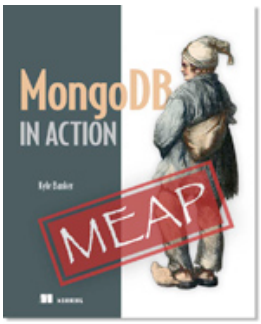
MANNING PUBLICATIONS



MongoDB in Action
By Kyle Banker

In this article, based on chapter 5 of MongoDB in Action, author Kyle Banker discusses MongoDB's general description queries, their semantics, and types.

MongoDB's Query Selectors

In this article, we present parts of MongoDB's query language. Specifically, we discuss general description queries, their semantics, and types.

Selector matching

The simplest way to specify a query is with a selector whose key-value pairs literally match the document you're looking for. A couple of examples follow.

```
db.users.find({last_name: "Banker"})
db.users.find({first_name: "Smith", age: 40})
```

The second query reads, "Find me a user with a first name is Smith *and* aged 40." Notice that, whenever you pass more than one key-value pair, both have to match; the query conditions are additive.[1]

Ranges

It's frequently necessary to query for documents whose values span a certain range. In SQL, we'd use <, <=, >, and >=; with MongoDB, we get the analogous set of operators `$gt`, `$gte`, `$lt`, and `$lte`. Their behavior is mostly predictable. However, beginners sometimes struggle with combining these operators. A common mistake is to repeat the search key:

```
db.users.find({age: {$gte: 0}, age: {$lte: 30})
```

But, because keys cannot be repeated, this query selector is invalid. This query is properly expressed as follows:

```
db.users.find({age: {$gte: 0, $lte: 30})
```

The only other surprise regarding the range operators involves types. In short, range queries will only match values having the same type as the value passed in. For example, suppose you have a collection with the following documents:

```
{ "_id" : ObjectId("4caf82011b0978483ea29ada"), "value" : 97 }
{ "_id" : ObjectId("4caf82031b0978483ea29adb"), "value" : 98 }
{ "_id" : ObjectId("4caf82051b0978483ea29adc"), "value" : 99 }
{ "_id" : ObjectId("4caf820d1b0978483ea29ade"), "value" : "a" }
{ "_id" : ObjectId("4caf820f1b0978483ea29adf"), "value" : "b" }
{ "_id" : ObjectId("4caf82101b0978483ea29ae0"), "value" : "c" }
```

You then issue the following query:

```
db.items.find({value: {$gte: 97})
```

[1] If you want express a logical OR, see the section below on logical operators.

Now, you may think to yourself that this query should return all six documents since the strings are numerically equivalent to the integers 97, 98, and 99. This is not the case. If you want the integer results, you need to issue the above query. If you want the string result, here's the query you'll want:

```
db.items.find({value: {$gte: "a"})
```

Of course, you won't have to worry about this type restriction as long as you never store multiple types for the same key within the same collection. Definitely a good base rule of thumb.

Set operators

Three query operators --`$in` , `$all`, and `$nin`-- take a list of one or more values as their predicate. `$in`, the most used of the three, returns a document if any of the given values matches the search key. We might use this operator to return all products belonging to some discrete set of categories. If the following list of category ids correspond to the lawn mowers, hand tools, and work clothing categories:

```
[ObjectId("6a5b1476238d3b4dd5000048"),
 ObjectId("6a5b1476238d3b4dd5000051"),
 ObjectId("6a5b1476238d3b4dd5000057")
]
```

Then finding all products belonging to these categories looks like this:

```
db.products.find(main_cat_id: { $in:
 [ObjectId("6a5b1476238d3b4dd5000048"),
  ObjectId("6a5b1476238d3b4dd5000051"),
  ObjectId("6a5b1476238d3b4dd5000057") })]
```

`$in` is frequently used with lists of ids. `$nin` matches only when none of the given elements matches. We might use `$nin` to find all products that are neither black nor blue:

```
db.products.find('details.color': { $nin: ["black", "blue"] }
```

Finally, `$all` matches if all the given elements match the search key. If we wanted to find all products tagged as gift and garden, `$all` would be a good choice:

```
db.products.find('tags': { $all: ["gift", "garden"] }
```

Naturally, this query only makes sense if `tags` is an array.

