8, 9, 10, 12,…

> db.users.find({“id\_num”: {“$not”: {“$mod”: [5, 1]}}})

“$not” có thể trở nên hữu dụng trong những câu lệnh cơ bản để tìm các document không khớp với điều kiện đặt ra.

##### Toán tử điều kiện “$where”

Đôi khi, ta cần thiết lập một số câu lệnh thủ tục làm điều kiện. Những câu lệnh bình thường không thể làm được điều đó, vì vậy toán tử “$where” giúp chúng ta giải quyết được vấn đề này, nó cho phép chúng ta thiết lập các hàm theo định dạng Javascript như một phần trong câu lệnh của MongoDB. Vì thế, câu lệnh gần như có thể làm được tất cả nhờ toán tử “$where”. Ví dụ ta có 2 documents như sau:

db.foo.insert({“apple”: 1, “banana”: 6, “peach”: 3})

db.foo.insert({“apple”: 8, “spinach”: 4, “watermelon”: 4})

Ta thấy rằng, trong document thứ 2, giá trị của “spinach” và “watermelon” là đều bằng nhau và bằng 4, ví dụ như chúng ta muốn lấy điều kiện này để trả về document, chúng ta sẽ không có toán tử $ nào hỗ trợ trực tiếp để giải quyết trường hợp này, vì vậy, chúng ta có thể sử dụng toán tử “$where” để làm việc này, ta làm như sau:

db.foo.find({“$where”: function () {

for (var current in this) {

for (var other in this) {

if (current != other && this[current] == this[other]) {

return true;

}

}

return false;

}});

Nếu hàm thiết đặt trả về giá trị true, document sẽ được trả về trong kết quả của câu truy vấn.

Có nhiều cách sử dụng toán tử “$where” nhưng vẫn trả về kết quả tương đương nhau. Ví dụ:

> db.foo.find({“$where”: “this.x + this.y == 10”})

> db.foo.find({“$where”: “function() {return this.x + this.y == 10;}”})

Toán tử “$where” được khuyên là không nên dùng nếu như không cần thiết, vì nó hoạt động chậm hơn các câu lệnh thông thường. Những document buộc phải chuyển đổi từ đình dạng BSON sang JavaScript để thực hiện với toán tử “$where”. Các chỉ số “Index” cũng không có tác dụng đối với các hàm được sử dụng với toán tử “$where”

#### Truy vấn chung chung

Những truy vấn chung chung tỏ ra rất hữu ích đối với những định dạng dạng chuỗi. Ví dụ, nếu như bạn muốn truy vấn rất cả những user có tên Joe hoặc joe, có thể sử dụng truy vấn này như sau:

> db.users.find({“name”: /joe/i})

Đối với những điều kiện truy vấn với điều kiện truy vấn là những kí tự có thể có hoặc không. Ví dụ, tìm tất cả những user có tên là joe hoặc joey, ta có thể sử dụng truy vấn như sau:

> db.users.find({“name”: /joey?/i})

MongoDB sử dụng thư viện PCRE (Perl Compatible Regular Expression) cho các toán tử truy vấn chung chung. Tất cả các cú pháp hợp lệ trong thư viện này, điều hợp lệ trong MongoDB.

#### Kiểu mảng

Cấu trúc cho kiểu dữ liệu mảng khá đơn giản. Kiểu mảng hầu như có thể áp dụng đối với tất cả các kiểu dữ liệu, nó sẽ duyệt và xử lý từng phần tử trong mảng. Ví dụ, nếu có 1 mảng danh sách các loại trái cây như sau:

> db.food.insert({“fruit”: [“apple”, “banana”, “peach”]})

Câu lệnh

> db.food.find({“fruit”: “apple”})

> db.food.find({“fruit”: “banana”})

> db.food.find({“fruit”: “peach”})

3 câu lệnh này sẽ hợp lệ và được thực thi trả về document mà ta vừa thêm trước đó. Chúng ta cũng có kết hợp viết cấu trúc khác như sau

> db.food.find({“fruit”:”apple”,”fruit”:”banana”,”fruit”:”peach”})

Câu truy vấn cũng sẽ trả về document ta vừa thêm trước đó.

##### Toán tử $all

Nếu như bạn cần tìm khớp những document với điều kiện nhiều hơn một giá trị, khi đó toán tử “$all” sẽ phát huy tác dụng. Nó cho phép so khớp 1 mảng danh sách con đối với đối tượng là danh sách cha. Ví dụ, giả sử chúng ta tạo 1 collection với 3 phần tử như sau:

> db.food.insert({“\_id”: 1,”fruit”: [“apple”,”banana”,”peach”]})

> db.food.insert({“\_id”: 2,”fruit”: [“apple”,”kumquat”,”orange”]})

> db.food.insert({“\_id”: 3,”fruit”: [“cherry”,”banana”,”apple”]})

Sau đó, ta có thể tìm tất cả documents có đồng thời cả 2 phần tử “apple” và “banana” bằng việc sử dụng toán tử “$all” như sau:

> db.food.find({fruit: {$all: [“apple”, “banana”]}})

{“\_id”: 1, “fruit”: [“apple”, “banana”, “peach”]}

{“\_id”: 3, “fruit”: [“cherry”, “banana”, “apple”]}

Thứ tự là không quan trọng.

