CHAPTER 10

Optimization

There is a tongue-in-cheek statement about MongoDB attributed to an unknown Twitter user: "If a MongoDB query runs for longer than 0ms, then something is wrong." This is typical of the kind of buzz that surrounded the product when it first burst onto the scene about a year ago.

The reality is that MongoDB is extraordinarily fast. But if you give it the wrong data structures, or you don't set up the database or collections with the right indexes, then MongoDB can slow down dramatically, like any data storage system.

MongoDB also contains some advanced features, such as grouping and map/reduce, that require some tuning to get them running with optimal efficiency.

The design of your data schemas can also have a big impact on performance; in this chapter, we will look at some techniques to shape your data into a form that makes maximum use of MongoDB's strengths and minimizes its weaknesses.

Before we look at improving the performance of the queries being run on the server or the ways of optimizing the structure of the data, we'll begin with a look at how MongoDB interacts with the hardware it runs on and the factors that affect performance.

Optimizing Your Server Hardware for Performance

Often the quickest and cheapest optimization you can make to a database server is to right-size the hardware it runs on. If a database server has too little memory or uses slow drives, it can impact database performance significantly. And, while some of these constraints may be acceptable for a development environment where the server may be running on a developer's local workstation, they may not be acceptable for production applications, where some care must be used in calculating the correct hardware configuration to achieve the best performance.

Understanding How MongoDB Uses Memory

MongoDB uses memory-mapped file I/O to access its underlying data storage. This method of file I/O has some characteristics that you should be aware of because they can affect both the type of operating system (OS) you run it under and the amount of memory you install.

The first notable characteristic of memory-mapped files is that, on 32-bit operating systems, the maximum file size that can be managed is 2 GB. This effectively limits the size of a MongoDB database on 32-bit operating systems. If you need to store more than 2 GB, then you should consider installing and running MongoDB on a 64-bit operating system where the limit is greater than 4 TB.

The second notable characteristic is that memory-mapped files use the operating system's virtual memory system to map the required parts of the database files into memory, as it needs them. This can result in the slightly alarming impression that MongoDB will use up your entire RAM. This is not really the case because MongoDB will share the virtual address space with other applications and the operating system, and it will release memory back to the operating system as it is needed. Using the free

memory total as an indicator of excessive memory consumption is not a good practice because a good OS will ensure there is little or no free memory. This is because all of your expensive memory is pressed into good use through caching or buffering disk I/O. Free memory is wasted memory.

By providing a suitable amount of memory, MongoDB can keep more of its data mapped into memory, which reduces the need for expensive disk I/O.

In general, the more memory you give to MongoDB, the faster it will run. However, if you have a 2 GB database, then adding more than 2–3 GB of memory will not make much difference because the whole database will sit in RAM anyway.

Choosing the Right Database Server Hardware

There is a general pressure to move to lower-power (energy) systems for hosting services. However, many of the lower-power servers use laptop or notebook components to achieve the lower power consumption. Unfortunately, lower-quality server hardware can use less expensive disk drives in particular. Such drives are not suited for heavy-duty server applications due to their disks' low rotation speed, which slows the rate at which data can be transferred to and from the drive. Also, make sure you use a reputable supplier, one that you trust to assemble a system that has been optimized for server operation.

If you plan to use replication or any kind of frequent backup system that would have to read across the network connections, then you should consider putting in an extra network card and forming a separate network so the servers can talk with each other. This reduces the amount of data being transmitted and received on the network interface used to connect the application to the server, which also has an effect on an application's performance.

Evaluating Query Performance

MongoDB has two main tools for optimizing query performance: `explain()` and the MongoDB Profiler (the db profiler). The db profiler is a good tool for finding those queries that are not performing well and selecting candidate queries for further inspection, while `explain()` is good for investigating a single query, so you can determine how well it is performing.

Those of you familiar with MySQL will probably also be familiar with the use of the "slow query log" which helps you find queries that are consuming a lot of time; MongoDB uses the db profiler to provide this capability.

