Advanced Queries

In MongoDB, just like in an RDBMS, creating appropriate indexes for queries is quite important for performance. See the Indexes page for more info.

Intro

MongoDB offers a rich query environment with lots of features. This page lists some of those features.

Queries in MongoDB are represented as JSON-style objects, very much like the documents we actually store in the database. For example:

```
// i.e., select * from things where x=3 and y="foo"
db.things.find( { x : 3, y : "foo" } );
```

Note that any of the operators on this page can be combined in the same query document. For example, to find all document where j is not equal to 3 and k is greater than 10, you'd query like so:

```
db.things.find({j: {$ne: 3}, k: {$gt: 10} });
```

Unless otherwise noted, the operations below can be used on array elements in the same way that they can be used on "normal" fields. For example, suppose we have some documents such as:

```
> db.things.insert({colors : ["blue", "black"]})
> db.things.insert({colors : ["yellow", "orange", "red"]})
```

Then we can find documents that aren't "red" using:

```
> db.things.find({colors : {$ne : "red"}})
{"_id": ObjectId("4dc9acea045bbf04348f9691"), "colors": ["blue","black"]}
```

Retrieving a Subset of Fields

See Retrieving a Subset of Fields

Conditional Operators

<, <=, >, >=

Use these special forms for greater than and less than comparisons in queries, since they have to be represented in the query document:

```
db.collection.find({ "field" : { $gt: value } } );   // greater than  : field > value
db.collection.find({ "field" : { $lt: value } } );   // less than  :  field < value
db.collection.find({ "field" : { $gte: value } } );  // greater than or equal to : field >= value
db.collection.find({ "field" : { $lte: value } } );  // less than or equal to : field <= value
```

For example:

```
db.things.find({j : {$lt: 3}});
db.things.find({j : {$gte: 4}});
```

You can also combine these operators to specify ranges:

```
db.collection.find({ "field" : { $gt: value1, $lt: value2 } } );    // value1 < field < value
```

$all

The `$all` operator is similar to `$in`, but instead of matching any value in the specified array all values in the array must be matched. For example, the object

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

would be matched by

```
db.things.find( { a: { $all: [ 2, 3 ] } } );
```

but not

```
db.things.find( { a: { $all: [ 2, 3, 4 ] } } );
```

An array can have more elements than those specified by the `$all` criteria. `$all` specifies a minimum set of elements that must be matched.

$exists

Check for existence (or lack thereof) of a field.

```
db.things.find( { a : { $exists : true } } ); // return object if a is present
db.things.find( { a : { $exists : false } } ); // return if a is missing
```

Before v2.0, $exists is not able to use an index. Indexes on other fields are still used.
$exists is not very efficient even with an index, and esp. with {`$exists:true`} since it will effectively have to scan all indexed values.

$mod

The `$mod` operator allows you to do fast modulo queries to replace a common case for where clauses. For example, the following $where query:

```
db.things.find( "this.a % 10 == 1")
```

can be replaced by:

```
db.things.find( { a : { $mod : [ 10 , 1 ] } } )
```

$ne

Use $ne for "not equals".

```
db.things.find( { x : { $ne : 3 } } );
```

$in

The `$in` operator is analogous to the SQL `IN` modifier, allowing you to specify an array of possible matches.

```
db.collection.find( { field : { $in : array } } );
```

Let's consider a couple of examples. From our *things* collection, we could choose to get a subset of documents based upon the value of the 'j' key:

```
db.things.find({j:{$in: [2,4,6]}});
```

Suppose the collection `updates` is a list of social network style news items; we want to see the 10 most recent updates from our friends. We could invoke:

```
db.updates.ensureIndex( { ts : 1 } ); // ts == timestamp
var myFriends = myUserObject.friends; // let's assume this gives us an array of DBRef's of my friends
var latestUpdatesForMe = db.updates.find( { user : { $in : myFriends } } ).sort( { ts : -1 }
).limit(10);
```

The target field's value can also be an array; if so then the document matches if any of the elements of the array's value matches any of the $in field's values (see the Multikeys page for more information.

