KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)
(For the candidates admitted from 2017 onwards)
DEPARTMENT OF CS, CA & IT

SUBJECT NAME : MONGODB
SEMESTER : II
SUBJECT CODE : 18CSP203 **CLASS: I M.Sc CS**

Instruction Hours / week: L: 4 T: 0 P: 0 **Marks:** Internal:**40** External:**60** Total: **100**
End Semester Exam : 3 Hours

Course Objectives
To provide students the knowledge and skills to master the NoSQL database mongoDB.

Course Outcomes(COs)
1. To provide students the right skills and knowledge needed to develop Applications on mongoDB

2. To provide students the right skills and knowledge needed to run Applications on mongoDB

Unit I - GETTING STARTED

A database for the modern web – MongoDB through the JavaScript shell – Writing programs using MongoDB.

Unit II - APPLICATION DEVELOPMENT

Document-oriented data – Principles of schema design – Designing an e-commerce data model – Nuts and bolts on databases, collections, and documents. Queries and aggregation – E-commerce queries – MongoDB"s query language – Aggregating orders – Aggregation in detail.

Unit III - UPDATES, ATOMIC OPERATIONS, AND DELETES

A brief tour of document updates – E-commerce updates – Atomic document processing – MongoDB updates and deletes. Indexing and query optimization: Indexing theory – Indexing in practice – Query optimization.

Unit IV – REPLICATION

Overview – Replica sets – Master-slave replication – Drivers and replication. Shading: Overview – A sample shard cluster – Querying and indexing a shard cluster – Choosing a shard key – sharding in production.

Unit V - DEPLOYMENT AND ADMINISTRATION

Deployment – Monitoring and diagnostics – Maintenance – Performance troubleshooting

SUGGESTED READINGS

1. Kyle Banker. (2012). MongoDB in Action. Manning Publications Co.
2. Rick Copeland. (2013). MongoDB Applied Design Patterns, 1st Edition, O"Reilly Media Inc.
3. Gautam Rege, (2012). Ruby and MongoDB Web Development Beginner's Guide. Packt Publishing Ltd
4. Mike Wilson.. (2013). Building Node Applications with MongoDB and Backbone, O"Reilly Media Inc.
5. David Hows. (2009). The definitive guide to MongoDB, 2nd edition, Apress Publication, 8132230485
6. Shakuntala Gupta Edward. 2016. Practical Mongo DB , 2nd edition, Apress Publications, 2016, ISBN 1484206487

WEBSITES

1. http://www.mongodb.org/about/production-deployments/
2. http://docs.mongodb.org/ecosystem/drivers/
3. http://www.mongodb.org/about/applications/
4. http://www.mongodb.org/

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)
(For the candidates admitted from 2017 onwards)
DEPARTMENT OF CS, CA & IT
LESSON PLAN
SUBJECT NAME : MONGODB (18CSP203)
SEMESTER : II

		UNIT I	
SI.NO	**Lecture Duration (Hr)**	**Topics to be covered**	**Support Materials**
1	1	Getting Started	T1:1
2	1	A database for the modern web	T1:3
3	1	A database for the modern web	T1:18
4	1	MongoDB through the JavaScript shell	T1:29
5	1	MongoDB through the JavaScript shell	T1:39, W1
6	1	Writing programs using MongoDB	T1:52
7	1	Writing programs using MongoDB	T1:59 R1:112
8	1	Recapitulation and Discussion of Important Question	
Total no. of Hours Planned for Unit I			**8**

		UNIT II	
SI.NO	**Lecture Duration (Hr)**	**Topics to be covered**	**Support Materials**
1	1	APPLICATION DEVELOPMENT - Document-oriented data	T1:71
2	1	Principles of schema design	T1:74
3	1	Designing an e-commerce data model	T1:75, R2:169
4	1	Nuts and bolts on databases, collections, and documents	T1:84
5	1	Queries and aggregation- E-commerce queries	T1:99, W2
6	1	MongoDB‟s query language	T1:103
7	1	Aggregating orders, Aggregation in detail	T1:120
8	1	Recapitulation and Discussion of Important Question	
Total Periods Planned for Unit II			**8**
		UNIT III	
SI.NO	**Lecture Duration (Hr)**	**Topics to be covered**	**Support Materials**
1	1	UPDATES, ATOMIC OPERATIONS, AND DELETES - A brief tour of document updates	T1:158

2	1	E-commerce updates	T1:162, R1:193
3	1	Atomic document processing	T1:171
4	1	MongoDB updates and deletes	T1:179
5	1	Indexing and query optimization: Indexing theory	T1:198
6	1	Indexing in practice	T1:207
7	1	Query optimization	T1:216, W2
8	1	Recapitulation and Discussion of Important Question	8
Total Periods Planned for Unit III			
		UNIT IV	
SI.NO	**Lecture Duration (Hr)**	**Topics to be covered**	**Support Materials**
1	1	REPLICATION- Overview, Replica sets	T1:297
2	1	Master Slave Replication – Drivers and Replication	T1:324
3	1	Shading: Overview	T1:334 R3:312
4	1	A sample shard cluster	T1:343

5	1	Querying and indexing a shard cluster	T1:355,w 2
6	1	Choosing a shard key	T1:359, w2
7	2	Sharding in production	T1:365
8	1	Recapitulation and Discussion of Important Question	
Total Periods Planned for Unit IV			**8**
UNIT V			
SI.NO	**Lecture Duration (Hr)**	**Topics to be covered**	**Support Materials**
1	1	DEPLOYMENT AND ADMINISTRATION - Deployment	
2	1	Monitoring and diagnostics	
3	1	Monitoring and diagnostics	
4	1	Maintenance	
5	1	Maintenance	
6	1	Performance troubleshooting	
7	2	Performance troubleshooting	
8	1	Recapitulation and Discussion of Important Question	
9	1	Discussion of Previous ESE Question Papers	

10	1	Discussion of Previous ESE Question Papers	
11	1	Discussion of Previous ESE Question Papers	
Total Periods Planned for Unit V			**12**
Total Periods			**44**

Text Book

T1	Kyle Banker. (2012). MongoDB in Action. Manning Publications Co.

References

R1	Rick Copeland. (2013). MongoDB Applied Design Patterns, 1st Edition, O‟Reilly Media Inc.
R2	Mike Wilson.(2013). Building Node Applications with MongoDB and Backbone, O‟Reilly Media Inc.
R3	Shakuntala Gupta Edward. 2016. Practical Mongo DB , 2nd edition, Apress Publications, 2016, ISBN 1484206487

Web Sites

w1	http://www.mongodb.org/
w2	W3schools.com/mongodb

Journals :

UNIT I
SYLLABUS

Getting Started: A database for the modern web – MongoDB through the JavaScript shell – Writing programs using MongoDB.

Getting Started: A database for the modern web

MongoDB is a database management system designed to rapidly develop web appli- cations and internet infrastructure. The data model and persistence strategies are built for high read-and-write throughput and the ability to scale easily with automatic failover. Whether an application requires just one database node or dozens of them, MongoDB can provide surprisingly good performance. If you've experienced difficul- ties scaling relational databases, this may be great news. But not everyone needs to operate at scale. Maybe all you've ever needed is a single database server.

MongoDB stores its information in documents rather than rows. What's a document? Here's an example:

```
{
  _id: 10,
  username: 'peter',
  email: 'pbbakkum@gmail.com'
}
```

This is a pretty simple document; it's storing a few fields of information about a user (he sounds cool). What's the advantage of this model? Consider the case where you'd like to store multiple emails for each user. In the relational world, you might create a separate table of email addresses and the users to which they're associated. MongoDB gives you another way to store these:

```
{
  _id: 10,
```

```
  username: 'peter',
  email: [
    'pbbakkum@gmail
    .com',
    'pbb7c@virginia.ed
    u'
  ]
}
```

MongoDB's document format is based on JSON, a popular scheme for storing arbi- trary data structures. JSON is an acronym for *JavaScript Object Notation*. As you just saw, JSON structures consist of keys and values, and they can nest arbitrarily deep. They're analogous to the dictionaries and hash maps of other programming languages.

A document-based data model can represent rich, hierarchical data structures. It's often possible to do without the multitable joins common to relational databases normalized relational data model, the information for any one product might be divided among dozens of tables.

Built for the internet

The history of MongoDB is brief but worth recounting, for it was born out of a much more ambitious project. In mid-2007, a startup in New York City called 10gen began work on a platform-as-a-service (PaaS), composed of an application server and a data- base, that would host web applications and scale them as needed. Like Google's App Engine, 10gen's platform was designed to handle the scaling and management of hardware and software infrastructure automatically, freeing developers to focus solely on their application code. 10gen ultimately discovered that most developers didn't feel comfortable giving up so much control over their technology stacks, but

users did

10gen has since changed its name to MongoDB, Inc. and continues to sponsor the database's development as an open source project. The code is publicly available and free to modify and use, subject to the terms of its license, and the community at large is encouraged to file bug reports and submit patches. Still, most of MongoDB's core developers are either founders or employees of MongoDB, Inc., and the project's roadmap continues to be determined by the needs of its user community and the overarching goal of creating a database that combines the best features of relational databases and distributed key-value stores.

MongoDB's key features

A database is defined in large part by its data model. In this section, you'll look at the document data model, and then you'll see the features of MongoDB that allow you to operate effectively on that model.

Document data model

MongoDB's data model is document-oriented. If you're not familiar with documents in the context of databases, the concept can be most easily demonstrated by an exam- ple. A JSON document needs double quotes everywhere except for numeric values. The following listing shows the JavaScript version of a JSON document where double quotes aren't necessary.

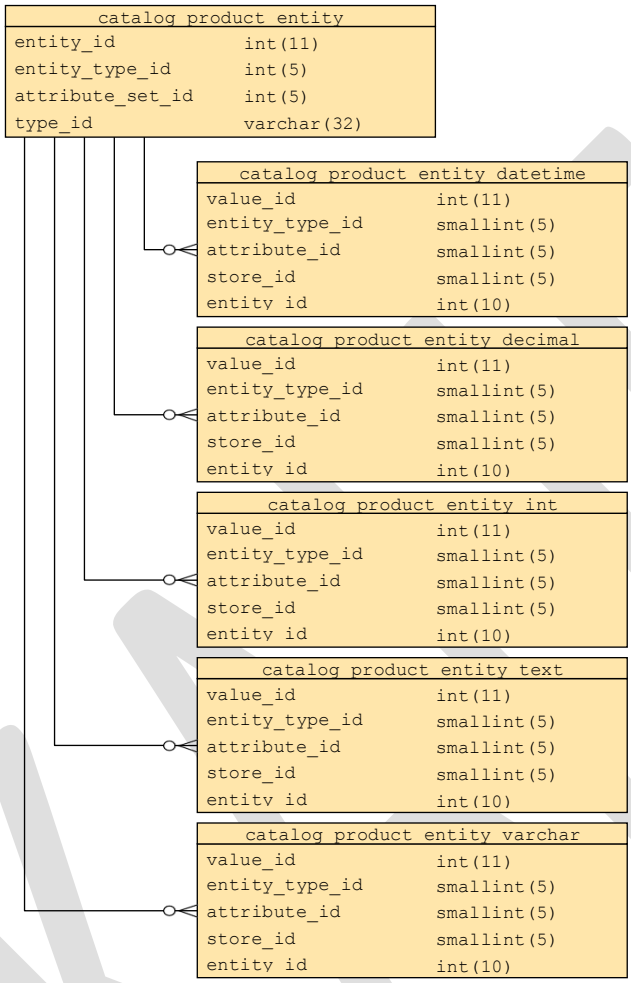
Listing 1.1 A document representing an entry on a social news site

```
{
  _id: ObjectID('4bd9e8e17cefd644108961bb'), title: 'Adventures in
  Databases',
  url: 'http://example.com/databases.txt', author: 'msmith',
```

```
vote_count: 20,
tags: ['databases', 'mongodb', 'indexing'], image: {
  url: 'http://example.com/db.jpg', caption: 'A database.',
  type: 'jpg', size: 75381, data: 'Binary'
},
                                comments: [
                                    {
      user: 'bjones',
      text: 'Interesting article.'
    },
    {
      user: 'sverch',
      text: 'Color me skeptical!'
    }
  ]
}
```

SCHEMA-LESS MODEL ADVANTAGES

This lack of imposed schema confers some advantages. First, your application code, and not the database, enforces the data's structure. This can speed up initial applica- tion development when the schema is changing frequently.



product catalog. There's no way of knowing in advance what attributes a product will have, so the application will need to account for that variability.

Ad hoc queries

Ad hoc queries are easy to take for granted if the only databases you've

ever used have been relational. But not all databases support dynamic queries. For instance, key-value stores are queryable on one axis only: the value's key.

A SQL query would look like this:

```
SELECT * FROM posts
  INNER JOIN posts_tags ON posts.id =
  posts_tags.post_id INNER JOIN tags ON
  posts_tags.tag_id == tags.id
  WHERE tags.text = 'politics' AND posts.vote_count > 10;
```

The equivalent query in MongoDB is specified using a document as a matcher. The special $gtkey indicates the greater-than condition:

```
db.posts.find({'tags': 'politics', 'vote_count': {'$gt': 10}});
```

Indexes

A critical element of ad hoc queries is that they search for values that you don't know when you create the database.

Indexes in MongoDB are implemented as a *B-tree* data structure. B-tree indexes, also used in many relational databases, are optimized for a variety of queries, includ- ing range scans and queries with sort clauses. But WiredTiger has support for log- structured merge-trees (LSM) that's expected to be available in the MongoDB 3.2 pro- duction release.

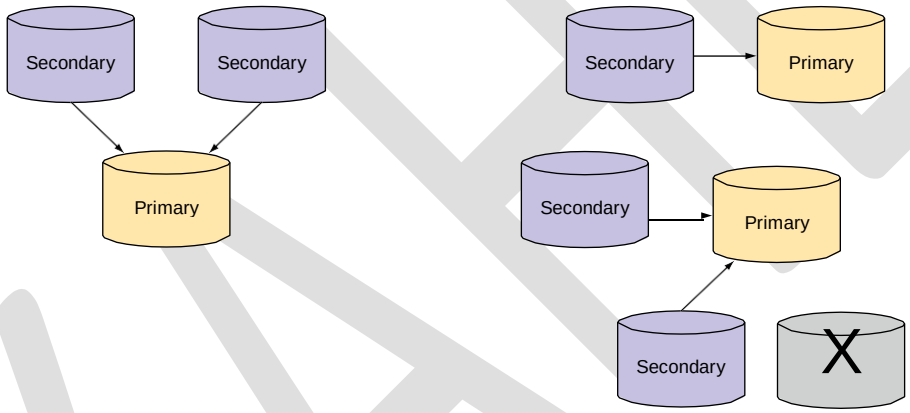
Replication

MongoDB provides database replication via a topology known as a replica set. *Replica sets* distribute data across two or more machines for redundancy and automate failover in the event of server and network outages. Additionally, replication is used to scale database reads. If you

have a read-intensive application, as is commonly the case on the web, it's possible to spread database reads across machines in the replica set cluster.