MongoDB Profiler

The MongoDB profiler is a tool that records statistical information and execution plan details for every query that meets the trigger criteria. You can enable this tool separately on each database.

Once enabled, MongoDB inserts a document with information about the performance and execution details of each query submitted by your application into a special *capped* collection called `system.profile`. You can use this collection to inspect the details of each query logged using normal collection querying commands.

The `system.profile` collection is limited to a maximum of 128 KB of data, so that the profiler will not fill the disk with logging information. This limit should be enough to capture a few thousand profiles of even the most complex queries.

■ **Warning** When the profiler is enabled, it will impact the performance of your server, so it is not a good idea to leave it running on a production server unless you are performing an analysis for some observed issue. Don't be tempted to leave it running permanently to provide a window on recently executed queries.

Enabling and Disabling the DB Profiler

It's a simple matter to turn on the MongoDB profiler:

```
$mongo
>use blog
>db.setProfilingLevel(2)
```

It is an equally simple matter to disable the profiler:

```
$mongo
>use blog
>db.setProfilingLevel(0)
```

MongoDB can also enable the profiler only for queries that exceed a specified execution time. The following example logs only queries that take more than half a second to execute:

```
$mongo
>use blog
>db.setProfilingLevel(1,500)
```

For profiling level 1, you can supply a maximum query execution time value in milliseconds (ms). If the query runs for longer than this amount of time, it is profiled and logged; otherwise, it is ignored. This provides the same functionality seen in MySQL's "slow query log."

Finding Slow Queries

A typical record in the `system.profile` collection looks like this:

```
> db.system.profile.find()
{
     "ts" : "Mon Aug 23 2010 22:06:09 GMT+0800 (PHT)",
    "info" : "query blog.posts reslen:21342 nscanned:202  \nquery: { Tags: \"even\" }
nreturned:101
        bytes:21326",
        "millis" : 15
 }
```

Each record contains fields, and the following list outlines what they are and what they do:

- `ts`: Displays a timestamp in UTC that indicates when the query was executed.
- `info`: Shows statistical information about the query and its structure.
- `milis`: Records the number of milliseconds it took to execute the query.

Because the `system.profile` collection is just a normal collection, you can use MongoDB's query tools to home in quickly on problematic queries.

The next example finds all the queries that are taking longer than 10ms to execute. In this case, you can just query for cases where `millis >10` in the `system.profile` collection, and then sort the results by the execution time in descending order:

```
> db.system.profile.find({millis:{$gt:10}}).sort({millis:-1})
{ "ts" : "Mon Aug 23 2010 22:06:09 GMT+0800 (PHT)", "info" :
        "query blog.posts reslen:21342 nscanned:202  \nquery: { Tags: \"even\" }
        nreturned:101 bytes:21326", "millis" : 15 }
```

If you also know your problem occurred during a specific time range, then you can use the `ts` field to add query terms that restrict the range to the required slice.

Analyzing a Specific Query with explain()

If you suspect a query is not performing as well as expected, then you can use the `explain()` modifier to break down how MongoDB is executing the query.

When you add the `explain()` modifier to a query, MongoDB executes the query, returning a document that describes how the query was handled, rather than a cursor to the results. The following query runs against a database of blog posts and indicates that the query had to scan 13325 records to form a cursor to return all the posts:

```
$mongo
>use blog
> db.posts.find().explain()
{
    "cursor" : "BasicCursor",
    "indexBounds" : [ ],
    "nscanned" : 13235,
    "nscannedObjects" : 13235,
    "n" : 13235,
    "millis" : 3,
    "allPlans" : [
        {
            "cursor" : "BasicCursor",
            "indexBounds" : [ ]
        }
    ]
}
```

You can see the fields returned by `explain()` listed in Table 10–1.