Khi sử dụng 1 phần tử với toán tử “$all” nó sẽ tương đương với toán tử “$all” sẽ trở nên vô nghĩa. Ví dụ, trong trường hợ trên, nếu như ta sử dụng biểu thức so khớp

> db.food.find({fruit: {$all: [“apple”]})

Nó sẽ tương đương với câu lệnh sau:

> db.food.find({fruit: “apple”})

Chúng ta sử dụng để so khớp chính xác một document. Tuy nhiên, nếu so khớp không đúng nếu như các phần tử bị thiếu, dư thừa, hoặc sai thứ tự. Ví dụ, câu lệnh truy vấn sau sẽ trả về document đầu tiên trong ví dụ trên.

> db.food.find({“fruit”: [“apple”, “banana”, “peach”]})

{“\_id”: 1, “fruit”: [ “apple”, “banana”, “peach” ]}

Nhưng những câu sau đây thì không

> db.food.find({“fruit”: [“apple”, “banana”]})

> db.food.find({“fruit”: [“banana”, “apple”, “peach”]})

Nếu chúng ta muốn xác định chính xác phần tử trong mảng so khớp, chúng ta có thể xác định chính xác bằng cách sử dụng chỉ số của phần tử so khớp.

Cấu trúc mẫu là: key.index.

Ví dụ: Tìm document có phần tử tứ 3 trong mảng của trường fruit, câu lệnh như sau:

> db.food.find({“fruit.2”: “peach”})

Mảng dùng chỉ số bắt đầu là 0, vì vậy câu lệnh trên sẽ trả về document với giá trị thứ 3 trong mảng có giá trị là “peach”.

##### Toán tử “$size”

Một trong những điều kiện so khớp hữu dụng đối với kiểu mảng là toán tử “$size”, nó cho phép chúng ta xác định document thông qua số lượng xác định phần của của trường. Dưới đây là ví dụ:

> db.food.find({“fruit”: {“$size”: 3}})

“$size” không thể kết hợp với bất kì toán tử $ nào khác (ví dụ như “$gt”). Tuy nhiên, câu truy vấn có thể được hoàn thành bằng cách thêm 1 trường “size” trong document. Sau đó, mỗi khi thêm 1 giá trị vào trong mảng, tăng giá trị của trường “size” lên 1 đơn vị. Ví cụ, nếu như theo mặc định, phương thức update để giá trị của mảng được tiến hành như sau:

> db.food.update({“$push”: {“fruit”: “strawberry”}})

Nó có thể được thay đổi đơn giản như sau:

> db.food.update({“$push”: {“fruit”: “strawberry”}, “$inc”: {“size”: 1}})

Toán tử “$inc” được thực thi rất nhanh, vì vậy nó sẽ không ảnh hưởng đến hiệu suất và tốc độ của phương thức update này. Lưu trữ document như vậy sẽ cho phép chúng ta có thể thực hiện câu truy vấn như sau:

> db.food.find({“size”: {“$gt”: 3}})

Câu lệnh sẽ trả về cho chúng ta những document có số lượng phần tử của trường fruit lớn hơn 3 phần tử.

##### Toán tử $slice

Toán tử “$slice” có thể được sử dụng để trả về các phần tử con trong mảng của 1 trường.

Ví dụ, giả sử chúng ta có document quản lý trang những lượt đăng tải comment của 1 trang blog và chúng ta muốn trả về 10 comment đầu tiên trong trường comment. Ta thực hiện như sau:

> db.blog.posts.findOne(criteria, {“comments”: {“$slice”: 10}})

Ngược lại, nếu như chúng ta muốn trả về 10 comment cuối cùng, ta thực hiện như sau:

> db.blog.posts.findOne(criteria, {“comments”: {“$slice”: -10}})

“$slice” cũng có thể trả về các phần tử ở giữa bằng cách đưa toán tử 2 tham số là vị trí bắt đầu duyệt và số lượng phần tử trả về. Ví dụ:

> db.blog.posts.findOne(criteria, {“comments”: {“$slice”: [23, 10]}})

Câu lệnh trên sẽ trả về các phần tử comment thứ 24 đến 34. Nếu như số lượng phần tử nhỏ hơn 34, nó sẽ trả về nhiều nhất có thể.