Speed and durability

To understand MongoDB's approach to durability, it pays to consider a few ideas first. In the realm of database systems there exists an inverse relationship between write speed and durability. *Write speed* can be understood as the volume of inserts, updates, and deletes that a database can process in a given time frame. *Durability* refers to level of assurance that these write operations have been made permanent.

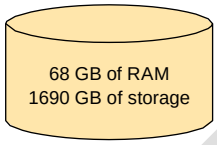


Scaling

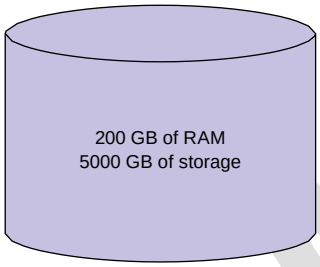
It then makes sense to consider scaling *horizontally*, or *scaling out* (see figure 1.4). Instead of beefing up a single node, scaling horizontally means distributing the data- base across multiple machines. A horizontally scaled architecture can run on many smaller, less expensive machines, often reducing your hosting costs.

MongoDB was designed to make horizontal scaling manageable. It does so via a range-based partitioning mechanism, known as *sharding*, which automatically manages

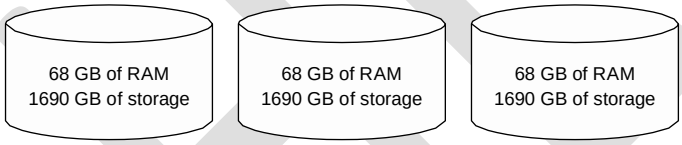
Original database



Scaling up increases the capacity of a single machine.



Scaling out adds more machines of similar size.



MongoDB's core server and tools

MongoDB is written in C++ and actively developed by MongoDB, Inc. The project compiles on all major operating systems, including Mac OS X, Windows, Solaris, and most flavors of Linux. Precompiled binaries are available for each of these platforms at http://mongodb.org. MongoDB is open source and licensed under the GNU-Affero General Public License (AGPL).

Core server

The core database server runs via an executable called mongod (mongodb.exe on Win- dows). The mongod server process receives commands over a network socket using a custom binary protocol. All the data files for a mongodprocess are stored by default in /data/db on Unix-like systems and in c:\data\db on Windows. Command-line tools

MongoDB is bundled with several command-line utilities:

- mongodump and mongorestore—Standard utilities for backing up and restoring a database. mongodump saves the database's data in its native BSON format and thus is best used for backups only; this tool has the advantage of being usable for hot backups, which can easily be restored with mongorestore.
- mongoexport and mongoimport—Export and import JSON, CSV, and TSV[7] data; this is useful if you need your data in widely supported formats. mongoimport can also be good for initial imports of large data sets, although before importing, it's often desirable to adjust the data model to take best advantage of MongoDB. In such cases, it's easier to import the data through one of the drivers using a custom script.
- mongosniff—A wire-sniffing tool for viewing operations sent to the database. It essentially translates the BSON going over the wire to human-readable shell statements.
- mongostat—Similar to iostat, this utility constantly polls MongoDB and the system to provide helpful stats, including the number of operations per second (inserts, queries, updates, deletes, and so on), the amount of virtual memory allocated, and the number of connections to the server.

- mongotop—Similar to top, this utility polls MongoDB and shows the amount of time it spends reading and writing data in each collection.
- mongoperf—Helps you understand the disk operations happening in a running MongoDB instance.

- mongooplog—Shows what's happening in the MongoDB oplog.
- Bsondump—Converts BSON files into human-readable formats including JSON. MongoDB versus other databases

The number of available databases has exploded, and weighing one against another can be difficult. Fortunately, most of these databases fall under one of a few catego- ries. In table 1.1, and in the sections that follow, we describe simple and sophisticated key-value stores, relational databases, and document databases, and show how these compare with MongoDB.

Table 1.1 Database families

	Examples	Data model	Scalability model	Use cases
Simple key-value stores	Memcached	Key-value, where the value is a binary blob.	Variable. Mem-cached can scale across nodes, converting all available RAM into a single, mono-lithic	Caching. Web ops.

			datastore.	
Sophisticated key-value stores	HBase, Cassandra, Riak KV, Redis, CouchDB	Variable. Cassandra uses a key- value structure known as a *col- umn*. HBase and Redis store binary blobs. CouchDB stores JSON documents.	Eventually consistent, multinode distribution for high availability and easy failover.	High-throughput verticals (activity feeds, message queues). Caching. Web ops.
Relational data- bases	Oracle Database, IBM DB2, Micro- soft SQL Server, MySQL, PostgreSQL	Tables.	Vertical scaling. Limited support for clustering and manual partition-ing.	System requiring transactions (banking, finance) or SQL. Normal-ized data model.

RELATIONAL DATABASES

Popular relational databases include MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database, IBM DB2, and so on; some are open-source and some are proprietary. MongoDB and rela- tional databases are both capable of representing a rich data model. Where relational databases use fixed-

schema tables, MongoDB has schema-free documents. Most rela- tional databases support secondary indexes and aggregations.

DOCUMENT DATABASES

Few databases identify themselves as document databases. As of this writing, the clos- est open-source database comparable to MongoDB is Apache's CouchDB. CouchDB's document model is similar, although data is stored in plain text as JSON, whereas MongoDB uses the BSON binary format. Like MongoDB, CouchDB supports secondary indexes; the difference is that the indexes in CouchDB are defined by writing map- reduce functions, a process that's more involved than using the declarative syntax used by MySQL and MongoDB. They also scale differently.

Use cases and production deployments

wEB APPLICATIONS

MongoDB can be a useful tool for powering a high-traffic website. This is the case with *The Business Insider (TBI)*, which has used MongoDB as its primary datastore since January 2008. TBI is a news site, although it gets substantial traffic, serving more than a million unique page views per day.

History of MongoDB

When the first edition of *MongoDB in Action* was released, MongoDB 1.8.x was the most recent stable version, with version 2.0.0 just around the corner. With this second edi- tion, 3.0.x is the latest stable version.[11]

A list of the biggest changes in each of the official versions is shown below. You should always use the most recent version available, if possible, in which case this list isn't particularly useful. If not, this list may help you determine how your version dif- fers from the content of this book. This is by no means an exhaustive list, and because of space constraints, we've listed only the top four or five items for each release.

VERSION 1.8.X (NO LONGER OFFICIALLY SUPPORTED)

- *Sharding*—Sharding was moved from "experimental" to production-ready status.
- *Replica sets*—Replica sets were made production-ready.
- *Replica pairs deprecated*—Replica set pairs are no longer supported by MongoDB, Inc.
- *Geo search*—Two-dimensional geo-indexing with coordinate pairs (2D indexes) was introduced.

VERSION 2.0.X (NO LONGER OFFICIALLY SUPPORTED)

- *Journaling enabled by default*—This version changed the default for new data- bases to enable journaling. Journaling is an important function that prevents data corruption.

- *$and queries*—This version added the $andquery operator to complement the $oroperator.
- *Sparse indexes*—Previous versions of MongoDB included nodes in an index for every document, even if the document didn't contain any of the fields being tracked by the index. Sparse indexing adds only document nodes that have rel- evant fields. This feature significantly reduces index size. In some cases this can improve performance because smaller indexes can result in more efficient use of memory.
- *Replica set priorities*—This version allows "weighting" of replica set members to ensure that your best servers get priority when electing a new primary server.
- *Collection level compact/repair*—Previously you could perform compact/repair only on a database; this enhancement extends it to individual collections.

VERSION 2.2.X (NO LONGER OFFICIALLY SUPPORTED)

- *Aggregation framework*—This version features the first iteration of a facility to make analysis and transformation of data much easier and

more efficient. In many respects this facility takes over where map/reduce leaves off; it's built on a pipeline paradigm, instead of the map/reduce model (which some find diffi- cult to grasp).

- *TTL collections*—Collections in which the documents have a time-limited lifespan are introduced to allow you to create caching models such as those provided by Memcached.
- *DB level locking*—This version adds database level locking to take the place of the global lock, which improves the write concurrency by allowing multiple opera- tions to happen simultaneously on different databases.
- *Tag-aware sharding*—This version allows nodes to be tagged with IDs that reflect their physical location. In this way, applications can control where data is stored in clusters, thus increasing efficiency (read-only nodes reside in the same data center) and reducing legal jurisdiction issues (you store data required to remain in a specific country only on servers in that country).

VERSION 2.4.X (OLDEST STABLE RELEASE)

- *Enterprise version*—The first subscriber-only edition of MongoDB, the Enterprise version of MongoDB includes an additional authentication module that allows the use of Kerberos authentication systems to manage database login data. The free version has all the other features of the Enterprise version.

- *Aggregation framework performance*—Improvements are made in the performance of the aggregation framework to support real-time analytics; chapter 6 explores the Aggregation framework.

- *Text search*—An enterprise-class search solution is integrated as an experimental feature in MongoDB; chapter 9 explores the new text search features.
- *Enhancements to geospatial indexing*—This version includes support for polygon intersection queries and GeoJSON, and features

an improved spherical model supporting ellipsoids.

- *V8 JavaScript engine*—MongoDB has switched from the Spider Monkey JavaScript engine to the Google V8 Engine; this move improves multithreaded operation and opens up future performance gains in MongoDB's JavaScript-based map/ reduce system.

VERSION 2.6.X (STABLE RELEASE)

- *$text queries*—This version added the $text query operator to support text search in normal find queries.
- *Aggregation improvements*—Aggregation has various improvements in this ver- sion. It can stream data over cursors, it can output to collections, and it has many new supported operators and pipeline stages, among many other features and performance improvements.

Additional resources

- *Improved wire protocol for writes*—Now bulk writes will receive more granular and detailed responses regarding the success or failure of individual writes in a batch, thanks to improvements in the way errors are returned over the network for write operations.
- *New update operators*—New operators have been added for update operations, such as $mul, which multiplies the field value by the given amount.
- *Sharding improvements*—Improvements have been made in sharding to better handle certain edge cases. Contiguous chunks can now be merged, and dupli- cate data that was left behind after a chunk migration can be cleaned up auto- matically.
- *Security improvements*—Collection-level access control is supported in this ver- sion, as well as user-defined roles. Improvements have also been made in SSL and x509 support.
- *Query system improvements* —Much of the query system has been refactored. This improves performance and predictability of queries.
- *Enterprise module*—The MongoDB Enterprise module has improvements and extensions of existing features, as well as support for auditing.

VERSION 3.0.X (NEWEST STABLE RELEASE)

- The MMAPv1 storage engine now has support for collection-level locking
- Replica sets can now have up to 50 members.
- Support for the WiredTiger storage engine; WiredTiger is only available in the 64-bit versions of MongoDB 3.0.
- The 3.0 WiredTiger storage engine provides document-level locking and compression.
- Pluggable storage engine API that allows third parties to develop storage engines for MongoDB.
- Improved explain functionality.
- SCRAM-SHA-1 authentication mechanism.
- The ensureIndex() function has been replaced by the createIndex() function and should no longer be used.

This topic covers

- Using CRUD operations in the MongoDB shell
- Building indexes and using `explain()`
- Understanding basic administration
- Getting help

Diving into the MongoDB shell

MongoDB's JavaScript shell makes it easy to play with data and get a tangible sense of documents, collections, and the database's particular query language. Think of the following walkthrough as a practical introduction to MongoDB.

Starting the shell

Follow the instructions in appendix A and you should quickly have a working MongoDB installation on your computer, as well as a running mongod instance. Once you do, start the MongoDB shell by running the mongo executable:

mongo

If the shell program starts successfully, your screen will look like figure 2.1. The shell heading displays the version of MongoDB you're running, along with some additional information about the currently selected database.

```
10:25 $ mongo
MongoDB shell version: 3.0.4
connecting to: test
>
```

Databases, collections, and documents

MongoDB divides collections into separate *databases*. Unlike the usual overhead that databases produce in the SQL world, databases in MongoDB are just namespaces to distinguish between collections. To query MongoDB, you'll need to know the data- base (or namespace) and collection you want to query for documents. If no other database is specified on startup, the shell selects a default database called test. As a way of keeping all the subsequent tutorial exercises under the same namespace, let's start by switching to the tutorial database:

```
> use tutorial
switched to db tutorial
```

The document contains a single key and value for storing Smith's username.

Inserts and queries

To save this document, you need to choose a collection to save it to.

Appropriately enough, you'll save it to the userscollection. Here's how:

```
> db.users.insert({username:
"smith"}) WriteResult({ "nInserted"
: 1 })
```

NOTE Note that in our examples, we'll preface MongoDB shell commands with a >so that you can tell the difference between the command and its output.

You may notice a slight delay after entering this code. At this point, neither the tuto- rial database nor the users collection has been created on disk. The delay is caused by the allocation of the initial data files for both.