Table 10–1. *Elements Returned by explain()*

Element	Description
`Cursor`	Indicates the type of cursor created to enumerate the results. Typically, this is `BasicCursor`, a natural order reading cursor.
`indexBounds`	Indicates the min/max values used on an indexed lookup.
`nScanned`	Indicates the number of references scanned to find all the objects in the query.

Element	Description
nScannedObjects	Indicates the number of Physical objects that "nScanned" resolves to.
n	Indicates the number of items on the cursor (i.e., the number of items to be returned).
millis	Indicates the number of milliseconds it took to execute the query.
allPlans	Indicates the number of alternative methods the optimizer could use to execute this query. If there are multiple indexes available that would satisfy the query, then those plans would be shown here. This information is useful if you want to use the hint() modifier to move the query to a different plan than the one currently selected.

Using Profile and explain() to Optimize a Query

Now let's walk through a real-world optimization scenario and look at how we can use MongoDB's profile and explain() modifier tools to fix a problem with a real application.

The example discussed in this chapter is based on the blog database from Chapter 8. This database has a function to get the posts associated with a particular tag; this chapter will use the "even" tag. Let's assume that you have noticed this function runs slowly, so you want to determine whether there is a problem.

Let's begin by writing a little program to fill the aforementioned database with data so that we have something to run queries against, to demonstrate the optimiazaiton process.

```
<?php

// Get a connection to the database

$mongo = new Mongo();
$db=$mongo->blog;

// First let's get the first AuthorsID
// We are going to use this to fake a author

$author = $db->authors->findOne();

if(!$author){
        die("There are no authors in the database");
}

for( $i = 1; $i < 10000; $i++){
        $blogpost=array();
        $blogpost['author'] = $author['_id'];
        $blogpost['Title']  = "Completely fake blogpost number {$i}";
        $blogpost['Message'] = "Some fake text to create a database of blog posts";
        $blogpost['Tags'] = array();
        if($i%2){
```

```
                // Odd numbered blogs
                $blogpost['Tags'] = array("blog", "post", "odd", "tag{$i}");
        } else {
                // Even numbered blogs
                $blogpost['Tags'] = array("blog", "post", "even", "tag{$i}");
        }
        $db->posts->insert($blogpost);
}
?>
```

The preceding program finds the first author in the `blog` database's `authors` collection, and then pretends that the author has been extraordinarily productive. It creates 10,000 fake blog postings in the author's name, all in the blink of an eye. The posts are not very interesting to read; nevertheless, they are alternatively assigned "`odd`" and "`even`" tags. These tags will serve to demonstrate how to optimize a simple query.

The next step is to save the program as `fastblogger.php` and then run it using the command-line PHP tool:

```
$php fastblogger.php
```

Next, you need to enable the database profiler, which you will use to determine whether you can improve the example's queries:

```
$ mongo
> use blog
switched to db blog
> show collections
authors
posts
...
system.profile
tagcloud
...
users
> db.setProfilingLevel(2)
{ "was" : 0, "ok" : 1 }
```

Now wait a few moments for the command to take effect, open the required collections, and then perform its other tasks. Next, you want to simulate having the blog website access all of the blog posts with the `even` tag. Do so by executing a query that the site can use to implement this function:

```
$Mongo
use blog
$db.posts.find({Tags:"even"})
...
```

If you query the `profiler` collection for results that exceed 5ms, then you should see something like this:

```
>db.system.profile.find({millis:{$gt:5}}).sort({millis:-1})
{"ts" : "Mon Aug 23 2010 22:06:09 GMT+0800 (PHT)",
        "info" : "query blog.posts reslen:21342 nscanned:202  \nquery:
                { Tags: \"even\" }  nreturned:101 bytes:21326", "millis" : 15 }
...
```

The results returned in the preceding example show that some queries are taking longer than 0ms (remember the quote at the beginning of the chapter).

Next, you want to reconstruct the query for the first (and worst performing) query, so you can see what is being returned. The preceding output indicates that the poor performing query is querying `blog.posts` and that the query term is `{Tags:"even"}`. Finally, you can see that this query is taking a whopping 15ms to execute.