Khi sử dụng toán tử “$slice” để so khớp cho một mảng giá trị thuộc 1 trường cụ thể, thì những trường khác vẫn được trả về, mặc dù nó không được đề cập đến. Ví dụ, ta có một document như sau:

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“title”: “A blog post”,

“content”: “...”,

“comments”: [

{

“name”: “joe”,

“email”: “joe@example.com”,

“content”: “nice post.”

},

{

“name”: “bob”,

“email”: “bob@example.com”,

“content”: “good post.”

}

]

}

Và toán tử “$slice” được sử dụng như sau:

> db.blog.posts.findOne(criteria, {“comments”: {“$slice”: -1}})

{

“\_id”: ObjectId(“4b2d75476cc613d5ee930164”),

“title”: “A blog post”,

“content”: “...”,

“comments”: [

{

“name”: “bob”,

“email”: “bob@example.com”,

“content”: “good post.”

}

]

}

Kết quả trả về bao gồm cả trường “title” và “content” vẫn được trả về, mặc dù chúng không được xác định trong trường của câu truy vấn.

#### Các câu lệnh dành cho các documents con

Có 2 cách để thiết lập câu lệnh cho các documents con: đó là xử lý cho tất cả các documents con hoặc xử lý cho từng document con.

Câu lệnh cho từng document hoạt động như những câu lệnh bình thường. Ví dụ, nếu chúng ta có một document có cấu trúc như sau:

{

“name”:

{

“first”: “Joe”,

“last”: “Schmoe”

},

“age”: 45

}

Chúng ta có thể thiết lập câu lệnh trả về có tên là Joe Schmoe như sau:

> db.people.find({“name”: {“first”: “Joe”, “last”: “Schmoe”}})

Tuy nhiên, với câu lệnh trên, nếu ta thêm 1 trường middle, vào document name, thì câu truy vấn sẽ không hoạt động, nó không tìm thấy document có đúng chính xác 2 trường (“first” và “last”) sau khi thêm trường middle.

Chúng ta cũng có một câu truy vấn với 1 chút sự thay đổi nhưng các hoạt động thì vẫn cho ra kết quả tương đương:

> db.people.find({“name.first”: “Joe”, “name.last”: “Schmoe”})

Đối với câu truy vấn này, nếu như ta thêm 1 trường middle vào document, thì câu truy vấn này vẫn hoạt động và trả về kết quả như trên. Tuy nhiên, đặt vào tình huống lúc nãy, nếu ta thêm trường middle vào document name, thì câu truy vấn vẫn hoạt động. Đầu tiên, câu truy vấn này sẽ so khớp trường là first, trả về document nếu đúng, nếu sau so khớp tiếp trường last trong document name, nếu đúng trả về document, ko cần quan tâm đến các trường khác của document này.

Ở một ví dụ phức tạp hơn, chúng ta cũng sẽ thấy được sự khác nhau giữa 2 cách sử dụng này. Giả sử ta có 1 document blog với cấu trúc sau:

{

“content”: “...”,

“comments”: [

{

“author”: “joe”,

“score”: 3,

“comment”: “nice post”

},

{

“author”: “mary”,

“score”: 6,

“comment”: “terrible post”

}

]

}

Bây giờ chúng có câu lệnh sau:

> db.blog.find({“comments”:{“author”:”joe”,”score”:{“$gte”: 5}}})

Câu lệnh so khớp trên phải so khớp đúng toàn bộ các giá trị chính xác, và đối với trường hợp này nó sẽ không trả về giá trị.

Ta thử với câu lệnh ở dạng khác, ta có như sau:

db.blog.find({“comments.author”:”joe”,”comments.score”:{“$gte”: 5}})

{

“\_id”: ObjectId(“54607fdf0c3dd65dd9531ec2”),

“content”: “...”,

“comments”: [

{

“author”: “joe”,

“score”: 3,

“comment”: “nice post”

},

{

“author”: “mary”,

“score”: 6,

“comment”: “terrible post”

}

]

}

Câu lệnh trên đã hoạt động, nhưng kết quả trả về là không đúng. Nó trả về document có “author”: “joe” vì so khớp đúng với điều kiện so khớp “comments.author”:”joe” và trả về document có “score”: 6 khi so khớp đúng với “comments.score”:{“$gte”: 5}.

Để khắc phục điều này, MongoDB hỗ trợ ta các nhóm các điều kiện xác định bằng cách sử dụng toàn tử “$elemMatch”.

db.blog.find({“comments”: {“$elemMatch”: {“author”: “joe”, “score”: {“$gte”: 5}}}})

“elemMatch” cho phép chúng ta nhóm các điều kiệm lại với nhau, một document được trả về phải thỏa mãn tất cả các điều kiện đó.