If the insert succeeds, you've just saved your first document. In the default MongoDB configuration, this data is now guaranteed to be inserted even if you kill the shell or suddenly restart your machine. You can issue a query to see the new document:

```
> db.users.find()
```

Since the data is now part of the users collection, reopening the shell and running the query will show the same result. The response will look something like this:

```
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

PASS A QUERY PREDICATE

Now that you have more than one document in the collection, let's look at some slightly more sophisticated queries. As before, you can still query for all the docu- ments in the collection:

```
> db.users.find()
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

You can also pass a simple query selector to the find method. A query selector is a document that's used to match against all documents in the collection. To query for all documents where the username is jones, you pass a simple document that acts as your query selector like this:

```
> db.users.find({username: "jones"})
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

Updating documents

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

OPERATOR UPDATE

The first type of update involves passing a document with some kind of operator description as the second argument to the update function. In this section, you'll see an example of how to use the $set operator, which sets a single field to the spec- ified value.

Suppose that user Smith decides to add her country of residence. You can record this with the following update:

```
> db.users.update({username: "smith"}, {$set: {country:
"Canada"}}) WriteResult({ "nMatched" : 1, "nUpserted" : 0,
"nModified" : 1 })
```

Deleting data

If given no parameters, a remove operation will clear a collection of all its docu- ments. To get rid of, say, a foocollection's contents, you enter:

```
> db.foo.remove()
```

You often need to remove only a certain subset of a collection's documents, and for that, you can pass a query selector to the remove() method. If you want to remove all users whose favorite city is Cheyenne, the expression is straightforward:

```
> db.users.remove({"favorites.cities":
"Cheyenne"}) WriteResult({ "nRemoved" : 1 })
```

Note that the remove() operation doesn't actually delete the collection; it merely removes documents from a collection. You can think of it as being analogous to SQL's DELETE command.

If your intent is to delete the collection along with all of its indexes, use the drop() method:

```
> db.users.drop()
```

Basic administration

Getting database information

showdbsprints a list of all the databases on the system:

```
> show dbs
admin    (empty)
local
          0.078
GB tutorial
0.078GB
```

showcollections displays a list of all the collections defined on the current data- base.[4] If the tutorial database is still selected, you'll see a list of the collections you worked with in the preceding tutorial:

```
> show
collections
numbers
system.indexes
users
```

The one collection that you may not recognize is system.indexes. This is a special collection that exists for every database. Each entry in

system.indexes defines an index for the database, which you can view using the getIndexes()method, as you saw earlier.

But MongoDB 3.0 deprecates direct access to the system.indexes collections; you should use createIndexes and listIndexes instead. The getIndexes() Java- Script method can be replaced by the db.runCommand({"listIndexes": "numbers"}) shell command.

For lower-level insight into databases and collections, the stats()method proves useful. When you run it on a database object, you'll get the following output:

```
> db.stats()
{
    "db" : "tutorial",
    "collections" : 4,
    "objects" : 20010,
    "avgObjSize" : 48.0223888055972,
    "dataSize" : 960928,
    "storageSize" : 2818048,
    "numExtents" : 8,
    "indexes" : 3,
    "indexSize" : 1177344,
    "fileSize" : 67108864,
    "nsSizeMB" : 16,
    "extentFreeList" :
    {
        "num" : 0,
        "totalSize" : 0
    },
    "dataFileVersion" :
        { "major" : 4,
        "minor" : 5
    },
    "ok" : 1
}
```

This topic covers

- Introducing the MongoDB API through Ruby
- Understanding how the drivers work
- Using the BSON format and MongoDB network protocol
- Building a complete sample application

MongoDB through the Ruby lens

Installing and connecting

Once you have RubyGems installed, run:

```
gem install mongo
```

You'll start by connecting to MongoDB. First, make sure that mongodis running by running the mongoshell to ensure you can connect. Next, create a file called connect.rb and enter the following code:

```
require
'rubygems'
require 'mongo'

$client = Mongo::Client.new([ '127.0.0.1:27017' ], :database =>
'tutorial') Mongo::Logger.logger.level = ::Logger::ERROR
$users =
$client[:users] puts
'connected!'
```

The first two require statements ensure that you've loaded the driver. The next three lines instantiate the client to localhost and connect to the tutorial database, store a ref- erence to the userscollection in the $usersvariable, and print the string connected!. We place a $in front of each variable to make it global so that it'll be accessible out- side of the connect.rb script. Save

the file and run it:

```
$ ruby connect.rb
D, [2015-06-05T12:32:38.843933 #33946] DEBUG -- : MONGODB | Adding
    127.0.0.1:27017 to the cluster. | runtime: 0.0031ms
D, [2015-06-05T12:32:38.847534 #33946] DEBUG -- : MONGODB |
COMMAND |
    namespace=admin.$cmd selector={:ismaster=>1} flags=[]
    limit=-1 skip=0 project=nil | runtime: 3.4170ms
connected!
```

Inserting documents in Ruby

To run interesting MongoDB queries you first need some data, so let's create some (this is the C in CRUD). All of the MongoDB drivers are designed to use the most natu- ral document representation for their language. In JavaScript, JSON objects are the obvious choice, because JSON is a document data structure; in Ruby, the hash data structure makes the most sense. The native Ruby hash differs from a JSON object in only a couple of small ways; most notably, where JSON separates keys and values with a colon, Ruby uses a hash rocket (=>).[2]

Here's an example:

```
$ irb -r ./connect.rb
irb(main):017:0> id = $users.insert_one({"last_name" => "mtsouk"})
=> #<Mongo::Operation::Result:70275279152800
documents=[{"ok"=>1, "n"=>1}]> irb(main):014:0>
$users.find().each do |user|
irb(main):015:1* puts user
irb(main):016:1> end
```

```
{"_id"=>BSON::ObjectId('55e3ee1c5ae119511d000000'),
"last_name"=>"knuth"}
{"_id"=>BSON::ObjectId('55e3f13d5ae119516a000000'),
"last_name"=>"mtsouk"}
=> #<Enumerator: #<Mongo::Cursor:0x70275279317980
@view=#<Mongo::Collection::View:0x70275279322740
```

namespace='tutorial.users @selector={} @options={}>>:each>

Updates and deletes

$users.find({"last_name" => "smith"}).update_one({"$set" => {"city" => "Chicago"}})

This update finds the first user with a last_name of smith and, if found, sets the value of city to Chicago. This update uses the $set operator. You can run a query to show the change:

$users.find({"last_name" => "smith"}).to_a

Database commands

First, you instantiate a Ruby database object referencing the admin database. You then pass the command's query specification to the command method:

```
$admin_db = $client.use('admin')
$admin_db.command({"listDatabases" => 1})
```

Note that this code still depends on what we put in the connect.rb script above because it expects the MongoDB connection to be in $client. The response is a Ruby hash listing all the existing databases and their sizes on disk:

```
#<Mongo::Operation::Result:70112905054200 documents=[{"databases"=>[
{
    "name"=>"local",
    "sizeOnDisk"=>83886
    080.0, "empty"=>false
},
{
    "name"=>"tutorial",
"sizeOnDisk"=>83886
080.0, "empty"=>false
```

```
},
{
    "name"=>"admin",
    "sizeOnDisk"=>1.0,
    "empty"=>true
}], "totalSize"=>167772160.0, "ok"=>1.0}]>
 => nil
```

How the drivers work

All MongoDB drivers perform three major functions. First, they generate Mon- goDB object IDs. These are the default values stored in the _id field of all documents. Next, the drivers convert any language-specific representation of documents to and from BSON, the binary data format used by MongoDB. In the previous examples, the driver serializes all the Ruby hashes into BSON and then deserializes the BSON that's returned from the database back to Ruby hashes.

The drivers' final function is to communicate with the database over a TCP socket using the MongoDB wire protocol. The details of the protocol are beyond the scope of this discussion. But the style of socket communication, in particular whether writes on the socket wait for a response, is important, and we'll explore the topic in this section.

Object ID generation

Every MongoDB document requires a primary key. That key, which must be unique for all documents in each collection, is stored in the document's _id field. Developers are free to use their own custom values as the _id, but when not provided, a MongoDB object ID will be used. Before sending a document to the server, the driver checks whether the _idfield is present. If the field is missing, an object ID will be generated and stored as _id.

MongoDB object IDs are designed to be globally unique, meaning they're guaran- teed to be unique within a certain context. How can this be guaranteed? Let's exam- ine this in more detail.

You've probably seen object IDs in the wild if you've inserted documents into MongoDB, and at first glance they appear to be a string of mostly random text, like 4c291856238d3b19b2000001.

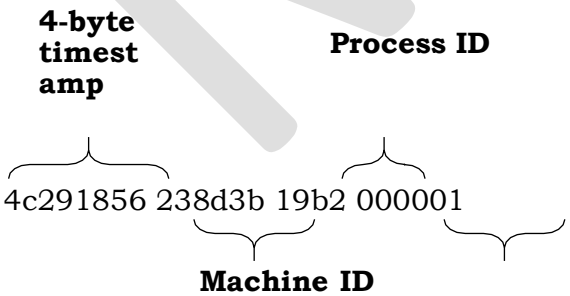


Figure 3.1 MongoDB object ID format

representation of 12 bytes, and actually stores some useful information. These bytes have a specific structure, as illustrated in figure 3.1.

The most significant four bytes carry a standard Unix (epoch) timestamp[3]. The next three bytes store the machine ID, which is followed by a two-byte process ID. The final three bytes store a process-local counter that's incremented each time an object ID is generated. The counter means that ids generated in the same process and second won't be duplicated.

Why does the object ID have this format? It's important to understand that these IDs are generated in the driver, not on the server. This is different than many RDBMSs, which increment a primary key on the server, thus creating a bottleneck for the server generating the key. If more than one driver is generating IDs and inserting docu- ments, they need a way of creating unique identifiers without talking to each other. Thus, the timestamp, machine ID, and process ID are included in the identifier itself to make it extremely unlikely that IDs will overlap.

You may already be considering the odds of this happening. In practice, you would encounter other limits before inserting documents at the rate required to overflow the counter for a given second (2^{24} million per second). It's slightly more conceivable (though still unlikely) to imagine that ifyou had many drivers distributed across many machines, two machines could have the same machine ID. For example, the Ruby driver uses the following:

@@machine_id = Digest::MD5.digest(Socket.gethostname)[0, 3]

For this to be a problem, they would still have to have started the MongoDB driver's process with the same process ID, and have the same counter value in a given second. In practice, don't worry about duplication; it's extremely unlikely.

One of the incidental benefits of using MongoDB object IDs is that they include a timestamp. Most of the drivers allow you to extract the timestamp, thus providing the document creation time, with resolution to the nearest second, for free. Using the Ruby

driver, you can call an object ID's generation_time method to get that ID's creation time as a Ruby Time object:

```
irb> require 'mongo'
irb> id = BSON::ObjectId.from_string('4c291856238d3b19b2000001')
=> BSON::ObjectId('4c291856238d3b19b2000001')
irb> id.generation_time
=> 2010-06-28 21:47:02 UTC
```

Naturally, you can also use object IDs to issue range queries on object creation time. For instance, if you wanted to query for all documents created during June 2013, you could create two object IDs whose timestamps encode those dates and then issue a range query on _id. Because Ruby provides methods for generating object IDs from any Time object, the code for doing this is trivial:[4]

```
jun_id = BSON::ObjectId.from_time(Time.utc(2013, 6, 1))
jul_id = BSON::ObjectId.from_time(Time.utc(2013, 7,
1)) @users.find({'_id' => {'$gte' => jun_id, '$lt' =>
jul_id}})
```

As mentioned before, you can also set your own value for _id. This might make sense in cases where one of the document's fields is important and always unique. For instance, in a collection of users you could store the username in _id rather than on object ID. There are advantages to both ways, and it comes down to your preference as a developer.

Building a simple application

Next you'll build a simple application for archiving and displaying Tweets. You can imagine this being a component in a larger application that allows users to keep tabs on search terms relevant to their businesses. This example will demonstrate how easy it is to consume JSON from an API like Twitter's

and convert that to MongoDB docu- ments. If you were doing this with a relational database, you'd have to devise a schema in advance, probably consisting of multiple tables, and then declare those tables. Here, none of that's required, yet you'll still preserve the rich structure of the Tweet documents, and you'll be able to query them effectively.

Let's call the app TweetArchiver. TweetArchiver will consist of two components: the archiver and the viewer. The archiver will call the Twitter search API and store the relevant Tweets, and the viewer will display the results in a web browser.

Setting up

This application requires four Ruby libraries. The source code repository for this chap- ter includes a file called Gemfile, which lists these gems. Change your working directory

```
gem install bundler bundle install
```

This will ensure the bundler gem is installed. Next, install the other gems using Bundler's package management tools. This is a widely used Ruby tool for ensuring that the gems you use match some predetermined versions: the versions that match our code examples.

Our Gemfile lists the mongo, twitter, bson and sinatra gems, so these will be installed. The mongo gem we've used already, but we include it to be sure we have the right version. The twitter gem is useful for communicating with the Twitter API.

We provide the source code for this example separately, but introduce it gradually to help you understand it. We recommend you experiment and try new things to get the most out of the example.

It'll be useful to have a configuration file that you can share between the archiver and viewer scripts. Create a file called config.rb (or copy it from the source code) that looks like this:

```
DATABASE_HOST =
'localhost'
```

```
DATABASE_PORT = 27017
DATABASE_NAME = "twitter-archive" COLLECTION_NAME = "tweets"
TAGS = ["#MongoDB", "#Mongo"]

CONSUMER_KEY = "replace me"
CONSUMER_SECRET = "replace me" TOKEN = "replace me" TOKEN_SECRET = "replace me"
```

First you specify the names of the database and collection you'll use for your applica- tion. Then you define an array of search terms, which you'll send to the Twitter API.

Twitter requires that you register a free account and an application for accessing the API, which can be accomplished at http://apps.twitter.com. Once you've regis- tered an application, you should see a page with its authentication information, per- haps on the API keys tab. You will also have to click the button that creates your access token. Use the values shown to fill in the consumer and API keys and secrets.

Gathering data

The next step is to write the archiver script. You start with a TweetArchiver class. You'll instantiate the class with a search term. Then you'll call the update method on the TweetArchiver instance, which issues a Twitter API call, and save the results to a MongoDB collection.

Let's start with the class's constructor:

```
def initialize(tag)
  connection = Mongo::Connection.new(DATABASE_HOST, DATABASE_PORT) db = connection[DATABASE_NAME]
  @tweets = db[COLLECTION_NAME]
  @tweets.ensure_index([['tags', 1], ['id', -1]])
```

```
  @tag = tag
  @tweets_found = 0


  @client = Twitter::REST::Client.new do
    |config| config.consumer_key        =
    API_KEY config.consumer_secret    =
    API_SECRET config.access_token    =
    ACCESS_TOKEN
    config.access_token_secret =
    ACCESS_TOKEN_SECRET
  en
d
end
```

The initialize method instantiates a connection, a database object, and the collec- tion object you'll use to store the Tweets.

You're creating a compound index on tags ascending and id descending. Because you're going to want to query for a particular tag and show the results from newest to oldest, an index with tags ascending and id descending will make that query use the index both for filtering results and for sorting them. As you can see here, you indicate index direction with 1 for *ascending* and -1 for *descending*. Don't worry if this doesn't make sense now—we discuss indexes with much greater depth in chapter 8.

You're also configuring the Twitter client with the authentication information from config.rb. This step hands these values to the Twitter gem, which will use them when calling the Twitter API. Ruby has somewhat unique syntax often used for this sort of con- figuration; the configvariable is passed to a Ruby block, in which you set its values.

In the future, Twitter may change its API so that different values are returned, which will likely require a schema change if you want to store these additional values. Not so with MongoDB. Its schema-less design allows you to save the document you get from the Twitter API without worrying about the exact format.

The Ruby Twitter library returns Ruby hashes, so you can pass these directly to your MongoDB collection object. Within your TweetArchiver, you add the following instance method:

```
def save_tweets_for(term)
  @client.search(term).each do
  |tweet|
    @tweets_found += 1
    tweet_doc =
    tweet.to_h
    tweet_doc[:tags] =
    term
    tweet_doc[:_id] =
    tweet_doc[:id]
    @tweets.insert_one(tweet_do
    c)
  en
d
end
```

Before saving each Tweet document, make two small modifications. To simplify later queries, add the search term to a tagsattribute. You also set the _id field to the ID of the Tweet, replacing the primary key of your collection and ensuring that each Tweet is added only once. Then you pass the modified document to the savemethod.

To use this code in a class, you need some additional code. First, you must config- ure the MongoDB driver so that it connects to the correct mongodand uses the desired database and collection. This is simple code that you'll replicate often as you use MongoDB. Next, you must configure the Twitter gem with your developer credentials. This step is necessary because Twitter restricts its API to registered developers. The next listing

also provides an update method, which gives the user feedback and calls save_tweets_for.

Listing 3.1 archiver.rb—A class for fetching Tweets and archiving them in MongoDB

```
$LOAD_PATH << File.dirname(_
FILE_) require 'rubygems'
require 'mongo'
require 'twitter'
require 'config'

class TweetArchiver

def initialize(tag)
client =
Mongo::Client.new(["#{DATABASE_HOST}:#{DATABASE_PORT}"],:
database => "#{DATABASE_NAME}")
@tweets =
client["#{COLLECTION_NAME}"]
@tweets.indexes.drop_all
@tweets.indexes.create_many([
{ :key => { tags: 1 }},
{ :key => { id: -1 }}
])
@tag = tag
@tweets_found = 0

    client = Twitter::REST::Client.new do |config| config.consumer_key =
      "#{API_KEY}" config.consumer_secret = "#{API_SECRET}"
      config.access_token = "#{ACCESS_TOKEN}"
      config.access_token_secret = "#{ACCESS_TOKEN_SECRET}"
    end end
```

Configure the Twitter client using the values found in config.rb.