The reconstructed query looks like this:

```
>db.posts.find({Tags:"even"})
{ "_id" : ObjectId("4c727cbd91a01b2a14010000"), "author" :
ObjectId("4c637ec8b8642fea02000000"), "Title" : "Completly fake blogpost number 2", "Message"
: "Some fake text to create a database of blog posts", "Tags" : [ "blog", "post", "even",
"tag2" ] }
{ "_id" : ObjectId("4c727cbd91a01b2a14030000"), "author" :
ObjectId("4c637ec8b8642fea02000000"), "Title" : "Completly fake blogpost number 4", "Message"
: "Some fake text to create a database of blog posts", "Tags" : [ "blog", "post", "even",
"tag4" ] }
{ "_id" : ObjectId("4c727cbd91a01b2a14050000"), "author" :
ObjectId("4c637ec8b8642fea02000000"), "Title" : "Completly fake blogpost number 6", "Message"
: "Some fake text to create a database of blog posts", "Tags" : [ "blog", "post", "even",
"tag6" ] }
...
```

The preceding output should come as no surprise; this query was created for the expressed purpose of demonstrating how to find and fix a slow query.

The goal is to figure how to make the query run faster, so use the `explain()` command to determine how MongoDB is performing this query:

```
> db.posts.find({Tags:"even"}).explain()
{
    "cursor" : "BasicCursor",
    "nscanned" : 9999,
    "nscannedObjects" : 9999,
    "n" : 4999,
    "millis" : 14,
    "indexBounds" : {

    }
}
```

You can see from the preceding output that the query is not using any indexes. The `explain()` command shows that the query is using a "`BasicCursor`", which means the query is just performing a simple scan of the collection's records. Specifically, it's scanning all of the records in the database one by one to find the tags (all 9999 of them); this process takes 14ms. This may not sound like a very long time, but, if you were to use this query on a popular page of your website, then it would be causing additional load to the disk I/O, as well as a tremendous amount of stress on the web server. Consequently, this query would cause the connection to the web browser to remain open for longer while the page is being created.

■ **Note** If you see a detailed query explanation that shows a significantly larger number of scanned records (`nscanned`) than it returns (`n`), then that query is probably a candidate for indexing.

The next step is to determine whether adding an index on the "Tags" field improves the query's performance:

```
> db.posts.ensureIndex({Tags:1})
```

Now run the `explain()` command again to see the effect of adding the index:

```
> db.posts.find({Tags:"even"}).explain()
{
    "cursor" : "BtreeCursor Tags_1",
    "nscanned" : 4999,
    "nscannedObjects" : 4999,
    "n" : 4999,
    "millis" : 4,
    "indexBounds" : {
        "Tags" : [
            [
                "even",
                "even"
            ]
        ]
    }
}
>
```

The performance of the query has improved significantly. You can see that the query is now using a `BtreeCursor` driven by the `Tags_1` index. The number of scanned records has been reduced from 9999 records to the same 4999 records you expect the query to return, while the execution time has dropped to 4ms.

■ **Note** The most common index type used by MongoDB is the btree (binary-tree type). A `BtreeCursor` is a MongoDB data cursor that uses the binary tree index to navigate from document to document. Btree indexes are very common in database systems because they provide fast inserts and deletes, yet also provide a reasonable performance when used to walk or sort data.

Managing Indexes

You've now seen how much of an impact the introduction of carefully selected indexes can have.

As you learned in Chapter 3, MongoDB's indexes are used for both queries (`find`, `findOne`) and sorts. If you intend to use a lot of sorts on your collection, then you should add indexes that correspond to your sort specifications. If you use `sort()` on a collection where there are no indexes for the fields in the sort specification, then you may get an error message if you exceed the maximum size of the internal sort buffer. So it is a good idea to create indexes for sorts. In the following sections, we'll touch again on the basics, but also add some details that relate to how to manage and manipulate the indexes in your system. We will also cover how such indexes relate to some of the samples.