#### Con trỏ

Thông thường, bên phía server, các database khi trả về kết quả thông qua phương thức find là sử dụng đến con trỏ. Bên phía client, chúng ta cũng có thể sử dụng cách thức hoạt động của con trỏ để duyệt lại các duyệt và xử lý các kết quả trả về này lại. Chúng ta có thể giới hạn lại số lượng kết quả, bỏ qua một vài phần tử của kết quả, sắp kết kết quả trả về theo bất kỳ điều kiện nào.

Để tạo ra một con trỏ, trước hết ta cần thêm 1 vài documents vào collection. Thực hiện truy vấn, sau đó, gán kết quả này đến một biến toàn cục (sử dụng từ khóa “var” để thực hiện điều này). Ví dụ, ta có một đoạn câu lệnh thực hiện điều này. Đầu tiên, chúng ta tạo một collection và sau đó thêm 100 document mẫu vào trong collection này, tiếp đó lưu trữ kết quả của câu lệnh truy vấn vào biến toàn cục cursor:

> for(i=0; i<100; i++) {

... db.c.insert({x: i});

...}

> var cursor = db.collection.find();

Để duyệt các giá trị trong con trỏ này ta dùng phương thức “next” và phương thức “hasNext” để duyệt. Câu lệnh vòng lặp được thực hiện như sau:

> while (cursor.hasNext()) {

... obj = cursor.next();

... // do stuff

...}

Câu lệnh cursor.hasNext() kiểm tra xem phần tử phía sau nó, và cursor.next(), chỉ định cho con trỏ đi đến vị trí tiếp theo, và thực hiện các thiết đặt của câu lệnh sau đó (“do stuff”).

Ngoài vòng lặp while như trên, ta cũng có thể dùng vòng vặp forEach đề xử lý:

> var cursor = db.people.find();

> cursor.forEach(function(obj) {

... // do stuff

... print(obj.name);

...});

Biến con trỏ còn cho phép chúng ta thêm các điều kiện truy vấn trước khi trả về kết quả. Ví dụ ta có các câu lệnh sau:

> var cursor = db.foo.find().sort({“x”: 1}).limit(1).skip(10);

> var cursor = db.foo.find().limit(1).sort({“x”: 1}).skip(10);

> var cursor = db.foo.find().skip(10).limit(1).sort({“x”: 1});

Với kết quả trả về, ta vẫn dùng phương thức “hasNext” và “Next” để duyệt như bình thường. không có sự khác biệt.

##### Các điều kiện truy vấn hay dùng:

Những điều kiện truy vấn bằng con trỏ phổ biến nhất là giới hạn kết quả trả về (Limits), bỏ qua một số lượng trước đó (Skips), và sắp xếp (Sorts). Tất cả những điều kiện này cần được thêm trước khi gửi yêu cầu đến server.

Phương thức giới hạn (Limit): Để cài đặt phương thức limit, ta chỉ việc nối thêm phương thức limit() vào sau phương thức find(). Ví dụ, chỉ trả về 3 document, ta thực hiện như sau:

> db.c.find().limit(3)

Nếu collection có ít hơn 3 documents, câu lệnh sẽ trả về toàn bộ số documents đó

Phương thức bỏ qua (Skip):

Cách cài đạt phương thức skip() cũng tương tự phương thức limit(), ví dụ:

> db.c.find().skip(3)

Câu lệnh sẽ bỏ qua 3 document đầu tiên và trả về những document còn lại sau đó. Nếu như collection có ít hơn 3 documents, thì câu lệnh sẽ không trả về document nào cả.

Phương thức sắp xếp (Sort):

Phương thức sort cần đối số: mỗi đối số là cặp giá trị trường và giá trị sắp xếp. Giá trị sắp xếp chính là chiều sắp xếp của trường. Giá trị là 1, phương thức sẽ sắp xếp theo thứ tự tăng dần của trường giá trị cho trước, ngược lại, giá trị là -1, phương thức sẽ sắp xếp theo thứ tự giảm dần của trường cho trước. Nếu như phương thức có nhiều đối số, kết quả sẽ sắp xếp theo thứ tự trước sau. Ví dụ, chúng ta cần sắp xếp kế quả trả về theo “username” tăng dần và “age” giảm dần, ta thiết đặt như sau:

> db.c.find().sort({username: 1, age: -1})

Ngoài ra, chúng ta còn có thể kết hợp 3 phương thức trên lại với nhau. Ví dụ trường hợp phân trang các kết quả trả về, giả sử trường hợp chạy trên một hệ thống lưu trữ các tập tin và muốn liệt kê các tập tin định dạng mp3, muốn hiển thị 50 kết quả trả về trong mỗi trang, và được sắp xếp theo giá giảm dần. ta thực hiện như sau:

> db.stock.find({“desc”: “mp3”}).limit(50).sort({“price”: -1})

Câu lệnh trên được trả vể kết quả cho trang đầu tiên. Sau đó, nếu chúng ta muốn xem các kết quả cho trang thứ 2, chúng ta cần phải bỏ qua 50 documents ở trang đầu để đi đến 50 documents tiếp theo, ta thực hiện như sau:

> db.stock.find({“desc”: “mp3”}).limit(50).skip(50).sort({“price”: -1})

Tuy nhiên, nếu bỏ qua (skip) một số lượng lớn tốc độ sẽ bị ảnh hưởng, vì vậy, nó không được khuyên dùng trong những trường hợp đặc biệt cần thiết.