```
def update
  puts "Starting Twitter search for '#{@tag}'..." save_tweets_for(@tag)
  print "#{@tweets_found} Tweets saved.\n\n" end

private
```

A user facing method to wrap save_tweets_for

```
  def save_tweets_for(term) @client.search(term).each do |tweet|
      @tweets_found += 1 tweet_doc = tweet.to_h tweet_doc[:tags] =
      term
      tweet_doc[:_id] = tweet_doc[:id] @tweets.insert_one(tweet_doc)
    end end
end
```

Search with the Twitter client and save the results to Mongo.

All that remains is to write a script to run the TweetArchiver code against each of the search terms. Create a file called update.rb (or copy it from the provided code) con- taining the following:

```
$LOAD_PATH << File.dirname(
FILE_) require 'config'
require 'archiver'

TAGS.each do |tag|
  archive =
```

```
  TweetArchiver.new(tag)
  archive.update
end
```

Next, run the update script:

```
ruby update.rb
```

You'll see some status messages indicating that Tweets have been found and saved. You can verify that the script works by opening the MongoDB shell and querying the col- lection directly:

```
> use twitter-archive
switched to db twitter-archive
> db.tweets.coun
t() 30
```

What's important here is that you've managed to store Tweets from Twitter searches in only a few lines of code.[5] Next comes the task of displaying the results.

Viewing the archive

You'll use Ruby's Sinatra web framework to build a simple app to display the results. Sinatra allows you to define the endpoints for a web application and directly specify the response. Its power lies in its simplicity. For example, the content of the index page for your application can be specified with the following:

```
get '/' do
  "respons
  e"
end
```

This code specifies that GET requests to the / endpoint of your application return the

value of response to the client. Using this format, you can write full web applications with many endpoints, each of which can execute arbitrary Ruby code before returning a response. You can find more information, including Sinatra's full documentation, at http://sinatrarb.com.

We'll now introduce a file called viewer.rb and place it in the same directory as the other scripts. Next, make a subdirectory called views, and place a file there called tweets.erb. After these steps, the project's file structure should look like this:

- config.rb
- archiver.rb
- update.rb
- viewer.rb
- /views
  - tweets.erb

Again, feel free to create these files yourself or copy them from the code examples. Now edit viewer.rb with the code in the following listing.

Listing 3.2 viewer.rb—Sinatra application for displaying the Tweet archive

```
$LOAD_PATH << File.dirname(_
FILE_) require 'rubygems'
require 'mongo'
require 'sinatra'
require 'config'
require 'open-uri'

configure do
  client = Mongo::Client.new(["#{DATABASE_HOST}:#{DATABASE_PORT}"],
  :database
      => "#{DATABASE_NAME}")
```

Required libraries

```
  TWEETS = client["#{COLLECTION_NAME}"]
end
```

```
get '/' do
  if params['tag']
    selector = {:tags => params['tag']} else
    selector = {} end
```

Instantiate collection
c for tweets

d Dynamically build query selector...

...or use
e blank selector

```
  @tweets = TWEETS.find(selector).sort(["id", -1]) erb :tweets
end
```

The first lines require the necessary libraries along with your config file B. Next there's a configuration block that creates a connection to MongoDB and stores a refer- ence to your tweets collection in the constant TWEETSc.

The real meat of the application is in the lines beginning with get'/'do. The code in this block handles requests to the application's root URL. First, you build your

query selector. If a tagsURL parameter has been provided, you create a query selector that restricts the result set to the given tags d. Otherwise, you create a blank selector, which returns all documents in the collection e. You then

issue the query f. By now, you should know that what gets assigned to the @tweetsvariable isn't a result set but a
cursor. You'll iterate over that cursor in your view.

The last line g renders the view file tweets.erb (see the next listing).

Listing 3.3 tweets.erb—HTML with embedded Ruby for rendering the Tweets

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <style>
    body
    {
      width: 1000px;
      margin: 50px
      auto;
      font-family: Palatino,
      serif; background-color:
      #dbd4c2; color: #555050;
    }
    h2 {
      margin-top: 2em;
      font-family: Arial, sans-
      serif; font-weight: 100;
    }
  </style>
</head>
<body>
<h1>Tweet Archive</h1>
<% TAGS.each do |tag| %>
  <a href="/?tag=<%= URI::encode(tag) %>"><%= tag %></a>
<% end %>
```

```
<% @tweets.each do |tweet| %>
  <h2><%= tweet['text'] %></h2>
  <p>
    <a href="http://twitter.com/<%= tweet['user']['screen_name'] %>">
      <%= tweet['user']['screen_name'] %>
    </a>
    on <%= tweet['created_at'] %>
  </p>
  <img src="<%= tweet['user']['profile_image_url'] %>" width="48" />
<% end %>
</body>
</html>
```

Most of the code is just HTML with some ERB (embedded Ruby) mixed in. The Sinatra app runs the tweets.erb file through an ERB processor and evaluates any Ruby code between <% and %> in the context of the application.

The important parts come near the end, with the two iterators. The first of these cycles through the list of tags to display links for restricting the result set to a given tag.

The second iterator, beginning with the @tweets.eachcode, cycles through each Tweet to display the Tweet's text, creation date, and user profile image. You can see results by running the application:

```
$ ruby viewer.rb
```

If the application starts without error, you'll see the standard Sinatra startup message that looks something like this:

```
$ ruby viewer.rb
[2013-07-05 18:30:19] INFO WEBrick 1.3.1
[2013-07-05 18:30:19] INFO ruby 1.9.3 (2012-04-20) [x86_64-darwin10.8.0]
== Sinatra/1.4.3 has taken the stage on 4567 for development with
     backup from WEBrick
[2013-07-05 18:30:19] INFO WEBrick::HTTPServer#start: pid=18465
```

port=4567

You can then point your web browser to http://localhost:4567. The page should look something like the screenshot in figure 3.2. Try clicking on the links at the top of the screen to narrow the results to a particular tag.

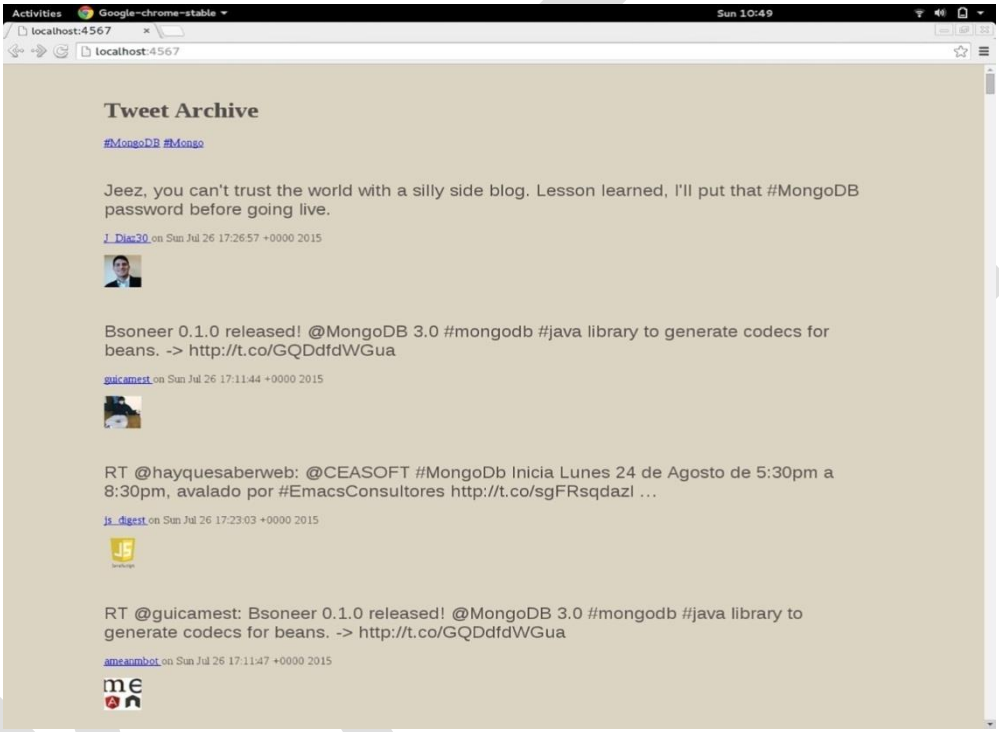


Figure 3.2 Tweet Archiver output rendered in a web browser

69

UNIT II
SYLLABUS

Application Development: Document-oriented data – Principles of schema design – Designing an e-commerce data model – Nuts and bolts on databases, collections, and documents. Queries and aggregation – E-commerce queries – MongoDB"s query language – Aggregating orders – Aggregation in detail.

This topic covers

- Schema design
- Data models for e-commerce
- Nuts and bolts of databases, collections, and documents

Principles of schema design

Database schema design is the process of choosing the best representation for a data set, given the features of the database system, the nature of the data, and the applica- tion requirements. The principles of schema design for relational database systems are well established. With RDBMSs, you're encouraged to shoot for a normalized data model,[1] which helps to ensure generic query ability and avoid updates to data that might result in inconsistencies. Moreover, the established patterns prevent developers from wondering how to model, say, one-to-many and many-to-many relationships.

What are your application access patterns?

What's the basic unit of data?

What are the capabilities of your database?

What makes a good unique id or primary key for a record?

Designing an e-commerce data model

E-commerce has the advantage of including a large number of famil- iar data modeling patterns. Plus, it's not hard to imagine how products, categories, product reviews, and orders are typically modeled in an RDBMS.

E-commerce has typically been done with RDBMSs for a couple of reasons. The first is that e-commerce sites generally require transactions, and transactions are an RDBMS staple.

The second is that, until recently, domains that require rich data models and sophisticated queries have been assumed to fit best within the realm of the RDBMS.

Schema basics

Products and categories are the mainstays of any e-commerce site. Products, in a nor- malized RDBMS model, tend to require a large number of tables. There's a table for basic product information, such as the name and SKU, but there will be other tables to relate shipping information and pricing histories.

This multitable schema will be facil- itated by the RDBMS's ability to join tables.

More concretely, listing 4.1 shows a sample product from a gardening store. It's advis- able to assign this document to a variable before inserting it to the database using db.products.insert(yourVariable) to be able to run the queries discussed over the next several pages.

Listing 4.1 A sample product document

```
{
  _id: ObjectId("4c4b1476238d3b4dd5003981"), slug: "wheelbarrow-9092",
  sku: "9092",
  name: "Extra Large Wheelbarrow", description: "Heavy duty
  wheelbarrow...", details: {
    weight: 47, weight_units: "lbs", model_num: 4039283402,
    manufacturer: "Acme", color: "Green"
  },
  total_reviews: 4,
  average_review: 4.5, pricing: {
                          retail: 589700,
                          sale: 489700,
```

```
    },
    price_history: [
                        {
                         retail: 529700,
                         sale: 429700,
        start: new Date(2010, 4, 1),
        end: new Date(2010, 4, 8)
                        },

                        {
                         retail: 529700,
                         sale: 529700,
        start: new Date(2010, 4, 9),
        end: new Date(2010, 4, 16)
```

b **Unique object ID**

c **Unique slug**

Nested
d **document**

e **One-to-many relationship**

```
      },
    ],

    primary_category: ObjectId("6a5b1476238d3b4dd5000048"),
    category_ids: [
      ObjectId("6a5b1476238d3b4dd5000048"),
      ObjectId("6a5b1476238d3b4dd5000049")
    ],
    main_cat_id: ObjectId("6a5b1476238d3b4dd5000048"),
    tags: ["tools", "gardening", "soil"],
}
```

Many-to-many relationship

ONE-TO-MANY RELATIONSHIPS

This is a one-to-many relationship, since a product only has one primary category, but a category can be the primary for many products.

MANY-TO-MANY RELATIONSHIPS

MongoDB doesn't support joins, so you need a different many-to-many strategy. We've defined a field called category_ids f containing an array of object IDs. Each object ID acts as a pointer to the _id field of some category document.

A RELATIONSHIP STRUCTURE

The next listing shows a sample category document. You can assign it to a new variable and insert it into the categories collection using db.categories.insert(newCategory). This will help you using it in forthcoming queries without having to type it again.

Listing 4.2 A category document

```
{
  _id:
  ObjectId("6a5b1476238d3b4dd50000
  48"), slug: "gardening-tools",
  name: "Gardening Tools",
  description: "Gardening gadgets galore!",
  parent_id:
  ObjectId("55804822812cb336b78728f9"),
    ancestors: [

        {
          name: "Home",
          _id:
          ObjectId("558048f0812cb336b78728f
          a"), slug: "home"
        },
```

```
        {
          name: "Outdoors",
          _id:
          ObjectId("55804822812cb336b78728
          f9"), slug: "outdoors"
        }
    ]
}
```

Nuts and bolts: On databases, collections, and documents

Databases

A database is a namespace and physical grouping of collections and their indexes. In this section, we'll discuss the details of creating and deleting databases. We'll also jump down a level to see how MongoDB allocates space for individual databases on the filesystem.