Each time you add an index to a collection, MongoDB must maintain it and update it every time you perform any write operation (e.g., `updates`, `inserts`, or `deletes`). If you have too many indexes on a collection, it can cause a negative impact on write performance.

Indexes are best used on collections where the majority of access is read access. For write-heavy collections such as those used in logging systems, introducing an index would reduce the peak documents per second that could be streamed into the collection.

■ **Warning** At this time, you can have a maximum of 40 indexes per collection.

Listing Indexes

MongoDB maintains a special collection called `system.indexes` inside each database. This collection keeps track of all the indexes that have been created on all the collections in the databases, including which fields or elements they refer to.

The `system.indexes` collection is just like any normal collection. This means you can list its contents, run queries against it, and otherwise perform the usual tasks you can accomplish with a typical collection.

The following example lists the indexes in your simple database:

```
$mongo
>use blog
>db.system.indexes.find()
{ "name" : "_id_", "ns" : "blog.posts", "key" : { "_id" : 1 } }
{ "name" : "_id_", "ns" : "blog.authors", "key" : { "_id" : 1 } }
```

The blog database does not have any user-defined indexes, but you can see the two indexes created automatically for the `_id` field on your two collections: `posts` and `authors`. You don't have to do anything to create or delete these *identity indexes*; MongoDB creates and drops them whenever a collection is created or removed.

When you define an index on an element, MongoDB will construct an internal btree index, which it will use to locate documents efficiently. If no suitable index can be found, then MongoDB will scan all the documents in the collection to find the records that will satisfy the query.

Creating a Simple Index

MongoDB provides the `ensureIndex()` function for adding new indexes to a collection. This function begins by checking whether an index has already been created with the same specification. If it has, then `ensureIndex()` just returns that index. This means you can call `ensureIndex()` as many times as you like, but it won't result in a lot of extra indexes being created for your collection.

The following example defines a simple index:

```
$mongo
>use blog
>db.posts.ensureIndex({Tags:1})
```

This preceding example creates a simple ascending btree index on the `Tags` field. Creating a descending index instead would require only small change:

```
>db.posts.ensureIndex({Tags:-1})
```

To index a field in an embedded document, you can use the normal dot notation addressing scheme; that is, if you have a `count` field that is inside a `comments` subdocument, then you can use this syntax to index it:

```
>db.posts.ensureIndex({"comments.count":1})
```

If you specify a document field, which is an array type, then the index will include all the elements of the array as separate index terms. This is known as a multi-key index, and each document is linked to multiple values in the index.

MongoDB has a special operator for performing queries where you wish to select only documents that have all of the terms you supply. In the `blog` database example, you have a `posts` collection with an element called `Tags`. This element has all the tags associated with the posting inside it. The following query finds all articles that have both the `sailor` and `moon` tags:

```
>db.posts.find({Tags:{$all: ['sailor', 'moon']}})
```

Without a multi-key index on the `Tags` field, the query engine would have to scan each document in the collection to see whether either term existed and, if so, to check whether both terms were present.

Creating a Compound Index

It may be tempting to simply create a separate index for each field mentioned in any of your queries. While this may speed up queries without having to apply too much thought, it would unfortunately have a significant impact on adding and removing data from your database, as these indexes need to be updated each time.

Compound indexes provide a good way to keep down the number of indexes you have on a collection, allowing you to combine multiple fields into a single index, so you should try to use compound indexes wherever possible.

There are two main types of compound indexes: subdocument indexes and manually defined compound indexes.

MongoDB has some rules that allow it to use compound indexes for queries that do not use all of the component keys. Understanding these rules enables you to construct a set of compound indexes that cover all of the queries you wish to perform against the collection, without having to individually index each element (thereby avoiding the attendant impact on insert/update performance mentioned earlier).