#### Những mẹo để tránh dùng “skip” với số lượng lớn

Như đã đề cập trước đó, sử dụng skip với số lượng nhỏ là ổn. Tuy nhiên, đối với số lượng lớn, phương thức “skip” thực thi sẽ chậm (điều này không chỉ xảy ra với mỗi MongoDB) và điều này nên được tránh. Vì vậy, để làm được điều này, chúng ta có một số mẹo để tránh việc sử dụng phương thức “skip” với số lượng lớn, có thể xử lý câu lệnh tiếp theo dựa trên kết quả của câu lệnh trước nó.

##### Phân trang kết quả không dùng “skip”

Cách đơn giản nhất để phân trang là sử dụng “limit” để trả về trang đầu tiên và sau đó, sử dụng phương thức “skip” để trả về các trang sau đó:

> var page1 = db.foo.find(criteria).limit(100)

> var page2 = db.foo.find(criteria).skip(100).limit(100)

> var page3 = db.foo.find(criteria).skip(200).limit(100)

...

Tuy nhiên, tùy vào câu lệnh, chúng ta có thể tìm ra cách để phân trang mà không cần sử dụng đến phương “skip”. Ví dụ, giả sử chúng ta muốn hiển thị các document theo thứ tự giảm dần của trường “date”. Ta có thể lấy kết quả của trang thứ 1 như sau:

> var page1 = db.foo.find().sort({“date”: -1}).limit(100)

Sau đó, chúng ta có thể dùng giá trị ở trường “date” ở document cuối cùng ở trang trước làm cơ sở để trả về kết quả cho trang tiếp theo:

var latest = null;

// display first page

while (page1.hasNext()) {

latest = page1.next();

display(latest);

}

// get next page

var page2 = db.foo.find({“date”: {“$gt”: latest.date}});

page2.sort({“date”: -1}).limit(100);

Bây giờ, câu lệnh đã không còn có phương thức “skip”, ta có thể thực hiện tương tự đối với các trang tiếp theo sau đó.

##### Tìm document ngẫu nhiên.

Một trong những vấn đề phổ biến là làm sao để có thể lấy được một document ngẫu nhiên trong collection. Một cách đơn giản nhất đó là đếm số lượng documents trong collection và sau đó truy suất, bỏ qua một số lượng ngẫu nhiên được lấy từ 0 đến số lượng document trong collection:

> var total = db.foo.count()

> var random = Math.floor(Math.random()\*total)

> db.foo.find().skip(random).limit(1)

Cách trên tuy hoạt động chính xác, nhưng không hiệu quả bằng cách này, vì nếu random là một số ngẫu nhiên lớn, thì hiệu suất hoạt động sẽ không tốt, và sẽ tốn nhiều thời gian để thực thi.

Để tránh điều này, mẹo là ta sẽ thêm 1 trường mở rộng ngẫu nhiên vào mỗi document, giá trị của biến này là Math.random() (phương thức này sẽ tạo một số thực ngẫu nhiên từ 0 đến 1)

> db.people.insert({“name”: “joe”, “random”: Math.random()})

> db.people.insert({“name”: “john”, “random”: Math.random()})

> db.people.insert({“name”: “jim”, “random”: Math.random()})

Bây giờ, ta có thể chọn một document ngẫu nhiên bằng cách, dùng 1 biến có chứa giá trị ngẫu nhiên, sau đó sử dụng biến này làm điều kiện so khớp lớn hơn hay nhỏ hơn đối với trường so khớp ngẫu nhiên với trường mà ta vừa thêm vào cho mỗi document, ta thực hiện như sau:

> var random = Math.random()

> result = db. people.findOne({“random”: {“$gt”: random}})

Nếu như không có trường nào có giá trị lớn hơn biến random, vì đặt trường hợp random là 1 số ngẫu nhiên lớn (~1), ta chỉ việc chuyển điều kiện so khớp thì “$gt” sang “$lt”, ta thực hiện thêm câu lệnh sau:

> if (result == null) {

... result = db. people.findOne({“random”: {“$lt”: random}})

...}

Để câu lệnh được thực thi nhanh, ta có thể thêm chỉ số index vào trường này:

> db.people.ensureIndex({“profession”: 1, “state”: 1, “random”: 1})

### Làm việc với chỉ số (Index)

Để tăng tốc độ xử lý của các câu lệnh, người ta thường đánh thêm chỉ số index vào trong các trường, Tuy nhiên, với một dữ liệu lớn thì việc đánh chỉ số cáo cho document cho toàn bộ collection tốn khoảng vài phút.