Logical operators

MongoDB's logical operators include `$ne`, `$not`, `$or`, and `$exists`. There is no `$and`, since adding conditions to a query selector always implies `AND`.

`$ne`, the not equal operator, works as expected. In practice, it's best used in combination with at least one other operator; otherwise, it's likely to be pretty inefficient. We might use `$ne` to find all products manufactured by ACME that aren't tagged "gardening."

```
db.products.find('details.manufacturer': 'ACME', tags: {$ne: "gardening"} }
```

`$ne` works on keys pointing to single values and to arrays, as shown in the foregoing example.

While `$ne` matches the negation of the specified values, `$not` will negate the result of another MongoDB operator or regular expression query. Keep in mind that most query operators already have a negated form (`$in` and `$nin`, `$gt` and `$lte`, and so on); `$not` shouldn't be used with any of these. Reserve `$not` for times when the operator or regex you're using lacks a negated form. If you wanted to query for all users with last names not beginning with B, you could use `$not` like so:

```
db.users.find(last_name: {$not: /^B/} }
```

`$or` expresses the logical disjunction of two values for two different keys. This is an important point: if the possible values are scoped to the same key, use `$in` instead. Trivially, finding all products that are either blue or green looks like this:

```
db.products.find('details.color': {$in: ['blue', 'green']} }
```

But finding all products that are either blue or made by ACME requires `$or`:

```
db.products.find({ $or: [{'details.color': 'blue'}, 'details.manufacturer': 'ACME'}] })
```

Notice that `$or` takes an array of query selectors, where each selector can be arbitrarily complex and contain other query operators.

Because collections don't enforce a fixed schema, we sometimes need a way to query for documents containing a particular key. For this, we can use the `$exists` operator. Suppose we had planned to use a product's `details` attribute to store custom fields. Assuming that not all products specify a set of colors, we can query for them:

```
db.products.find({'details.color': {$exists: false}})
```

The opposite query is also possible:

```
db.products.find({'details.color': {$exists: true}})
```

Now, before moving on, it's worth noting that there's another way to check for existence that's practically equivalent: by querying for `null`.[2] This would alter the above two queries slightly. The first could be expressed like so:

```
db.products.find({'details.color': null})
```

And the second:

```
db.products.find({'details.color': {$ne: null}})
```

The main advantage of querying for `null` is that such a query can use an index. While `$exists` always performs a full table scan, these queries for a `null` or non-null value will use an index if it exists. That's a good thing. So prefer the query for `null` if your application allows it.[3]

Objects

Some of the entities in our e-commerce data model have keys that point to a single embedded object. The product details attribute is one good example. Here's part of the relevant document, expressed as JSON:

```
{ _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",

  details: {
      model_num: 4039283402,
      manufacturer: "Acme",
      manufacturer_id: 432,
      color: "Green"
    }
}
```

We can query on such objects by separating the relevant keys with a dot. For instance, if we want to find all products manufactured by Acme, here would be our query:

```
db.products.find({'details.manufacturer_id': 432});
```

Such queries can be specified arbitrarily deep. Thus, supposing we had the following slightly modified representation:

```
{ _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",

  details: {
      model_num: 4039283402,
      manufacturer: { name: "Acme",
                      id: 432 },

      color: "Green"
    }
}
```

The query would be as expected:

```
db.products.find({'details.manufacturer.id': 432});
```

But, in addition to querying on a single object attribute, we can query on an object as a whole. This sometimes comes in quite handy. For instance, imagine you're using MongoDB to store stock data. To save space, you may want to forgo the standard object id and replace it with a compound key consisting of the stock symbol and a timestamp. Here's how a representative document might look:[4]

```
{ _id: {sym: 'gs', date: 20101005}
  open: 40.23,
  high: 45.50,
  low: 38.81,
  close: 41.22
```

---

[2] The queries differ in one subtle way: the query selector {name: {$exists: true}} will match the document {name: null} whereas a selector {name: {$ne: null}} will not. For most applications, this distinction probably isn't important. But for the time when it is, be aware of this difference.