MANAGING DATABASES

There's no explicit way to create a database in MongoDB. Instead, a database is cre- ated automatically once you write to a collection in that database. Have a look at this Ruby code:

```
connection = Mongo::Client.new( [ '127.0.0.1:27017' ], :database =>
'garden' ) db = connection.database
```

Recall that the JavaScript shell performs this connection when you start it, and then allows you to select a database like this:

```
use garden
```

Assuming that the database doesn't exist already, the database has yet to be created on disk even after executing this code. All you've done is instantiate an instance of the class Mongo::DB, which represents a MongoDB database. Only when you write to a col- lection are the data files created. Continuing on in Ruby,

```
products = db['products']
products.insert_one({:name => "Extra Large Wheelbarrow"})
```

When you call insert_one on the products collection, the driver tells MongoDB to insert the product document into the garden.products collection. If that collec- tion doesn't exist, it's created; part of this involves allocating the garden database on disk.

You can delete all the data in this collection by calling:

```
products.find({}).delete_many
```

This removes all documents which match the filter {}, which is all documents in the collection. This command doesn't remove the collection itself; it only empties it. To remove a collection entirely, you use the drop method, like this:

```
products.drop
```

To delete a database, which means dropping all its collections, you issue a special com- mand. You can drop the garden database from Ruby like so:

```
db.drop
```

From the MongoDB shell, run the dropDatabase()method using JavaScript:

```
use garden
db.dropDatabas
e();
```

Be careful when dropping databases; there's no way to undo this operation since it erases the associated files from disk. Let's look in more detail at how databases store their data.

DATA FILES AND ALLOCATION

When you create a database, MongoDB allocates a set of data files on disk. All collec- tions, indexes, and other metadata for the database are stored in these files. The data files reside in whichever directory you designated as the dbpath when starting mongod. When left unspecified, mongod stores all its files in /data/db.[3] Let's see how this direc- tory looks after creating the garden database:

```
$ cd /data/db
$ ls -lah
drwxr-xr-x  81 pbakkum admin  2.7K 1 Jul 10:42 .
drwxr-xr-x   5 root    admin  170B 19 Sep 2012 ..
-rw-------   1 pbakkum admin   64M 1 Jul 10:43 garden.0
-rw-------   1 pbakkum admin  128M 1 Jul 10:42 garden.1
-rw-------   1 pbakkum admin   16M 1 Jul 10:43 garden.ns
-rwxr-xr-x   1 pbakkum admin    3B 1 Jul 08:31 mongod.lock
```

Collections

Collections are containers for structurally or conceptually similar documents. Here,

MANAGING COLLECTIONS

As you saw in the previous section, you create collections implicitly by inserting docu- ments into a particular namespace. But because more than one collection type exists, MongoDB also provides a command for creating collections. It provides this com- mand from the JavaScript shell:

```
db.createCollection("users")
```

When creating a standard collection, you have the option of preallocating a specific number of bytes. This usually isn't necessary but can be done like this in the Java- Script shell:

```
db.createCollection("users", {size: 20000})
```

Collection names may contain numbers, letters, or . characters, but must begin with a letter or number. Internally, a collection name is identified by its namespace name, which includes the name of the database it belongs to. Thus, the products collection is technically referred to as garden.productswhen referenced in a mes- sage to or from the core server. This fully qualified collection name can't be longer than 128 characters.

It's sometimes useful to include the . character in collection names to provide a kind of virtual namespacing. For instance, you can imagine a series of collections with titles like the following:

products.categor
ies
products.images
products.reviews

Keep in mind that this is only an organizational principle; the database treats collec- tions named with a .like any other collection.

Collections can also be renamed. As an example, you can rename the products col- lection with the shell's renameCollectionmethod:

```
db.products.renameCollection("store_products")
```

Listing 4.6 Simulating the logging of user actions to a capped collection

```
require 'mongo'


VIEW_PRODUCT = 0      # action type constants ADD_TO_CART = 1
CHECKOUT        = 2
PURCHASE        = 3



client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'garden')
```

```
    client[:user_actions].drop
actions = client[:user_actions, :capped => true, :size => 16384]
actions.create

500.times do |n|          # loop 500 times, using n as the iterator doc = {
    :username => "kbanker",
    :action_code => rand(4), # random value between 0 and 3, inclusive
    :time => Time.now.utc,
                                                          :n => n
                                                        }
  actions.insert_one(d
oc) end
```

First, you create a 16 KB capped collection called user_actionsusing client.[6] Next, you insert 500 sample log documents B. Each document contains a username, an action code (represented as a random integer from 0 through 3), and a timestamp. You've included an incrementing integer, n, so that you can identify which documents have aged out. Now you'll query the collection from the shell:

```
> use garden
> db.user_actions.coun
t(); 160
```

Even though you've inserted 500 documents, only 160 documents exist in the collec- tion.[7] If you query the collection, you'll see why:

```
db.user_actions.find().pretty();
{
    "_id" :
    ObjectId("51d1c69878b10e1a0e000040"
    ), "username" : "kbanker",
    "action_code" : 3,
    "time" : ISODate("2013-07-
    01T18:12:40.443Z"), "n" : 340
}
```

"_id" :
ObjectId("51d1c69878b10e1a0e000041"),
"username" : "kbanker",
"action_code" : 2,
"time" : ISODate("2013-07-
01T18:12:40.444Z"), "n" : 341

"_id" :
ObjectId("51d1c69878b10e1a0e000042"),
"username" : "kbanker",
"action_code" : 2,
"time" : ISODate("2013-07-
01T18:12:40.445Z"), "n" : 342

TTL COLLECTIONS

MongoDB also allows you to expire documents from a collection after a certain amount of time has passed. These are sometimes called time-to-live (TTL) collections, though this functionality is actually implemented using a special kind of index. Here's how you would create such a TTL index:

```
>
>
>
> db.reviews.createIndex({time_field: 1}, {expireAfterSeconds: 3600})
```

This command will create an index on time_field.

between time_field and the current time is greater than your expireAfterSeconds setting, then the document will be removed automatically. In this example, review documents will be deleted after an hour.

Using a TTL index in this way assumes that you store a timestamp in time_field.

Here's an example of how to do this:

```
> db.reviews.insert({
    time_field: new
    Date(),
    …
  })
```

SYSTEM COLLECTIONS

Part of MongoDB's design lies in its own internal use of collections. Two of these spe- cial system collections are system.namespaces and system.indexes. You can query the former to see all the namespaces defined for the current database:

```
> db.system.namespaces.find();
{ "name" : "garden.system.indexes" }
{ "name" : "garden.products.$_id_" }
{ "name" : "garden.products" }
{ "name" : "garden.user_actions.$_id_" }
{ "name" : "garden.user_actions", "options" : { "create" :
"user_actions", "capped" : true, "size" : 1024 } }
```

The first collection, system.indexes, stores each index definition for the current database. To see a list of indexes you've defined for the garden database, query the collection:

```
> db.system.indexes.find();
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "garden.products", "name" : "_id_" }
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "garden.user_actions", "name" :
"_id_" }
{ "v" : 1, "key" : { "time_field" : 1 }, "name" : "time_field_1", "ns" :
"garden.reviews", "expireAfterSeconds" : 3600 }
```

Documents and insertion

DOCUMENT SERIALIZATION, TYPES, AND LIMITS

All documents are serialized to BSON before being sent to MongoDB; they're later deserialized from BSON. The driver handles this process and translates it from and to the appropriate data types in its programming language. Most of the drivers provide a simple interface for serializing from and to BSON; this happens automatically when reading and writing documents. You don't need to worry about this normally, but we'll demonstrate it explicitly for educational purposes.

In the previous capped collections example, it was reasonable to assume that the sample document size was roughly 102 bytes. You can check this assumption by using the Ruby driver's BSON serializer:

```
doc = {
  :_id => BSON::ObjectId.new,
  :username => "kbanker",
  :action_code => rand(5),
  :time => Time.now.utc,
  :n => 1
}
bson = doc.to_bson
puts "Document #{doc.inspect} takes up #{bson.length} bytes as BSON"
```

Deserializing BSON is as straightforward with a little help from the StringIOclass.

Try running this Ruby code to verify that it works:

```
string_io = StringIO.new(bson)
deserialized_doc =
String.from_bson(string_io)
puts "Here's our document deserialized from
BSON:" puts deserialized_doc.inspect
```

STRINGS

All string values must be encoded as UTF-8. Though UTF-8 is quickly becoming the standard for character encoding, there are plenty of situations when an older encod- ing is still used. Users typically encounter issues with this when importing data gener- ated by legacy systems into MongoDB.

NUMBERS

BSON specifies three numeric types: double, int, and long. This means that BSON can encode any IEEE floating-point value and any signed integer up to 8 bytes in length. When serializing integers in dynamic languages, such as Ruby and Python, the driver will automatically determine whether to encode as an int or a long. In fact, there's only one common situation where a number's type must be made explicit: when you're inserting numeric data via the JavaScript shell. JavaScript, unhappily, natively

---

supports only a single numeric type called Number, which is equivalent to an IEEE 754 Double. Consequently, if you want to save a numeric value from the shell as an integer, you need to be explicit, using either NumberLong() or NumberInt(). Try this example:

```
db.numbers.save({n: 5});
db.numbers.save({n:
NumberLong(5)});
```

You've saved two documents to the numbers collection, and though their values are equal, the first is saved as a double and the second as a long integer. Querying for all documents where n is 5 will return both documents:

```
> db.numbers.find({n: 5});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

DATETIMES

The BSON datetime type is used to store temporal values. Time values are represented using a signed 64-bit integer marking milliseconds since the

Unix epoch. A negative value marks milliseconds prior to the epoch.[10]

VIRTUAL TYPES

What if you must store your times with their time zones? Sometimes the basic BSON types don't suffice. Though there's no way to create a custom BSON type, you can compose the various primitive BSON values to create your own virtual type in a sub- document. For instance, if you wanted to store times with zone, you might use a docu- ment structure like this, in Ruby:

```
{
  time_with_zone:
    { time: new
    Date(), zone:
    "EST"
  }
}
```

It's not difficult to write an application so that it transparently handles these compos- ite representations. This is usually how it's done in the real world. For example, Mongo-Mapper, an object mapper for MongoDB written in Ruby, allows you to define to_mongo and from_mongo methods for any object to accommodate these sorts of cus- tom composite types.

LIMITS ON DOCUMENTS

BSON documents in MongoDB v2.0 and later are limited to 16 MB in size. The limit exists for two related reasons. First, it's there to prevent developers from creating ungainly data models. Though poor data models are still possible with this limit, the 16 MB limit helps discourage schemas with oversized documents.

If you find yourself needing to store documents greater than 16 MB, consider whether your schema should split data into smaller documents, or whether a MongoDB document is even the right place to store such information—it may be better managed as a file.

The second reason for the 16 MB limit is performance-related. On the

server side, querying a large document requires that the document be copied into a buffer before being sent to the client. This copying can get expensive, especially (as is often the case) when the client doesn't need the entire document.[12] In addition, once sent, there's the work of transporting the document across the network and then deserializ- ing it on the driver side. This can become especially costly if large batches of multi-megabyte documents are being requested at once.

MongoDB documents are also limited to a maximum nesting depth of 100. Nesting occurs whenever you store a document within a document. Using deeply nested docu- ments—for example, if you wanted to serialize a tree data structure to a MongoDB

---

document—results in documents that are difficult to query and can cause problems during access. These types of data structures are usually accessed through recursive function calls, which can outgrow their stack for especially deeply nested documents.

BULK INSERTS

All of the drivers make it possible to insert multiple documents at once. This can be extremely handy if you're inserting lots of data, as in an initial bulk import or a migra- tion from another database system. Here's a simple Ruby example of this feature:

```
docs = [                          # define an array of documents
  { :username => 'kbanker' },
  { :username => 'pbakkum' },
  { :username => 'sverch' }
]
@col = @db['test_bulk_insert']
@ids = @col.insert_many(docs)   # pass the entire array to
insert puts "Here are the ids from the bulk insert:
#{@ids.inspect}"
```

Constructing Queries

This topic covers

- Querying an e-commerce data model
- The MongoDB query language in detail
- Query selectors and options

E-commerce queries

For instance, _idlookups shouldn't be a mystery at this point. But we'll also show you a few more sophisticated patterns, including querying for and displaying a category hierarchy, as well as providing filtered views of product listings.

Products, categories, and reviews

Most e-commerce applications provide at least two basic views of products and catego- ries. First is the product home page, which highlights a given product, displays reviews, and gives some sense of the product's categories. Second is the product listing page, which allows users to browse the category hierarchy and view thumbnails of all the products within a selected category. Let's begin with the product home page, in many ways the simpler of the two.

Imagine that your product page URLs are keyed on a product slug (you learned about these user-friendly permalinks in chapter 4). In that case, you can get all the data you need for your product page with the following three queries:

```
roduct = db.products.findOne({'slug': 'wheel-barrow-
9092'}) db.categories.findOne({'_id':
product['main_cat_id']}) db.reviews.find({'product_id':
product['_id']})
```

FINDONE VS. FIND QUERIES

The findOnemethod is similar to the following, though a cursor is returned even when you apply a limit:

```
db.products.find({'slug': 'wheel-barrow-9092'}).limit(1)
```

SKIP, LIMIT, AND SORT QUERY OPTIONS

Most applications paginate reviews, and for enabling this MongoDB provides skipand
limitoptions. You can use these options to paginate the review document like this:

```
db.reviews.find({'product_id': product['_id']}).skip(0).limit(12)


db.reviews.find({'product_id': product['_id']}).
                    sort({'helpful_votes': -1}).
                    limit(12)


page_number = 1
product = db.products.findOne({'slug': 'wheel-barrow-
9092'}) category = db.categories.findOne({'_id':
product['main_cat_id']}) reviews_count =
db.reviews.count({'product_id': product['_id']}) reviews =
db.reviews.find({'product_id': product['_id']}).
                            skip((page_number - 1) *
                            12). limit(12).
                            sort({'helpful_votes': -1})
```

MongoDB's query language

Query criteria and selectors

Query criteria allow you to use one or more query selectors to specify the query's results. MongoDB gives you many possible selectors. This section provides an overview.

SELECTOR MATCHING

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