Giả sử ta câu lệnh cho 1 trường của document như sau:

> db.people.find({“username”: “mark”})

Khi chỉ có một trường được sử dụng trong câu lệnh, trường này có thể được đánh chỉ số index để tăng tốc độ xử lý của câu lệnh. Trong trường hợp này, chúng ta có thể tạo index cho trường “username”. Để làm đượ điều này, ta thực hiện như sau:

> db.people.ensureIndex({“username”: 1})

Chỉ số của một trường chỉ thêm được một lần cho collection. Nếu chung ta tạo một lần nữa với 1 index tương tự, sẽ không có điều gì xảy ra nữa.

Chỉ số Index của một trường sẽ khiến cho câu lệnh của nó chạy nhanh hơn, tuy nhiên, không phải tất cả các câu lệnh đều nhanh hơn, mặc dù nó có chứa chỉ số Index. Ví dụ, câu lệnh dưới đây không nhanh hơn với chỉ số Index được tạo phía trên.

> db.people.find({“date”: date1}).sort({“date”: 1, “username”: 1})

Với câu lệnh này, mặc dù trường “username” đã sở hữu index, nhưng “username” lại nằm ở phương thức “sort” cùng với trường “date” nên index không phát huy được tác dụng của nó. Đầu tiên, server cần phải duyệt tất cả các document có trong collection để tìm thấy điều kiện so khớp của “date” trong phương thức “find”, vì trường “date” không có Index, điều này sẽ được thực khi rất chậm nếu số lượng document trong collection lớn.

Để giải quyết được tình trạng trên, chúng ta nên tạo nên những chỉ số index bao gồm tất cả các trường trong câu lệnh. Trong trường hợp này, để tối ưu câu truy suất trên, chúng ta cần tạo index bao gồm cả “date” và “username” như sau:

> db.ensureIndex({“date”: 1, “username”: 1})

Các đối số truyền vào phương thức tạo index ensureIndex() cũng tương tự như đối số truyền vào phương thức sort(), giá trị của mỗi trường là 1 hoặc -1, tùy vào chiều tăng giảm của các giá trị trường mà thiết đặt, 1 là tăng dần, -1 là giảm dần.

Nếu như chúng ta có nhiều hơn 1 trường thể đánh chỉ mục index, ta cần nghĩ đến thứ tư ưu tiên sắp xếp, ví dụ ta có 1 collection như sau:

{“\_id”: ..., “username”: “smith”, “age”: 48, “user\_id”: 0}

{“\_id”: ..., “username”: “smith”, “age”: 30, “user\_id”: 1}

{“\_id”: ..., “username”: “john”, “age”: 36, “user\_id”: 2}

{“\_id”: ..., “username”: “john”, “age”: 18, “user\_id”: 3}

{“\_id”: ..., “username”: “joe”, “age”: 36, “user\_id”: 4}

{“\_id”: ..., “username”: “john”, “age”: 7, “user\_id”: 5}

{“\_id”: ..., “username”: “simon”, “age”: 3, “user\_id”: 6}

{“\_id”: ..., “username”: “joe”, “age”: 27, “user\_id”: 7}

{“\_id”: ..., “username”: “jacob”, “age”: 17, “user\_id”: 8}

{“\_id”: ..., “username”: “sally”, “age”: 52, “user\_id”: 9}

{“\_id”: ..., “username”: “simon”, “age”: 59, “user\_id”: 10}

Nếu ta thiết lập index với các thông số như sau {“username”: 1, “age”: -1}. MongoDB sẽ tổ chức lại cho chúng ta như sau:

{“\_id”: ..., “username”: “jacob”, “age”: 17, “user\_id”: 8}

{“\_id”: ..., “username”: “joe”, “age”: 36, “user\_id”: 4}

{“\_id”: ..., “username”: “joe”, “age”: 27, “user\_id”: 7}

{“\_id”: ..., “username”: “john”, “age”: 36, “user\_id”: 2}

{“\_id”: ..., “username”: “john”, “age”: 18, “user\_id”: 3}

{“\_id”: ..., “username”: “john”, “age”: 7, “user\_id”: 5}

{“\_id”: ..., “username”: “sally”, “age”: 52, “user\_id”: 9}

{“\_id”: ..., “username”: “simon”, “age”: 59, “user\_id”: 10}

{“\_id”: ..., “username”: “simon”, “age”: 3, “user\_id”: 6}

{“\_id”: ..., “username”: “smith”, “age”: 48, “user\_id”: 0}

{“\_id”: ..., “username”: “smith”, “age”: 30, “user\_id”: 1}

Trường “username” được sắp xếp theo thứ tự alphabet tăng dần, nếu như có một nhóm tên trùng nhau, thì “age” sẽ được sắp xếp giảm dần. Sự sắp xếp tăng dần hay giảm dần phụ thuộc vào thông số được thiết lập trong câu lệnh tạo Index.