$nin

The `$nin` operator is similar to `$in` except that it selects objects for which the specified field does not have any value in the specified array. For example

```
db.things.find({j:{$nin: [2,4,6]}});
```

would match {`j:1,b:2`} but not {`j:2,c:9`}.

$nor

The $nor operator lets you use a boolean or expression to do queries. You give $nor a list of expressions, none of which can satisfy the query.

$or

v1.6+

The $or operator lets you use boolean or in a query. You give $or an array of expressions, any of which can satisfy the query.

Simple:

```
db.foo.find( { $or : [ { a : 1 } , { b : 2 } ] } )
```

With another field

```
db.foo.find( { name : "bob" , $or : [ { a : 1 } , { b : 2 } ] } )
```

The $or operator retrieves matches for each or clause individually and eliminates duplicates when returning results.

> ⚠ $or can be nested as of v2.0, however nested $or clauses are not handled as efficiently by the query optimizer as top level $or clauses.

$and

v2.0+

The $and operator lets you use boolean and in a query. You give $and an array of expressions, all of which must match to satisfy the query.

```
db.foo.insert( { a: [ 1, 10 ] } )
db.foo.find( { $and: [ { a: 1 }, { a: { $gt: 5 } } ] } )
```

In the above example documents with an element of a having a value of a equal to 1 and a value of a greater than 5 will be returned. Thus the inserted document will be returned given the multikey semantics of MongoDB.

$size

The $size operator matches any array with the specified number of elements. The following example would match the object {a:["foo"]}, since that array has just one element:

```
db.things.find( { a : { $size: 1 } } );
```

You cannot use $size to find a range of sizes (for example: arrays with more than 1 element). If you need to query for a range, create an extra size field that you increment when you add elements. Indexes cannot be used for the $size portion of a query, although if other query expressions are included indexes may be used to search for matches on that portion of the query expression.

$type

The $type operator matches values based on their BSON type.

```
db.things.find( { a : { $type : 2 } } ); // matches if a is a string
db.things.find( { a : { $type : 16 } } ); // matches if a is an int
```

Possible types are:

Type Name	Type Number
Double	1
String	2

Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular expression	11
JavaScript code	13
Symbol	14
JavaScript code with scope	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

For more information on types and BSON in general, see http://www.bsonspec.org.

Regular Expressions

You may use regexes in database query expressions:

```
db.customers.find( { name : /acme.*corp/i } );
db.customers.find( { name : { $regex : 'acme.*corp', $options: 'i' } } );
```

If you need to combine the regex with another operator, you need to use the `$regex` clause. For example, to find the customers where name matches the above regex but does not include `'acmeblahcorp'`, you would do the following:

```
db.customers.find( { name : { $regex : /acme.*corp/i, $nin : ['acmeblahcorp'] } } );
```

Note that `$regex` queries are escaped slightly differently than JavaScript native regular expressions. For example:

```
> db.x.insert({someId : "123[456]"})
> db.x.find({someId : /123\[456\]/}) // use "\" to escape
> db.x.find({someId : {$regex : "123\\[456\\]"}}) // use "\\" to escape
```

An index on the field queried by regexp can increase performance significantly, as follows:

- Simple prefix queries (also called rooted regexps) like `/^prefix/` will make efficient use of the index (much like most SQL databases that use indexes for a `LIKE 'prefix%'` expression). This only works if the expression is left-rooted and the `i` (case-insensitivity) flag is not used.
- All other queries will not make an efficient use of the index: all values in the index will be scanned and tested against the regular expression.

While `/^a/`, `/^a.*/`, and `/^a.*$/` are equivalent, they will have different performance characteristics. The latter two will be slower as they have to scan the whole string. The first format can stop scanning after the prefix is matched.