One area where compound indexes may not be useful is when using the index in a sort. Sorting is not good at using the compound index unless the list of terms and sort directions exactly matches the index structure. In this case, individual simple indexes on each of the term fields may be a better choice.

Creating Subdocument Compound Indexes

You can create a compound index using an entire subdocument; in this case, MongoDB will create a compound index where each element in the embedded document becomes part of the index. For example, assume you have an `author` subdocument with `name` and `email` elements inside it. The following snippet creates a compound index with two key terms in it: `author.name` and `author.email`:

```
>db.posts.ensureIndex({Author:1})
```

The preceding serves as a nice shortcut for creating compound indexes. However, you lose the ability to set the order of the keys in the index in this example because you can't set the direction of each.

Constructing a Compound Index Manually

When you use a subdocument as an index key, the order of the elements used to build the multi-key index matches the order in which they appear in the subdocument's internal BSON representation. In many cases, this does not give you sufficient control over the process of creating the index.

To get around this while guaranteeing that the query uses an index constructed in the desired fashion, you need to make sure you use the same subdocument structure to create the index that you used when forming the query, as in the following example:

```
>db.articles.find({author:{name: 'joe', email: 'joe@blogger.com'}))
```

You can also create a compound index explicitly by naming all the fields that you wish to combine in the index, and then specifying the order to combine them in. The following example illustrates how to construct a compound index manually:

```
>db.posts.ensureIndex({"author.name":1, "author.email":1})
```

Specifying Index Options

You can specify several interesting options when creating an index, such as creating unique indexes or enabling background indexing (you'll learn more about these options in the upcoming sections). You specify these options as additional parameters to the `ensureIndex()` function, as in the following example:

```
>db.posts.ensureIndex({author:1}, {option1:true, option2:true, ... })
```

Creating an Index in the Background with {background:true}

When you instruct MongoDB to create an index by using the `ensureIndex()` function for the first time, the server must read all the data in the collection and create the specified index. By default, this is done in the foreground and all operations on the specified collection block until the index operation completes.

MongoDB also includes a feature that allows indexes to be created in the background. Operations by other connections on that collection are not blocked while that index is being built. No queries will use the index until it has been built, but the server will allow read and write operations to continue. Once the index operation has been completed, all queries that require the index will immediately start using it.

■ **Note** The index will be built in the background. However, the connection that initiates the request will block, if you issue the command from the MongoDB shell. A command issued on this connection won't return until the indexing operation is complete. At first sight, this appears to contradict the idea that it is a *background* index. However, if you have another MongoDB shell open at the same time, you will find that queries and updates on that collection run unhindered while the index build is in progress. It is only the initiating connection that is blocked. This differs from the non-background behavior you see with a simple `ensureIndex()` command, where operations on the second MongoDB shell would also be blocked.

Killing the Indexing Process

You can also kill the current indexing process if you think it has hung or is otherwise taking too long. You can do this by invoking the `killOp()` function:

```
> db.killOp(<operation id>)
```

■ **Note** When invoking the `killOp()` command, the partial index will also be removed again. This prevents broken or irrelevant data from building up in the database.

Creating an Index with a Unique Key {unique:true}

When you specify the `unique` option, MongoDB creates an index where all the keys must be different. This means that MongoDB will return an error if you try to insert a document where the index key matches the key of an existing document. This is useful for a field where you want to ensure that no two people can have the same identity (e.g., `userid`).

However, if you want to add a unique index to an existing collection that is already populated with data, then you must make sure that you have *deduped* the key(s). In this case, your attempt to create the index will fail if any two keys are not unique.

The `unique` option works for simple and compound indexes, but not for multi-key value indexes, where they wouldn't make much sense.

If a document is inserted with a field missing that is specified as a unique key, then MongoDB will automatically insert the field, but will set its value to `null`. This means that you can only insert one document with a missing key field into such a collection; any additional `null` values would mean the key is not unique, as required.