```
db.users.find({'last_name': "Banker"})
db.users.find({'first_name': "Smith", birth_year:
1975})
```

RANGES

Table 5.1 shows the range query operators most commonly used in MongoDB.

Table 5.1 Summary of range query operators

Operator	Description
$lt	Less than
$gt	Greater than
$lte	Less than or equal
$gte	Greater than or equal

Beginners sometimes struggle with combining these operators. A common mistake is to repeat the search key:

```
db.users.find({'birth_year': {'$gte': 1985}, 'birth_year': {'$lte': 2015}})
```

The aforementioned query only takes into account the last condition. You can prop- erly express this query as follows:

```
db.users.find({'birth_year': {'$gte': 1985, '$lte': 2015}})
```

SET OPERATORS

Three query operators—$in, $all, and $nin—take a list of one or more values as their predicate, so these are called set operators. $in returns a document if any of the given values matches the search key.

Table 5.2 Summary of set operators

Operator	Description
$in	Matches if any of the arguments are in the referenced set
$all	Matches if all of the arguments are in the referenced set and is used in documents that contain arrays
$nin	Matches if none of the arguments are in the referenced set

If the following list of category IDs

```
[
  ObjectId("6a5b1476238d3b4dd5
  000048"),
  ObjectId("6a5b1476238d3b4dd5
  000051"),
  ObjectId("6a5b1476238d3b4dd5
  000057")
]
```

corresponds to the lawnmowers, hand tools, and work clothing categories, the query to find all products belonging to these categories looks like this:

```
db.products.find({
    'main_cat_id':
    {
      '$in': [
        ObjectId("6a5b1476238d3b4dd5
        000048"),
        ObjectId("6a5b1476238d3b4dd5
        000051"),
```

```
            ObjectId("6a5b1476238d3b4dd5
            000057")
        ]
    }
})
```

Table 5.3 Summary of Boolean operators

Operator	Description
$ne	Matches if the argument is not equal to the element
$not	Inverts the result of a match
$or	Matches if any of the supplied set of query terms is true
$nor	Matches if none of the supplied set of query terms are true
$and	Matches if all of the supplied set of query terms are true
$exists	Matches if the element exists in the document.

QUERYING FOR A DOCUMENT WITH A SPECIFIC KEY

The final operator we'll discuss in this section is $exists. This operator is necessary because collections don't enforce a fixed schema, so you occasionally need a way to query for documents containing a particular key. Recall that you'd planned to use

each product's detailsattribute to store custom fields. You might, for instance, store a color field inside the details attribute. But if only a subset of all products specify a set of colors, then you can query for the ones that don't like this:

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

The opposite query is also possible:

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

ARRAYS

Arrays give the document model much of its power. As you've seen in the e-commerce example, arrays are used to store lists of strings, object IDs, and even other docu- ments.

Arrays afford rich yet comprehensible documents; it stands to reason that MongoDB would let you query and index the array type with ease. And it's true: the simplest array queries look like queries on any other document type, as you can see in table 5.4.

Table 5.4 Summary of array operators

Operator	Description
$elemMatch	Matches if all supplied terms are in the same subdocument
$size	Matches if the size of the array subdocument is the same as the supplied literal value

Let's look at these arrays in action. Take product tags again. These tags are repre- sented 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 and uses the same syntax as query- ing a single document value:

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

Importantly, this query can take advantage of an index on the tags field. If you build the required index and run your query with explain(), you'll see that a B-tree cursor[3] is used:

```
db.products.ensureIndex({tags: 1})
db.products.find({tags:
"soil"}).explain()
```

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

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

REGULAR EXPRESSIONS

The $regexoperator is summarized here:

- $regex Match the element against the supplied regex term

MongoDB is a case-sensitive system, and when using a regex, unless you use the /i modifier (that is, /best|worst/i), the search will have to exactly match the case of the fields being searched. But one caveat is that if you do use /i, it will disable the use of indexes. If you want to do indexed case-insensitive search of the contents of string fields in documents, consider either storing a duplicate field with the contents forced to lowercase specifically for searching or using MongoDB's text search capabili- ties, which can be combined with other queries and does provide an indexed case- insensitive search.

MISCELLANEOUS QUERY OPERATORS

Two more query operators aren't easily categorized and thus deserve their own sec- tion. The first is $mod, which allows you to query documents matching a given modulo operation, and the second is $type, which matches values by their BSON type. Both are detailed in table 5.5.

Table 5.5 Summary of miscellaneous operators

Operator	Description
$mod [(quotient),(result)]	Matches if the element matches the result when divided by the quotient
$type	Matches if the element type matches a specified BSON type
$text	Allows you to performs a text search on the content of the fields indexed with a text index

For instance, $modallows you to find all order subtotals that are evenly divisible by 3 using the followingquery:

```
db.orders.find({subtotal: {$mod: [3, 0]}})
```

You can see that the $modoperator takes an array having two values. The first is the divisor and the second is the expected remainder. This query technically reads, "Find all documents with subtotals that return a remainder of 0 when divided by 3." This is a contrived example, but it demonstrates the idea. If you end up using the $mod opera- tor, keep in mind that it won't use an index.

The second miscellaneous operator, $type, matches values by their 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 lets you test against type.

Table 5.6 BSON types

BSON type	$type number	Example
Double	1	123.456
String (UTF-8)	2	“Now is the time”
Object	3	{ name:"Tim",age:"myob" }
Array	4	[123,2345,"string"]
Binary	5	BinData(2,"DgAAAEltIHNvbWUgYmluYXJ5")
ObjectId	7	ObjectId("4e1bdda65025ea6601560b50") true
Boolean	8	ISODate("2011-02-24T21:26:00Z")
Date	9	null
Null	10	/test/i
Regex	11	function() {return false;}
JavaScript	13	Not used; deprecated in the standard function
Symbol	14	(){return false;}
Scoped JavaScript	15	10
32-bit integer	16	{ "t" : 1371429067, "i" : 0 }
Timestamp	17	NumberLong(10)
64-bit integer	18	
Maxkey	127	
Minkey	255	
Maxkey	128	

		{"$maxKey": 1} { "$minKey" : 1} {"maxkey" : { "$maxKey" : 1 }}

PROJECTIONS

- Projections are most commonly defined as a set of

fields to return:

```
db.users.find({}, {'username': 1})
```

SORTING

```
db.reviews.find({}).sort({'rating': -1})
```

Naturally, it might be more useful to sort by helpfulness and then by rating:

```
db.reviews.find({}).sort({'helpful_votes':-1, 'rating': -1})
```

SKIP AND LIMIT

```
db.docs.find({}).skip(500000).limit(10).sort({date: -1})
```

becomes this:

```
previous_page_date = new Date(2013, 05, 05)
db.docs.find({'date': {'$gt': previous_page_date}}).limit(10).sort({'date': -1})
```

This topic covers

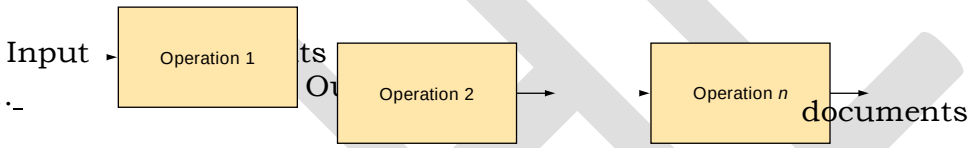
- Aggregation on the e-commerce data model
- Aggregation framework details
- Performance and limitations
- Other aggregation capabilities

Aggregation framework overview

A call to the aggregation framework defines a pipeline (figure 6.1), the *aggregation pipeline*, where the output from each step in the pipeline provides input to the next step. Each step executes a single operation on the input documents to transform the input and generate output documents.

Aggregation pipeline operations include the following:

- $project—Specify fields to be placed in the output document (projected).
- $match—Select documents to be processed, similar to find().



- $limit—Limit the number of documents to be passed to the next step.
- $skip—Skip a specified number of documents.
- $unwind—Expand an array, generating one output document for each array entry.
- $group—Group documents by a specified key.
- $sort—Sort documents.
- $geoNear—Select documents near a geospatial location.

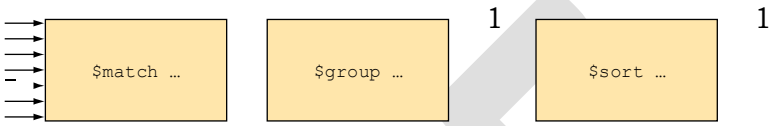
- $out—Write the results of the pipeline to a collection (new in v2.6).
- $redact—Control access to certain data (new in v2.6).

Most of these operators will look familiar if you've read the previous chapter on con- structing MongoDB queries. Because most of the aggregation framework operators work similarly to a function used for MongoDB queries, you should make sure you have a good understanding of section 5.2 on the MongoDB query language before continuing.

This code example defines an aggregation framework pipeline that consists of a match, a group, and then a sort:

db.products.aggregate([{$match: …}, {$group: …}, {$sort: …}])

This series of operations is illustrated in figure 6.2.



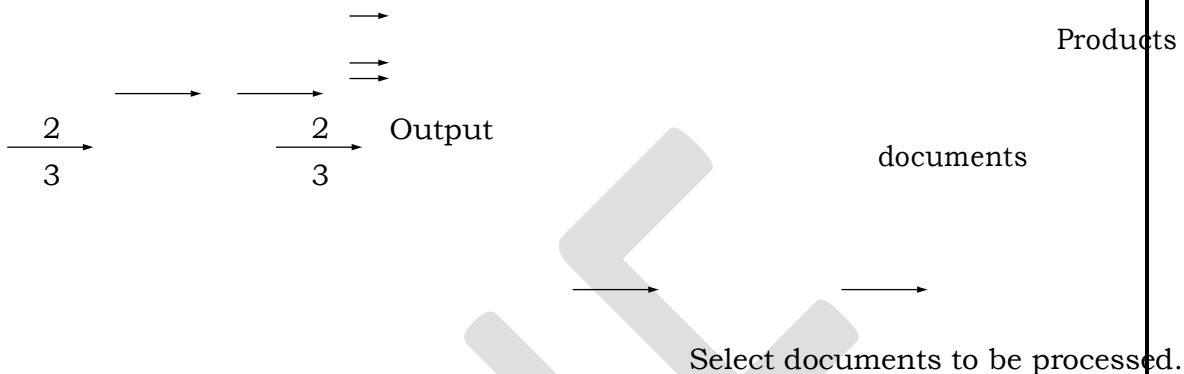


Table 6.1 SQL versus aggregation framework comparison

SQL command	Aggregation framework operator
SELECT	$project
	$group functions: $sum, $min, $avg, etc.
FROM	db.collectionName.aggregate(...)
JOIN	$unwind
WHER	$match
E	$group
GROU	$match
PBY	
HAVING	

Products, categories, and reviews

Now let's look at a simple example of how the aggregation framework can be used to summarize information about a product. Chapter 5 showed an example of counting the number of reviews for a given product using this query:

product = db.products.findOne({'slug': 'wheelbarrow-

```
9092'}) reviews_count = db.reviews.count({'product_id':
product['_id']})
```

Let's see how to do this using the aggregation framework. First, we'll look at a query that will calculate the total number of reviews for all products:

```
db.reviews.aggregate([
 {$group : {
 _id:'$product_id',
                count:{$sum:1}
                }}
]);
```

Group the input documents by product_id.

Count the number of reviews for each product.

This single operator pipeline returns one document for each product in your data- base that has a review, as illustrated here:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"), "count" : 2 }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```

Outputs one document for each product

Next, add one more operator to your pipeline so that you select only the one prod- uct you want to get a count for:

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})
```

```
ratingSummary = db.reviews.aggregate([
 {$match : { product_id: product['_id']} },
 {$group : { _id:'$product_id',
```

Select only a single product.

```
]).next();
Count:{$sum:1} }}
```

Return the first document in the results.

is example returns the one product you're interested in and assigns it to the vari- able ratingSummary. Note that the result from the aggregation pipeline is a *cursor*, a pointer to your results that allows you to process results of almost any size, one docu- ment at a time. To retrieve the single document in the result, you use the next() func- tion to return the first document from the cursor:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```

The parameters passed to the $matchoperator, {'product_id':product['_id']}, should look familiar. They're the same as those used for the query taken from chap- ter 5 to calculate the count of reviews for a product:

```
db.reviews.count({'product_id': product['_id']})
```

CALCULATING THE AVERAGE REVIEW

To calculate the average review for a product, you use the same pipeline as in the pre- vious example and add one more field:

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})

                    ratingSummary = db.reviews.aggregate([
    {$match : {'product_id': product['_id']}},
    {$group : { _id:'$product_id', average:{$avg:'$rating'}, count:
        {$sum:1}}}
]).next();
```

Calculate the average rating for a product.

The previous example returns a single document and assigns it to the variable rating- Summarywith the content shown here:

```
{
    "_id" :
    ObjectId("4c4b1476238d3b4dd5003981
    "), "average" : 4.333333333333333,
    "count" : 3
}
```

This example uses the $avgfunction to calculate the average rating for the product. Notice also that the field being averaged, rating, is specified using '$rating'in the
$avgfunction. This is the same convention used for specifying the field for the $group _idvalue, where you used this:

```
_id:'$product_id'.
```

```
                count:{$sum:1}}}
                        ]).toArray();
```

As shown in this snippet, you've once again produced a count using the $sum func- tion; this time you counted the number of reviews for each rating. Also note that the result of this aggregation call is a cursor that you've converted to an array and assigned to the variable countsByRating.

SQL query

For those familiar with SQL, the equivalent SQL query would look something like this:

```
SELECT RATING, COUNT(*) AS COUNT
  FROM REVIEWS
 WHERE PRODUCT_ID = '4c4b1476238d3b4dd5003981'
 GROUP BY RATING
```

This aggregation call would produce an array similar to this:

```
[ { "_id" : 5, "count" : 5 },
  { "_id" : 4, "count" : 2 },
  { "_id" : 3, "count" : 1 } ]
```

JOINING COLLECTIONS

Next, suppose you want to examine the contents of your database and count the num- ber of products for each main category. Recall that a product has only one main cate- gory. The aggregation command looks like this:

```
db.products.aggregate([
    {$group : { _id:'$main_cat_id',
                count:{$sum:1}}}
]);
```

This command would produce a list of output documents. Here's an example:

```
{ "_id" : ObjectId("6a5b1476238d3b4dd5000048"), "count" : 2 }
```

option is to use the forEachfunction to process the cursor returned from the aggre- gation command and add the name using a pseudo-join. Here's an example:

```
b.mainCategorySummary.remove({});

db.products.aggregate([
    {$group : { _id:'$main_cat_id',
                count:{$sum:1}}}
]).forEach(function(doc){
```

Remove existing documents from mainCategorySummary collection

Read category for a result