MongoDB uses PCRE for regular expressions. Valid flags are:

- `i` - Case insensitive. Letters in the pattern match both upper and lower case letters.
- `m` - Multiline. By default, Mongo treats the subject string as consisting of a single line of characters (even if it actually contains newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline.
  When `m` it is set, the "start of line" and "end of line" constructs match immediately following or immediately before internal newlines in the subject string, respectively, as well as at the very start and end. If there are no newlines in a subject string, or no occurrences of ^ or $ in a pattern, setting `m` has no effect.
- `x` - Extended. If set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class. Whitespace does not include the VT character (code 11). In addition, characters between an unescaped # outside a character class and the next newline, inclusive, are also ignored.
  This option makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence `(?(` which introduces a conditional subpattern.
- `s` - Dot all. **New in 1.9.0**. Allows the dot (`.`) to match all characters including new lines. By default, `/a.*b/` will not match the string `"apple\nbanana"`, but `/a.*b/s` will.

Note that javascript regex objects only support the `i` and `m` options (and `g` which is ignored by mongod). Therefore if you want to use other options, you will need to use the $regex, $options query syntax.

Value in an Array

To look for the value "red" in an array field `colors`:

```
db.things.find( { colors : "red" } );
```

That is, when "colors" is inspected, if it is an array, each value in the array is checked. This technique may be mixed with the embedded object technique below.

$elemMatch

v1.4+

Use $elemMatch to check if an element in an array matches the specified match expression.

```
> t.find( { x : { $elemMatch : { a : 1, b : { $gt : 1 } } } } )
{ "_id" : ObjectId("4b5783300334000000000aa9"),
  "x" : [ { "a" : 1, "b" : 3 }, 7, { "b" : 99 }, { "a" : 11 } ]
}
```

Note that a single array element must match all the criteria specified; thus, the following query is semantically different in that each criteria can match a different element in the x array:

```
> t.find( { "x.a" : 1, "x.b" : { $gt : 1 } } )
```

See the dot notation page for more.

You only need to use this when more than one field must be matched in the array element.

Value in an Embedded Object

For example, to look `author.name=="joe"` in a postings collection with embedded author objects:

```
db.postings.find( { "author.name" : "joe" } );
```

See the dot notation page for more.

Meta operator: $not

The `$not` meta operator can be used to negate the check performed by a standard operator. For example:

```
db.customers.find( { name : { $not : /acme.*corp/i } } );
```

```
db.things.find( { a : { $not : { $mod : [ 10 , 1 ] } } } );
```

The `$not` meta operator can **only** affect *other operators*. The following **do not** work. For such a syntax use the `$ne` operator.

```
db.things.find( { a : { $not : true } } ); // syntax error
```

$not is not supported for regular expressions specified using the {`$regex: ...`} syntax. When using $not, all regular expressions should be passed using the native BSON type (e.g. {`"$not": re.compile("acme.*corp")`} in PyMongo)

Javascript Expressions and `$where`

In addition to the structured query syntax shown so far, you may specify query expressions as Javascript. To do so, pass a string containing a Javascript expression to `find()`, or assign such a string to the query object member `$where`. The database will evaluate this expression for each object scanned. When the result is true, the object is returned in the query results.

For example, the following mongo shell statements all do the same thing:

```
> db.myCollection.find( { a : { $gt: 3 } } );
> db.myCollection.find( { $where: "this.a > 3" } );
> db.myCollection.find("this.a > 3");
> f = function() { return this.a > 3; } db.myCollection.find(f);
```

You may mix mongo query expressions and a $where clause. In that case you must use the $where syntax, e.g.:

```
> db.myCollection.find({registered:true, $where:"this.a>3"})
```

Javascript executes more slowly than the native operators listed on this page, but is very flexible. See the server-side processing page for more information.

Cursor Methods

`count()`

The `count()` method returns the number of objects matching the query specified. It is specially optimized to perform the count in the MongoDB server, rather than on the client side for speed and efficiency:

```
nstudents = db.students.find({'address.state' : 'CA'}).count();
```

Note that you can achieve the same result with the following, but the following is slow and inefficient as it requires all documents to be put into memory on the client, and then counted. Don't do this:

```
nstudents = db.students.find({'address.state' : 'CA'}).toArray().length; // VERY BAD: slow and uses
excess memory
```

On a query using skip() and limit(), count ignores these parameters by default. Use count(true) to have it consider the skip and limit values in the calculation.

```
n = db.students.find().skip(20).limit(10).count(true);
```