Dropping Duplicates Automatically with {dropdups:true}

If you want to create a unique index for a field where you know that duplicate keys exist, then you can specify the `dropdups` option. This option instructs MongoDB to remove documents that would cause the index creation to fail. In this case, MongoDB will retain the first document it finds in its natural ordering of the collection, but then drop any other documents that would result in an index-constraint violation.

You need to be very careful when using this option because it *will* result in documents being deleted from your collection. You should be extremely aware of the data in your collection before using this option; otherwise, you might get unexpected (not to mention unwanted) behavior. It is an extremely good idea to run a group query against a collection for which you intend to make a key unique; this will enable you to determine the number of documents that would be regarded as duplicates *before* you executed this option.

Dropping an Index

You can elect to drop all indexes or just one specific index from a collection. Use the following function to remove all indexes from a collection:

```
>db.posts.dropIndexes()
```

To remove a single index from a collection, you use syntax that mirrors the syntax used to create the index with `ensureIndex()`:

```
>db.posts.dropIndex({"author.name":1, "author.email":1});
```

Re-Indexing a Collection

If you suspect that the indexes in a collection are damaged—for example, if you're getting inconsistent results to your queries—then you can force a re-indexing of the affected collection.

This will force MongoDB to drop and re-create all the indexes on the specified collection (see Chapter 9 for more information on how to detect and solve problems with your indexes), as in the following example:

```
> db.posts.reIndex()
{
    "nIndexesWas" : 2,
    "msg" : "indexes dropped for collection",
    "ok" : 1,
    "nIndexes" : 2,
    "indexes" : [
        {
            "name" : "_id_",
            "ns" : "blog.posts",
            "key" : {
                "_id" : 1
            }
        },
        {
            "_id" : ObjectId("4c7282c6d36e34765e2e6741"),
            "ns" : "blog.posts",
            "key" : {
                "Tags" : 1
            },
            "name" : "Tags_1"
        }
    ,
    "ok" : 1
}
```

The preceding output lists all the indexes the command has rebuilt, including the keys. Also, the `nIndexWas:` field shows how many indexes existed before running the command, while the `nIndex:` field gives the total number of indexes after the command has completed. If the two values are not the same, the implication is that there was a problem re-creating some of the indexes in the collection.

How MongoDB Selects Which Indexes It Will Use

When a database system needs to run a query, it has to assemble a *query plan*, which is a list of steps it must run to perform the query. Each query may have multiple query plans that could produce the same result equally well. However, each plan may have elements in it that are more *expensive* to execute than others. For example, a scan of all the records in a collection is an expensive operation, and any plan that incorporates such an approach could be slow. These plans can also include alternative lists of indexes to use for query and sort operations.

Road directions serve as a good illustration of this concept. If you want to get to the diagonally opposite side of a block from one corner, then "take a left, then take a right" and "take a right, then take a left" are equally valid plans for getting to the opposite corner. However, if one of the routes has two stop signs and the other has none, then the former approach is a more expensive plan, while the latter

approach is the best plan to use. Collection scans would qualify as potential stop signs when executing your queries.

MongoDB ships with a component called the query analyzer. This component takes a query and the collection the query targets. The component then produces a set of plans for MongoDB to execute. The `explain()` function described earlier in this chapter lists both the plan used and the set of alternative plans produced for a given query.

MongoDB also ships with a query optimizer component. The job of this component is to select which execution plan is best suited to run a particular query. In most relational database systems, a query optimizer uses statistics about the distribution of keys in a table, the number of records, the available indexes, the effectiveness of previous choices, and an assortment of weighting factors to calculate the cost of each approach. It then selects the least expensive plan to use.

The query optimizer in MongoDB is simultaneously dumber and smarter than the typical RDBMS query analyzer. For example, it does not use a cost-based method of selecting an execution plan; instead, it runs all of them in parallel and uses the one that returns the results fastest, terminating all the others after the winner crosses the finish line. Thus, the query analyzer in MongoDB uses a simple mechanism (dumb) to ensure that it always gets the fastest result (smart).