[3] $exists will eventually use an index. See SERVER-393.

[4] If this were a high-throughput scenario, I'd do everything possible to limit the document size. This could be accomplished, in part, by using short key names. Thus, the key name o could be used in lieu of open. I might also store the date as an integer.

```
}
```

We could then find the summary of GS for October 5, 2010 with the following `_id` query:

```
db.ticks.find({_id: {sym: 'gs', date: 20101005} });
```

Keep in mind that these queries do a strict, byte-by-byte comparison, which means that the order of the keys matters. The following query is not equivalent and won't match our sample document:

```
db.ticks.find({_id: {date: 20101005, sym: 'gs'} });
```

And while the order of keys will be preserved in JSON documents entered via the shell, this is not necessarily true for document representations in the various language drivers. In particular, hashes in Ruby 1.8 are not order-preserving. To preserve order in Ruby 1.8, you'll need to explicitly construct an `OrderedHash`:

```
doc = BSON::OrderedHash.new
doc['sym'] = 'gs'
doc['date'] = 20101005

@ticks.find(doc)
```

Arrays

Arrays give the document model its power. They're used to store lists of strings, object ids, and even other documents. Arrays afford us rich yet comprehensible documents; it stands to reason that MongoDB would let us query and index the array type with ease. Indeed, the simplest array queries look just like queries on any other document type. Take product tags, for example. These tags are represented as a simple list of strings:

```
{ _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",

  tags: ["tools", "equipment", "soil"]
}
```

Querying for products with the tag soil is trivial:

```
db.products.find({tags: "soil"})
```

This query is also efficient so long as we have an index on the `tags` field:

```
db.products.ensureIndex({tags: 1})
```

When we need more control over our queries, we can use dot-notation to query for a value at a particular position within the array. Here's how we'd restrict the previous query to the first of a product's tags:

```
db.products.find({'tags.0': "soil"})
```

It might not make much sense to query tags in this way, but imagine we're dealing with user addresses. In this case, we have an array of sub-documents.

```
{ _id: BSON::ObjectID("4c4b1476238d3b4dd5000001")
  username: "kbanker",

  addresses: [
    {name: "home",
     street: "588 5th Street",
     city: "Brooklyn",
     state: "NY",
     zip: 11215},

    {name: "work",
     street: "1 E. 23rd Street",
     city: "New York",
     state "NY",
     zip 10010},
  ]
}
```

We might stipulate that the first of these addresses always be the user's primary shipping address. Thus, to find all users whose primary shipping address is in New York, we could again use the specify the zero position and combine that with a dot to target the `state` field:

```
db.users.find({'addresses.0.state': "NY"})
```

We can just as easily omit the position and use a specify a field alone. Following query will return a user document if *any* of the addresses in the list are in New York.

```
db.users.find({'addresses.state': "NY"})
```

As we did before, we can index this dotted field:

```
db.users.ensureIndex({'addresses.state': 1})
```

Notice that the same dot-notation is used regardless of whether a field points to a subdocument or to an array of subdocuments. This is powerful, and the consistency is reassuring. However, ambiguity can arise when querying for more than attribute within an array of subobjects. For example, suppose we want to fetch a list of all users whose home address is in New York. Can you think of a way to express this query?

```
db.users.find({'addresses.name': 'home', 'addresses.state': 'NY'})
```

The problem with this query is that the field references aren't restricted to a single address. In other words, this query will match as long as one of the addresses is designated as "home" and one is in New York, but these need to be the *same* address. To restrict multiple conditions to the same subdocument, you have to use the `$elemMatch` operator. Here's how we'd use `$elemMatch` to ensure that home and NY appear in the same subdocument:

```
db.users.find({addresses: {$elemMatch: {name: 'home', state: 'NY'}}})
```

Logically, `$elemMatch` is used only when you need to match two or more attributes in a subdocument.