`limit()`

`limit()` is analogous to the LIMIT statement in MySQL: it specifies a maximum number of results to return. For best performance, use `limit()` whenever possible. Otherwise, the database may return more objects than are required for processing.

```
db.students.find().limit(10).forEach( function(student) { print(student.name + "<p>"); } );
```

In the shell (and most drivers), a limit of `0` is equivalent to setting no limit at all.

skip()

The `skip()` expression allows one to specify at which object the database should begin returning results. This is often useful for implementing "paging". Here's an example of how it might be used in a JavaScript application:

```
function printStudents(pageNumber, nPerPage) {
   print("Page: " + pageNumber);
   db.students.find().skip((pageNumber-1)*nPerPage).limit(nPerPage).forEach( function(student) {
print(student.name + "<p>"); } );
}
```

Paging Costs
Unfortunately skip can be (very) costly and requires the server to walk from the beginning of the collection, or index, to get to the offset/skip position before it can start returning the page of data (limit). As the page number increases skip will become slower and more cpu intensive, and possibly IO bound, with larger collections.

Range based paging provides better use of indexes but does not allow you to easily jump to a specific page.

snapshot()

Indicates use of snapshot mode for the query. Snapshot mode assures no duplicates are returned, or objects missed, which were present at both the start and end of the query's execution (even if the object were updated). If an object is new during the query, or deleted during the query, it may or may not be returned, even with snapshot mode.

Note that short query responses (less than 1MB) are effectively snapshotted.

Currently, snapshot mode may not be used with sorting or explicit hints.

For more information, see How to do Snapshotted Queries in the Mongo Database.

sort()

`sort()` is analogous to the ORDER BY statement in SQL - it requests that items be returned in a particular order. We pass `sort()` a key pattern which indicates the desired order for the result.

```
db.myCollection.find().sort( { ts : -1 } ); // sort by ts, descending order
```

`sort()` may be combined with the `limit()` function. In fact, if you do not have a relevant index for the specified key pattern, `limit()` is recommended as there is a limit on the size of sorted results when an index is not used. Without a `limit()`, or an index, a full in-memory sort must be done. However, using a `limit()` reduces the required memory footprint and increases the speed of the operation by using an optimized sorting algorithm.

batchSize()

`batchSize()` determines the number of documents MongoDB returns in each batch to the client. MongoDB considers batchSize(), limit(), and the size in bytes of each document when deciding how many documents to send in each batch.

```
db.myCollection.find().batchSize(10);
```

The shell and most drivers present results to your application code as if they came in a single batch, so you normally do not need to think about batchSize.

Meta query operators

$returnKey

Only return the index key:

```
db.foo.find()._addSpecial("$returnKey" , true )
```

$maxScan

Limit the number of items to scan:

```
db.foo.find()._addSpecial( "$maxScan" , 50 )
```

$orderby

Sort results:

```
db.foo.find()._addSpecial( "$orderby", {x : -1} )
// same as
db.foo.find().sort({x:-1})
```

$explain

Explain the query instead of actually returning the results:

```
db.foo.find()._addSpecial( "$explain", true )
// same as
db.foo.find().explain()
```

$snapshot

Snapshot query:

```
db.foo.find()._addSpecial( "$snapshot", true )
// same as
db.foo.find().snapshot()
```

$min and $max

Set index bounds (see min and max Query Specifiers for details):

```
db.foo.find()._addSpecial("$min" , {x: -20})._addSpecial("$max" , { x : 200 })
```

$showDiskLoc

Show disk location of results:

```
db.foo.find()._addSpecial("$showDiskLoc" , true)
```

$hint

Force query to use the given index:

```
db.foo.find()._addSpecial("$hint", {_id : 1})
```

$comment

You can put a $comment field on a query to make looking in the profiler logs simpler.

```
db.foo.find()._addSpecial( "$comment" , "some comment to help find a query" )
```

```

```

`group()`

The `group()` method is analogous to GROUP BY in SQL. `group()` is more flexible, actually, allowing the specification of arbitrary reduction operations. See the Aggregation section of the Mongo Developers' Guide for more information.

See Also

- Indexes
- Optimizing Queries (including explain() and hint())

SDASDSSDD