Using Hint() to Force Using a Specific Index

The query optimizer in MongoDB selects the best index from the set of available indexes for a collection. It uses the methods just outlined to try to match the best index or set of indexes to a given query. However, there may be cases where the query optimizer does not make the correct choice, in which case it may be necessary to give the component a helping hand.

You can provide a *hint* to the query optimizer in such cases, nudging the component into making a better choice. For example, if you have used `explain()` to show which indexes are being used by your query, and you think you would like it to use a different index for a given query, then you can force the query optimizer to do so.

Let's look at an example. Assume that you have an index on a subdocument called `author` with `name` and `email` fields inside it. Also assume that you have the following defined index:

```
>db.posts.ensureIndex({Author:1})
```

You can use the following hint to force the query optimizer to use the preceding defined index:

```
>db.posts.find({author:{name:'joe', email: 'joe@10gen.com'}}).hint({Author:1})
```

If for some reason you want to force a query to use no indexes, that is, if you want to use collection document scanning as the means of selecting records, then you can use the following hint to do this:

```
>db.posts.find({author:{name: 'joe', email: 'joe@10gen.com'}}).hint({$natural:1})
```

Optimizing the Storage of Small Objects

Indexes are the key to speeding up data queries. But another factor that can affect the performance of your application is the size of the data it accesses. Unlike database systems with fixed schemas, MongoDB stores all the schema data for each record inside the record itself. Thus, for large records with large data contents per field, the ratio of schema data to record data is low; however, for small records with small data values, this ratio can grow surprisingly large.

Consider a common problem in one of the application types that MongoDB is well suited for: logging. MongoDB's extraordinary write rate makes streaming events as small documents into a collection very efficient. However, if you want to optimize further the speed at which you can perform this functionality, you can do a couple of things.

First, you can consider *batching* your inserts. MongoDB ships with a multiple document `insert()` call. You can use this call to place several events into a collection at the same time. This results in fewer round-trips through the database interface API.

Second (and more importantly), you can reduce the size of your field names. If you have smaller field names, then MongoDB can pack more event records into memory before it has to flush them out to disk. This makes the whole system more efficient.

For example, assume you have a collection that is used to log three fields: a time stamp, a counter, and a four-character string used to indicate the source of the data. The total storage size of your data is shown in Table 10–2.

Table 10–2. *The Logging Example Collection Storage Size*

Field	Size
Timestamp	8 bytes
Integer	4 bytes
string	4 bytes
Total	16 bytes

If you use `ts`, `n`, and `src` for the field names, then the total size of the field names is 6 bytes. This is a relatively small value compared to the data size. But now assume you decided to name the fields `WhenTheEventHappened`, `NumberOfEvents`, and `SourceOfEvents`. In this case, the total size of the field names is 48 bytes, or three times the size of the data itself. If you wrote 1 TB of data into a collection, then you would be storing 750 GB of field names, but only 250 GB of actual data.

This does more than waste disk space. It also affects all other aspects of the system's performance, including the index size, data transfer time, and (probably more importantly) the use of precious system RAM to cache the data files.

Another recommendation for logging applications is that you need to avoid adding indexes on your collections when writing records; as explained earlier, indexes take time and resources to maintain. Instead, you should add the index immediately before you start analyzing the data.

Also, you should consider using a schema that splits the event stream into multiple collections. For example, you might write each day's events into a separate collection. Smaller collections take less time to index and analyze.

Summary

In this chapter, we looked at some tools for tracking down slow performance in MongoDB queries, as well as potential solutions for speeding up the slow queries that surface as a result. We also looked at some of the ways for optimizing data storage. For example, we looked at ways to ensure that we are making full use of the resources available to the MongoDB server.

The specific techniques described in this chapter enable you to optimize your data and tune the MongoDB system it is stored in. The best approach to take will vary from application to application, and it will be dependent on a lot of factors, including the application type, data access patterns, read/write ratios, and so on.