JavaScript

If you can't express your query with the tools described thus far, then you may have to write some JavaScript. You can use the special `$where` operator to pass a JavaScript expression to any query. Within the JavaScript context, `this` refers to the current document. To take an example:

```
db.reviews.find({$where: "function() { return this.helpful_votes > 3; }"})
```

There's also an abbreviated form for simple expressions like this one:

```
db.reviews.find({$where: "this.helpful_votes > 3"})
```

This query works but can be easily expressed with the standard query language. JavaScript expressions can't use an index, and they incur a substantial cost because they must be evaluated within a JavaScript interpreter context. For these reasons, you should issue JavaScript queries only when the standard query language can't express your query. If you do find yourself needing JavaScript, try to combine the JavaScript expression with another query operator. The standard query operator will prompt the server to pare down the result set, thus limiting the number of documents that must be loaded into a JS context. Let's take a quick example to see how this would work.

Imagine that, for each user, we've calculated a rating reliability factor. This is essentially an integer that, when multiplied by the user's rating, results in a more normalized rating. Suppose further that we want to query a particular user's reviews and only return a normalized rating greater than 3. Here's how that query would look:

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
$where: "(this.rating * .92) > 3"})
```

This query meets both recommendations: it uses a standard query, on a presumably-indexed field, and it employs a JavaScript `user_id` expression that's absolutely beyond the capabilities of the standard query language.

In addition to the attendant performance penalties, it's good to be aware of the possibility of JavaScript injection attacks. An injection attack becomes possible whenever a user is allowed to enter code directly into a JavaScript query. While there's never any danger of the user writing or deleting data, a user might be able to read sensitive data. A naive JavaScript query in Ruby might look something like this:

```
@users.find({$where => "this.#{attribute} == #{value}"})
```

Assuming that the user controls the values of `attribute` and `value`, he can use manipulate query to search the `users` collection on any attribute pair. While this might not be the worst imaginable intrusion, we'd be wise to prevent this.

Regular expressions

You can use a regular expression within a query. You would use, for example, a prefix expression, `/^B/`, to find last names beginning with a B, and this query would use an index. In fact, much more is possible. MongoDB is compiled with PCRE,[5] which supports a huge gamut of regular expressions.

Now, with the exception of the prefix-style query just described, regular expressions queries can't use an index. Thus, I recommend using them as you would a JavaScript expression, that is, in combination with at least one other query term. Here's how you might query a given user's reviews for review text containing one of the words "best" or "worst." Notice that we use the `i` regex flag to indicate case insensitivity:

[5] http://en.wikipedia.org/wiki/Perl_Compatible_Regular_Expressions

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
text: /best|worst/i })
```

If the language you're using has a native regex type, you can use a native regex object to perform the query. The above query, in Ruby, is practically identical:

```
@reviews.find({:user_id => BSON::ObjectId("4c4b1476238d3b4dd5000001"),
:text => /best|woest/i })
```

If you're querying from an environment that doesn't support a native regex type, you can use the special `$regex` and `$options` operators. Using these operators from the shell, the previous query would be expressed like so:

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
text: {$regex: "best|worst", $options: "i" })
```

Special query operators

There are two more query operators that aren't easily categorized and, thus, deserve their own section. The first is `$mod`, which allows you to query for the modulo of a given value.

Finally, there's the `$type` operator, which matches a value if it's represented as a particular BSON type. I don't recommend storing multiple types for the same field within a collection but, if the situation ever arises, you have a query operator that essentially lets you test against type. This came in handy recently when we had discovered a user whose `_id` queries weren't always matching when they should have. This problem, it turned out, was that the user had been storing ids as both strings and as proper object ids. These are represented as BSON types 2 and 7, respectively; to a new user, this distinction is easy to miss.

Correcting the issue first required finding all documents with ids stored as strings. The `$type` operator can help us to just that:

```
db.users.find({_id: {$type: 2}})
```

Summary

Queries allow us to get the data as it's stored. You've learned about general description queries, their semantics, and types. If you're ever unsure of how a particular combination of query operators will serve you, the shell is always a ready test bed.

Here are some other Manning titles you might be interested in:

SQL Server MVP Deep Dives
Paul Nielsen, Kalen Delaney, Greg Low, Adam Machanic, Paul S. Randal, and Kimberly L. Tripp

SQL Server DMVs in Action
Ian W. Stirk

SQL Server 2008 Administration in Action
Rod Colledge

Last updated: February 27, 2011