Chỉ số index cho 2 trường “username” và “age” cũng làm cho câu lệnh của một trường “username” nhanh hơn.

Theo nguyên tắc, nếu chỉ số index có N trường, nó sẽ giúp cho các cậu lệnh truy vấn có các tiền tố của index chạy nhanh hơn. Ví dụ, nếu như ta có chỉ số index có các trường {“a” : 1, “b” : 1, “c” : 1, ..., “z” : 1}, có cũng có chức năng như các index {“a” : 1}, {“a” : 1, “b” : 1}, {“a” : 1, “b” : 1, “c” :1}, hoặc tương tự là tiền tố của index được tạo. Nhưng các câu lệnh có các index có các trường không theo đúng thứ tự {“b” : 1}, {“a” : 1, “c” :1}, hoặc tương tự sẽ không có tác dụng (không là tiền tố của index được tạo). Chỉ có các index là tiền tố của index được tạo mới có tác dụng.

Nhược điểm của việc sử dụng index là sẽ phát sinh chi phí trong các trình insert, update, remove. Điều này xảy ra là do database không chỉ thực hiện các câu lệnh trên mà còn phải thực hiện cập nhật lại index sau khi thực thi các câu lệnh trong collection.

#### Chỉ số index cho các document con

Chỉ số index có thể được tạo cho các trường của các document con, cách tạo cũng tương tự đối với những trường thông thường. Ví dụ, nếu chúng ta muốn tìm những bình luận trong document blog theo ngày tháng, ta co thể tạo chỉ số index cho trường “date” trong document con của document “comments”, ta thực hiện như sau:

> db.blog.ensureIndex({“comments.date” : 1})

Chỉ số index của trường trong document được xếp ngang hàng với index của doccument cha.

#### Khai báo tên cho Index

Với mỗi index trong document đều có những chuỗi ký tự “name” lưu tên, và những chuỗi ký tự không trùng nhau. Nó được server dùng để xác định chính xác index cần xử lý. Theo mặc định, chỉ số index được sử dụng tên như sau:

keyname1\_dir1\_keyname2\_dir2\_...\_keynameN\_dirN

trong đó keynameX là tên của trường và dirX là chiều chỉ số index của trường tương ứng (1 hoặc -1). Sẽ rất có sử dụng, tác với chỉ số index với chuỗi tên mặc định như vậy, vì thế, chúng ta có thể thiết đặt thông số này trong đối số của hàm ensureIndex như sau:

> db.foo.ensureIndex({“a” : 1, “b” : 1, “c” : 1, ..., “z” : 1}, {“name” : “alphabet”})

Với tên được người sử dụng đặt, sẽ dễ dàng thực thi hơn thông qua tên của Index do chính người dùng đặt.

#### Giá trị duy nhất (Unique Index)

Unique đảm bảo rằng với mỗi giá trị của trường được thêm, với mỗi document trong collection thì giá trị đó là duy nhất. Ví dụ, nếu chúng ta muốn đảm bảo rằng, không có giá trị trùng lặp trong trường “username” trong các documents trong collection “people”, ta thực hiện tạo unique index như sau:

> db.people.ensureIndex({“username” : 1}, {“unique” : true})

Theo mặc định, khi thêm 1 document, MongoDB kiểm tra chỉ cần câu lệnh hợp lệ cú pháp, nó sẽ được thêm vào. Tuy nhiên sau khi thêm unique index, trước mỗi khi thêm 1 document, MongoDB sẽ kiểm tra xem trên những giá trị trên unique index có bị trùng lắp hay không, nếu có thì báo lỗi cho người sử dụng, điều này hoạt động tương tự với trường “\_id” collection.

#### Index loại bỏ trùng lắp (Dropping Duplicate)

Khi tạo index cho một collection đã tồn tại trước đó, có thể có những giá trị bị trùng lắp. Nếu có bất kì trường trong unique index được tạo có giá trị bị trùng lắp, câu lệnh sẽ báo lỗi, khi đó, có thể bạn muốn loại bỏ đi tất cả các document có trường giá trị bị trùng lắp, việc làm này có thể được thực hiện một cách tự động. Thông số “dropDups” trong câu lệnh “ensureIndex” có thể giúp chúng ta làm việc này, nó sẽ giữ lại document đầu tiên hợp lý, sau đó, nó sẽ loại bỏ bất kỳ các document có giá trị trường bị trùng lắp trước đó:

> db.people.ensureIndex({“username” : 1}, {“unique” : true, “dropDups” : true})

Cách này có thể gây thất thoát dữ liệu, vì vậy cần xem xét kỹ nếu xử lý trên các dữ liệu quan trọng.

## Một số ví dụ câu lệnh được ánh xạ từ SQL

Dưới đây là một số ví dụ về các cách sử dụng các phương thức khác nhau ứng với nhiều trường hợp khác nhau:

|  |  |
| --- | --- |
| **SQL Statement** | **Mongo Statement** |
| INSERT INTO USERS VALUES(3,5) | db.users.insert({a:3,b:5}) |
|  |  |
| SELECT a,b FROM users | db.users.find({}, {a:1,b:1}) |
| SELECT \* FROM users | db.users.find() |
| SELECT \* FROM users WHERE age=33 | db.users.find({age:33}) |
| SELECT a,b FROM users WHERE age=33 | db.users.find({age:33}, {a:1,b:1}) |
| SELECT \* FROM users WHERE age=33 ORDER BY name | db.users.find({age:33}).sort({name:1}) |
| SELECT \* FROM users WHERE age>33 | db.users.find({age:{$gt:33}}) |
| SELECT \* FROM users WHERE age!=33 | db.users.find({age:{$ne:33}}) |
| SELECT \* FROM users WHERE name LIKE “%Joe%” | db.users.find({name:/Joe/}) |
| SELECT \* FROM users WHERE name LIKE “Joe%” | db.users.find({name:/^Joe/}) |
| SELECT \* FROM users WHERE age>33 AND age<=40 | db.users.find({“age”:{$gt:33,$lte:40}}) |
| SELECT \* FROM users ORDER BY name DESC | db.users.find().sort({name:-1}) |
| SELECT \* FROM users WHERE a=1 and b=“q” | db.users.find({a:1,b:”q”}) |
| SELECT \* FROM users LIMIT 10 SKIP 20 | db.users.find().limit(10).skip(20) |
| SELECT \* FROM users WHERE a=1 or b=2 | db.users.find( {$or: [ {a: 1}, {b: 2} ]} ) |
| SELECT \* FROM users LIMIT 1 | db.users.findOne() |
| SELECT order\_id FROM orders o, order\_line\_items li WHERE li.order\_id=o.order\_id AND li.sku=12345 | db.orders.find({“items.sku”:12345},{\_id:1}) |
| SELECT customer.name FROM customers,orders WHERE orders.id=“q179” AND orders.custid=customer.id | var o = db.orders.findOne({\_id:”q179”});  var name = db.customers.findOne({\_id:o.custid}) |
|  |  |
| SELECT DISTINCT last\_name FROM users | db.users.distinct(“last\_name”) |
| SELECT COUNT(\*y)  FROM users | db.users.count() |
| SELECT COUNT(\*y)  FROM users where AGE > 30 | db.users.find({age: {“$gt”: 30}}).count() |
| SELECT COUNT(AGE) from users | db.users.find({age: {“$exists”: true}}).count() |
|  |  |
| CREATE INDEX myindexname ON users(name) | db.users.ensureIndex({name:1}) |
| CREATE INDEX myindexname ON users(name,ts DESC) | db.users.ensureIndex({name:1,ts:-1}) |
|  |  |
| EXPLAIN SELECT \* FROM users WHERE z=3 | db.users.find({z:3}).explain() |
|  |  |
| UPDATE users SET a=1 WHERE b=“q” | db.users.update({b:”q”}, {$set:{a:1}}, false, true) |
| UPDATE users SET a=a+2 WHERE b=“q” | db.users.update({b:”q”}, {$inc:{a:2}}, false, true) |
|  |  |
| DELETE FROM users WHERE z=“abc” | db.users.remove({z:”abc”}); |
|  |  |
| SELECT COUNT(\*) FROM users | db.users.aggregate([  {$group: {\_id:null, count:{$sum:1}}}  ]) |
| SELECT SUM(price) FROM orders | db.users.aggregate([  {$group: {\_id:null,  total:{$sum:”$price”}}}  ]) |
| SELECT cust\_id,SUM(price) FROM orders GROUP BY cust\_id | db.users.aggregate([  {$group: {\_id:”$cust\_id”,total:{$sum:”$price”}}}  ]) |
| SELECT cust\_id,SUM(price) FROM orders WHERE active=true GROUP BY cust\_id | db.users.aggregate([  {$match:{active:true}},  {$group:{\_id:”$cust\_id”,total:{$sum:”$price”}}}  ]) |