```
    var category = db.categories.findOne({_id:doc._id});
    if (category !== null) {

        doc.category_name = category.name;
                        }
                        else {
        doc.category_name = 'not found';
                        }
    db.mainCategorySummary.insert(doc);
})
```

mainCategorySummary:

```
db.products.aggregate([
    {$group : { _id:'$main_cat_id',
                count:{$sum:1}}},
    {$out : 'mainCategorySummary'}
```

])

User and order

When the first edition of this book was written, the aggregation framework, first intro- duced in MongoDB v2.2, hadn't yet been released. The first edition used the MongoDB map-reducefunction in two examples, grouping reviews by users and summarizing sales by month. The example grouping reviews by user showed how many reviews each reviewer had and how many helpful votes each reviewer had on average. Here's what this looks like in the aggregation framework, which provides a much simpler and more intuitive approach:

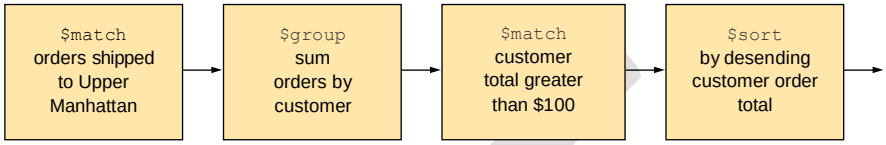
```
db.reviews.aggregate([
    {$group :
        {_id : '$user_id',
         count : {$sum : 1},
         avg_helpful : {$avg : '$helpful_votes'}}
    }
])
```

The result from this call looks like this:

```
{ "_id" :
    ObjectId("4c4b1476238d3b4dd50000
    03"), "count" : 1, "avg_helpful" : 10 }
{ "_id" :
    ObjectId("4c4b1476238d3b4dd50000
    02"), "count" : 2, "avg_helpful" : 4 }
{ "_id" :
    ObjectId("4c4b1476238d3b4dd50000
    01"), "count" : 2, "avg_helpful" : 5 }
```

FINDING BEST MANHATTAN CUSTOMERS

Now let's extend that query to find the highest spenders in Upper Manhattan. This pipeline is summarized in figure 6.5. Notice that the $match is the first step in the pipeline, greatly reducing the number of documents your pipeline has to process.



The query includes these steps:

- $match—Find orders shipped to Upper Manhattan.
- $group—Sum the order amounts for each customer.
- $match—Select those customers with order totals greater than $100.
- $sort—Sort the result by descending customer order total.

UNIT III
SYLLABUS

Updates, atomic operations, and deletes: A brief tour of document updates – E-commerce updates – Atomic document processing – MongoDB updates and deletes. Indexing and query optimization: Indexing theory – Indexing in practice – Query optimization.

This Topic covers

- Updating documents
- Processing documents atomically
- Applying complex updates to a real-world example
- Using update operators
- Deleting documents

To update is to write to existing documents. Doing this effectively requires a thorough understanding of the kinds of document structures available and of the query expressions made possible by MongoDB. Having studied the e-commerce data model in the last two chapters, you should have a good sense of the ways in which schemas are designed and queried. We'll use all of this knowledge in our study of updates.

Brief tour of document updates

If you need to update a document in MongoDB, you have two ways of going about it. You can either replace the document altogether, or you can use update operators to modify specific fields within the document. As a way of setting the stage for the more detailed examples to come, we'll begin this chapter with a simple demonstration of these two techniques. We'll then provide reasons for preferring one over the other.

To start, recall the sample user document we developed in chapter 4. The document includes a user's first and last names, email address, and shipping addresses. Here's a simplified example:

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

```
  email: "kylebanker@gmail.com",
  first_name: "Kyle",
  last_name: "Banker",
  hashed_password: "bd1cfa194c3a603e7186780824b04419",
  addresses: [
    {
      name: "work",
      street: "1 E. 23rd Street",
      city: "New York",
      state: "NY",
      zip: 10010
    }
  ]
}
```

Modify by replacement

To replace the document altogether, you first query for the document, modify it on the client side, and then issue the update with the modified document. Here's how that looks in the JavaScript shell:

```
user_id = ObjectId("4c4b1476238d3b4dd5003981")
doc = db.users.findOne({_id: user_id})
doc['email'] = 'mongodb-user@mongodb.com'
print('updating ' + user_id)
db.users.update({_id: user_id}, doc)
```

With the user's `_id` at hand, you first query for the document. Next you modify the document locally, in this case changing the `email` attribute. Then you pass the modified document to the `update` method. The final line says, "Find the document in the `users` collection with the given `_id`, and replace that document with the one we've provided." The thing to remember is that the update operation replaces the entire document, which is why it must be fetched first. If multiple users update the same document, the last write will be the one that will be stored.

Modify by operator

That's how you modify by replacement; now let's look at modification by operator:

```
user_id = ObjectId("4c4b1476238d3b4dd5000001")
db.users.update({_id: user_id},
  {$set: {email: 'mongodb-user2@mongodb.com'}})
```

The example uses `$set`, one of several special update operators, to modify the email address in a single request to the server. In this case, the update request is much more targeted: find the given user document and set its `email` field to `mongodb-user2@mongodb.com`.

Both methods compared

How about another example? This time you want to increment the number of reviews on a product. Here's how you'd do that as a document replacement:

```
product_id = ObjectId("4c4b1476238d3b4dd5003982")
doc = db.products.findOne({_id: product_id})
doc['total_reviews'] += 1       // add 1 to the value in total_reviews
db.products.update({_id: product_id}, doc)
```

And here's the targeted approach:

```
db.products.update({_id: product_id}, {$inc: {total_reviews: 1}})
```

The replacement approach, as before, fetches the user document from the server, modifies it, and then resends it. The update statement here is similar to the one you used to update the email address. By contrast, the targeted update uses a different update operator, `$inc`, to increment the value in `total_reviews`.

Deciding: replacement vs. operators

Modification by replacement is the more generic approach. Imagine that your application presents an HTML form for modifying user information. With document replacement, data from the form post, once validated, can be passed right to MongoDB; the code to perform the update is the same regardless of which user attributes are modified. For instance, if you were going to build a MongoDB object mapper that needed to generalize updates, then updates by replacement would probably make for a sensible default.[1] of concurrent updates, each `$inc` will be applied in isola- tion, all or nothing.[2]

AVERAGE PRODUCT RATINGS

Products are amenable to numerous update strategies. Assuming that administrators are provided with an interface for editing product information, the easiest update involves fetching the current product document, merging that data with the user's edits, and issuing a document replacement. At other times, you may only need to

update a couple of values, where a targeted update is clearly the way to go. This is the case with average product ratings. Because users need to sort product listings based on average product rating, you store that rating in the product document itself and update the value whenever a review is added or removed.

Here's one way of issuing this update in JavaScript:

```
product_id = ObjectId("4c4b1476238d3b4dd5003981")
count = 0
total = 0
db.reviews.find({product_id: product_id}, {rating: 4}).forEach(
  function(review) {
    total += review.rating
    count++
  })
average = total / count
db.products.update({_id: product_id},
  {$set: {total_reviews: count, average_review: average}})
```

This code aggregates and produces the `rating` field from each product review and then produces an average. You also use the fact that you're iterating over each rating to count the total ratings for the product. This saves an extra database call to the `count` function. With the total number of reviews and their average rating, the code issues a targeted update, using `$set`.

If you don't want to hardcode an `ObjectId`, you can find a specific `ObjectId` as follows and use it afterwards:

```
product_id = db.products.findOne({sku: '9092'}, {'_id': 1})
```

Performance-conscious users may balk at the idea of re-aggregating all product reviews for each update. Much of this depends on the ratio of reads to writes; it's likely that more users will see product reviews than write their own, so it makes sense to re-aggregate on a write. The method provided here, though conservative, will likely be acceptable for most situations, but other strategies are possible. For instance, you could store an extra field on the product document that caches the review ratings total, making it possible to compute the average incrementally. After inserting a new review, you'd first query for the product to get the current total number of reviews and the ratings total. Then you'd calculate the average and issue an update using a selector like the following:

```
db.products.update({_id: product_id},
  {
    $set: {
      average_review: average,
```

```
        ratings_total: total
      },
      $inc: {
        total_reviews: 1
      }
    })
```

With many databases, there's no easy way to represent a category hierarchy. This is true of MongoDB, although the document structure does help the situation somewhat. Documents encourage a strategy that optimizes for reads because each category can contain a list of its denormalized ancestors. The one tricky requirement is keeping all the ancestor lists up to date. Let's look at an example to see how this is done.

First you need a generic method for updating the ancestor list for any given category. Here's one possible solution:

```
var generate_ancestors = function(_id, parent_id) {
  ancestor_list = []
  var cursor = db.categories.find({_id: parent_id})
  while(cursor.size() > 0) {
    parent = cursor.next()
    ancestor_list.push(parent)
    parent_id = parent.parent_id
    cursor = db.categories.find({_id: parent_id})
  }
  db.categories.update({_id: _id}, {$set: {ancestors: ancestor_list}})
}
```

This method works by walking backward up the category hierarchy, making successive queries to each node's `parent_id` attribute until reaching the root node (where `parent_id` is `null`). All the while, it builds an in-order list of ancestors, storing that result in the `ancestor_list` array. Finally, it updates the category's `ancestors` attribute using `$set`.

Now that you have that basic building block, let's look at the process of inserting a new category. Imagine you have a simple category hierarchy that looks like the one in figure 7.1.

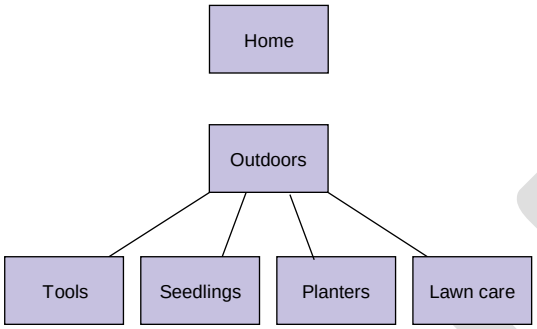


Figure 7.1 An initial category hierarchy

Suppose you want to add a new category called Gardening and place it under the Home category. You insert the new category document and then run your method to generate its ancestors:

```
parent_id = ObjectId("8b87fb1476238d3b4dd50003")
category = {
  parent_id: parent_id,
  slug: "gardening",
  name: "Gardening",
  description: "All gardening implements, tools, seeds, and soil."
}
db.categories.save(category)
generate_ancestors(category._id, parent_id)
```

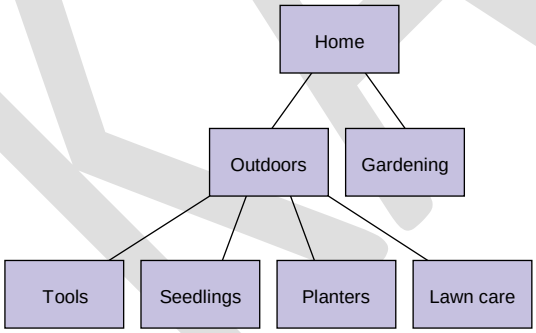


Figure 7.2 Adding a Gardening category

That's easy enough. But what if you now want to place the Outdoors category underneath Gardening? This is potentially complicated because it alters the ancestor lists of a number of categories. You can start by changing the `parent_id` of Outdoors

to the `_id` of Gardening. This turns out to be not too difficult provided that you already have both an `outdoors_id` and a `gardening_id` available:

```
db.categories.update({_id: outdoors_id}, {$set: {parent_id: gardening_id}})
```

Because you've effectively moved the Outdoors category, all the descendants of Outdoors are going to have invalid ancestor lists. You can rectify this by querying for all categories with Outdoors in their ancestor lists and then regenerating those lists. MongoDB's power to query into arrays makes this trivial:

```
db.categories.find({'ancestors.id': outdoors_id}).forEach(
  function(category) {
    generate_ancestors(category._id, outdoors_id)
  })
```

That's how you handle an update to a category's `parent_id` attribute, and you can see the resulting category arrangement in figure 7.3.

But what if you update a category name? If you change the name of Outdoors to The Great Outdoors, you also have to change Outdoors wherever it appears in the ancestor lists of other categories. You may be justified in thinking, "See? This is where denormalization comes to bite you," but it should make you feel better to know that you can perform this update without recalculating any ancestor list. Here's how:

```
doc = db.categories.findOne({_id: outdoors_id})
doc.name = "The Great Outdoors"
db.categories.update({_id: outdoors_id}, doc)
db.categories.update(
  {'ancestors._id': outdoors_id},
  {$set: {'ancestors.$': doc}},
  {multi: true})
```

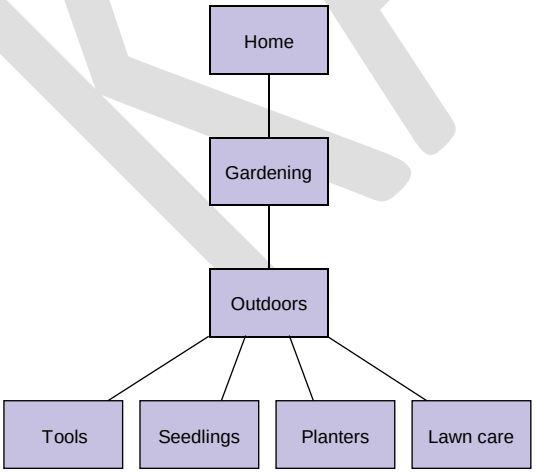


Figure 7.3 The category

tree in its final state

You first grab the Outdoors document, alter the `name` attribute locally, and then update via replacement. Now you use the updated Outdoors document to replace its occurrences in the various ancestor lists. The `multi` parameter `{multi: true}` is easy to understand; it enables multi-updates causing the update to affect all documents matching the selector—without `{multi: true}` an update will only affect the first matching document. Here, you want to update each category that has the Outdoors category in its ancestor list.

The positional operator is more subtle. Consider that you have no way of knowing where in a given category's ancestor list the Outdoors category will appear. You need a way for the update operator to dynamically target the position of the Outdoors category in the array for any document. Enter the positional operator. This operator (here the `$` in `ancestors.$`) substitutes the array index matched by the query selector with itself, and thus enables the update.

Here's another example of this technique. Say you want to change a field of a user address (the example document shown in section 7.1) that has been labeled as "work." You can accomplish this with a query like the following:

```
db.users.update({
     _id: ObjectId("4c4b1476238d3b4dd5000001"),
     'addresses.name': 'work'},
     {$set: {'addresses.$.street': '155 E 31st St.'}})
```

Because of the need to update individual subdocuments within arrays, you'll always want to keep the positional operator at hand. In general, these techniques for updating the category hierarchy will be applicable whenever you're dealing with arrays of subdocuments.

Reviews

Not all reviews are created equal, which is why this application allows users to vote on them. These votes are elementary; they indicate that the given review is helpful. You've modeled reviews so that they cache the total number of helpful votes and keep a list of each voter's ID. The relevant section of each review document looks like this:

```
{
  helpful_votes: 3,
  voter_ids: [
```

```
    ObjectId("4c4b1476238d3b4dd5000041"),
    ObjectId("7a4f0376238d3b4dd5000003"),
    ObjectId("92c21476238d3b4dd5000032")
  ]
}
```

You can record user votes using targeted updates. The strategy is to use the `$push` operator to add the voter's ID to the list and the `$inc` operator to increment the total number of votes, both in the same JavaScript console update operation:

```
db.reviews.update({_id: ObjectId("4c4b1476238d3b4dd5000041")}, {
    $push: {
      voter_ids: ObjectId("4c4b1476238d3b4dd5000001")
    },
    $inc: {
      helpful_votes: 1
    }
  })
```

This is almost correct. But you need to ensure that the update happens only if the voting user hasn't yet voted on this review, so you modify the query selector to match only when the `voter_ids` array doesn't contain the ID you're about to add. You can easily accomplish this using the `$ne` query operator:

```
query_selector = {
  _id: ObjectId("4c4b1476238d3b4dd5000041"),


  voter_ids: {
    $ne: ObjectId("4c4b1476238d3b4dd5000001")
  }
}
db.reviews.update(query_selector, {
    $push: {
      voter_ids: ObjectId("4c4b1476238d3b4dd5000001")
    },
    $inc : {
      helpful_votes: 1
    }
  })
```

This is an especially powerful demonstration of MongoDB's update mechanism and how it can be used with a document-oriented schema.

atomic and efficient. The update is atomic because selection and modification occur in the same query. The atomicity ensures that, even in a high-concurrency environment, it will be impossible for any one user to vote more than once. The efficiency lies

in the fact that the test for voter membership and the updates to the counter and the voter list all occur in the same request to the server.

.

Orders

The atomicity and efficiency of updates that you saw in reviews can also be applied to orders. Specifically, you're going to see the MongoDB calls needed to implement an `add_to_cart` function using a targeted update. This is a three-step process. First, you construct the product document that you'll store in the order's line-item array. Then you issue a targeted update, indicating that this is to be an *upsert*—an update that will insert a new document if the document to be updated doesn't exist.

doesn't yet exist, seamlessly handling both initial and subsequent additions to the shopping cart.[3]

Let's begin by constructing a sample document to add to the cart:

```
cart_item = {
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku:  "9092",
  name: "Extra Large Wheel Barrow",
  pricing: {
    retail: 5897,
    sale:   4897
  }
}
```

You'll most likely build this document by querying the `products` collection and then extracting whichever fields need to be preserved as a line item. The product's `_id`, `sku`, `slug`, `name`, and `price` fields should suffice. Next you'll ensure that there's an order for the customer with a status of `'CART'` using the parameter `{upsert: true}`. This operation will also increment the order `sub_total` using the `$inc` operator:

```
selector = {
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART'
}
update = {
  $inc: {
    sub_total: cart_item['pricing']['sale']
  }
}
db.orders.update(selector, update, {upsert: true})
```

INITIAL UPSERT TO CREATE ORDER DOCUMENT

To make the code clearer, you're constructing the query selector and the update document separately. The update document increments the order subtotal by the sale price of the cart item. Of course, the first time a user executes the `add_to_cart` function, no shopping cart will exist. That's why you use an upsert here. The upsert will construct the document implied by the query selector including the update. Therefore, the initial upsert will produce an order document like this:

```
{
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART',
  subtotal: 9794
}
```

3 You then perform an update of the order document to add the line item if it's not already on the order:

```
selector = {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
    state: 'CART',
    'line_items._id':
        {'$ne': cart_item._id}
    }

update = {'$push': {'line_items': cart_item}}
db.orders.update(selector, update)
```

ANOTHER UPDATE FOR QUANTITIES

Next you'll issue another targeted update to ensure that the item quantities are correct. You need this update to handle the case where the user clicks Add to Cart on an item that's already in the cart. In this case the previous update won't add a new item to the cart, but you'll still need to adjust the quantity:

```
selector = {
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART',
  'line_items._id': ObjectId("4c4b1476238d3b4dd5003981")
}


update = {
  $inc: {
    'line_items.$.quantity': 1
  }
}
db.orders.update(selector, update)
```

We use the `$inc` operator to update the quantity on the individual line item. The update is facilitated by the positional operator, `$`, introduced previously. Thus, after

the user clicks Add to Cart twice on the wheelbarrow product, the cart should look like this:

```
{
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART',
  line_items: [
    {
      _id: ObjectId("4c4b1476238d3b4dd5003981"),
      quantity:  2,
      slug: "wheel-barrow-9092",
      sku:  "9092",
      name: "Extra Large Wheel Barrow",
      pricing: {
        retail: 5897,
        sale:   4897
      }
    }
  ],
  subtotal: 9794
}
```

Atomic document processing

One tool you won't want to do without is MongoDB's `findAndModify` command.[4] This command allows you to atomically update a document and return it in the same round-trip. An atomic update is one where no other operation can interrupt or interleave itself with the update. What if another user tries to change the document after you find it but before you modify it? The find might no longer apply. An atomic update prevents this case; all other operations must wait for the atomic update to finish.

Every update in MongoDB is atomic, but the difference with `findAndModify` is that it also atomically returns the document to you. Why is this useful? If you fetch and then update a document (or update then fetch it), there can be changes made to the document by another MongoDB user in between those operations. Thus it's impossible to know the true state of the document you updated, before or after the update, even though the update is atomic, unless you use `findAndModify`. The other option is to use the optimistic locking mentioned in section 7.1, but that would require additional application logic to implement.

This atomic update capability is a big deal because of what it enables. For instance, you can use `findAndModify` to build job queues and state machines. You can then use these primitive constructs to implement basic transactional semantics, which greatly expand the range of applications you can build using MongoDB. With these transac-

tion-like features, you can construct an entire e-commerce site on MongoDB—not just the product content, but the checkout mechanism and the inventory management as well.

To demonstrate, we'll look at two examples of the `findAndModify` command in action. First, we'll show how to handle basic state transitions on the shopping cart. Then we'll look at a slightly more involved example of managing a limited inventory.

ring a valid initial state, and an update that effects the change of state. Let's skip forward a few steps in the order process and assume that the user is about to click the Pay Now button to authorize the purchase. If you're going to authorize the user's credit card synchronously on the application side, you need to ensure these four things:

1. You authorize for the amount that the user sees on the checkout screen.
2. The cart's contents never change while in the process of authorization.
3. Errors in the authorization process return the cart to its previous state.
4. If the credit card is successfully authorized, the payment information is posted to the order, and that order's state is transitioned to PRE-SHIPPING.

The state transitions that you'll use are shown in figure 7.5.

PREPARE THE ORDER FOR CHECKOUT

The first step is to get the order into the new PRE-AUTHORIZE state. You use `findAndModify` to find the user's current order object and ensure that the object is in a CART state:

```
newDoc = db.orders.findAndModify({
    query: {
      user_id: ObjectId("4c4b1476238d3b4dd5000001"),
      state: 'CART'
    },
    update: {
      $set: {
        state: 'PRE-AUTHORIZE'
      }
    },
    'new': true
  })
```

contents. This is because all updates to the cart always ensure a state of CART. `findAndModify` is useful here because you want to know the state of the document exactly when you changed its state to PRE-AUTHORIZE. What would happen to the total calculations if another thread was also attempting to move the user through the check-

out process?

VERIFY THE ORDER AND AUTHORIZE

Now, in the preauthorization state, you take the returned order object and recalculate the various totals. Once you have those totals, you issue a new `findAndModify` that only transitions the document's state to AUTHORIZING if the new totals match the old totals. Here's what that `findAndModify` looks like:

```
oldDoc = db.orders.findAndModify({
    query: {
      user_id: ObjectId("4c4b1476238d3b4dd5000001"),
      total: 99000,
      state: "PRE-AUTHORIZE"
    },
    update: {
      '$set': {
        state: "AUTHORIZING"
      }
    }
  })
```

If this second `findAndModify` fails, then you must return the order's state to CART and report the updated totals to the user. But if it succeeds, you know that the total to be authorized is the same total that was presented to the user. This means you can move on to the actual authorization API call. Thus, the application now issues a credit card authorization request on the user's credit card. If the credit card fails to authorize, you record the failure and, as before, return the order to its CART state.

FINISHING THE ORDER

If the authorization is successful, you write the authorization information to the order and transition it to the next state. The following strategy does both in the same `findAndModify` call. Here, the example uses a sample document representing the authorization receipt, which is attached to the original order:

```
auth_doc = {
  ts: new Date(),
  cc: 3432003948293040,
  id: 29238382910293844839493 48,
  gateway: "Authorize.net"
}
db.orders.findAndModify({
    query: {
      user_id: ObjectId("4c4b1476238d3b4dd5000001"),
      state: "AUTHORIZING"
    },
```

```
    update: {
      $set: {
        state: "PRE-SHIPPING",
        authorization: auth_doc
      }
    }
  })
```

Inventory management

Not every e-commerce site needs strict inventory management. Most commodity items can be replenished in enough time to allow any order to go through regardless of the actual number of items on hand. In cases like these, managing inventory is easily handled by managing expectations; as soon as only a few items remain in stock, adjust the shipping estimates.

One-of-a-kind items present a different challenge. Imagine you're selling concert tickets with assigned seats or handmade works of art. These products can't be hedged; users will always need a guarantee that they can purchase the products they've selected. Here we'll present a possible solution to this problem using MongoDB. This will further illustrate the creative possibilities in the `findAndModify` command and the judicious use of the document model. It will also show how to implement transactional semantics across multiple documents. Although you'll only see a few of the key MongoDB calls used by this process, the full source code for the `InventoryFetcher` class is included with this book.

inventory collection. If there are 10 shovels in the warehouse, there are 10 shovel documents in the database. Each inventory item is linked to a product by `sku`, and each of these items can be in one of four states: `AVAILABLE` (0), `IN_CART` (1), `PRE_ORDER` (2), or `PURCHASED` (3).

Here's a method that inserts three shovels, three rakes, and three sets of clippers as available inventory. The examples in this section are in Ruby, since transactions require more logic, so it's useful to see a more concrete example of how an application would implement them:

```
3.times do
  $inventory.insert_one({:sku => 'shovel',   :state => AVAILABLE})
  $inventory.insert_one({:sku => 'rake',     :state => AVAILABLE})
  $inventory.insert_one({:sku => 'clippers', :state => AVAILABLE})
end
```

We'll handle inventory management with a special inventory fetching class. We'll first look at how this fetcher works and then we'll peel back the covers to reveal its

implementation.

INVENTORY FETCHER

The inventory fetcher can add arbitrary sets of products to a shopping cart. Here you create a new order object and a new inventory fetcher. You then ask the fetcher to add three shovels and one set of clippers to a given order by passing an order ID and two documents specifying the products and quantities you want to the `add_to_cart` method. The fetcher hides the complexity of this operation, which is altering two collections at once:

```
$order_id = BSON::ObjectId('561297c5530a69dbc9000000')
$orders.insert_one({
    :_id => $order_id,
    :username => 'kbanker',
    :item_ids => []
  })

@fetcher = InventoryFetcher.new({
    :orders => $orders,
    :inventory => $inventory
  })

@fetcher.add_to_cart(@order_id,
    [
      {:sku => "shovel", :quantity => 3},
      {:sku => "clippers", :quantity => 1}
    ])

$orders.find({"_id" => $order_id}).each do |order|
    puts "\nHere's the order:"
    p order
end
```

The `add_to_cart` method will raise an exception if it fails to add every item to a cart. If it succeeds, the order should look like this:

```
{
  "_id" => BSON::ObjectId('4cdf3668238d3b6e3200000a'),
  "username" => "kbanker",
  "item_ids" => [
    BSON::ObjectId('4cdf3668238d3b6e32000001'),
    BSON::ObjectId('4cdf3668238d3b6e32000004'),
    BSON::ObjectId('4cdf3668238d3b6e32000007'),
    BSON::ObjectId('4cdf3668238d3b6e32000009')
     ]
  }
```

The `_id` of each physical inventory item will be stored in the order document. You can query for each of these items like this:

```
puts "\nHere's each item:"
order['item_ids'].each do |item_id|
  item = @inventory.find({"_id" => item_id}).each do |myitem|
    p myitem
  end
end
```

```
{
  "_id" => BSON::ObjectId('4cdf3668238d3b6e32000001'),
  "sku"=>"shovel",
  "state"=>1,
  "ts"=>"Sun Nov 14 01:07:52 UTC 2010"
}


{
  "_id"=>BSON::ObjectId('4cdf3668238d3b6e32000004'),
  "sku"=>"shovel",
  "state"=>1,
  "ts"=>"Sun Nov 14 01:07:52 UTC 2010"
}
{
  "_id"=>BSON::ObjectId('4cdf3668238d3b6e32000007'),
  "sku"=>"shovel",
  "state"=>1,
  "ts"=>"Sun Nov 14 01:07:52 UTC 2010"
}
```

INVENTORY MANAGEMENT

command resides at its core. The full source code for the `InventoryFetcher` is included with the source code of this book. We're not going to look at every line of code, but we'll highlight the three key methods that make it work.

First, when you pass a list of items to be added to your cart, the fetcher attempts to transition each item from the state of `AVAILABLE` to `IN_CART`. If at any point this operation fails (if any one item can't be added to the cart), the entire operation is rolled back. Have a look at the `add_to_cart` method that you invoked earlier:

```
def add_to_cart(order_id, *items)
  item_selectors = []
  items.each do |item|
    item[:quantity].times do
      item_selectors << {:sku => item[:sku]}
    end
  end
  transition_state(order_id, item_selectors,
```

```
      {:from => AVAILABLE, :to => IN_CART})
end
```

The `*items` syntax in the method arguments allows the user to pass in any number of objects, which are placed in an array called `items`. This method doesn't do much. It takes the specification for items to add to the cart and expands the quantities so that one item selector exists for each physical item that will be added to the cart. For instance, this document, which says that you want to add two shovels

```
{:sku => "shovel", :quantity => 2}
```

becomes this:

```
[{:sku => "shovel"}, {:sku => "shovel"}]
```

You need a separate query selector for each item you want to add to your cart. Thus, the method passes the array of item selectors to another method called `transition_state`. For example, the previous code specifies that the state should be transitioned from `AVAILABLE` to `IN_CART`:

```
def transition_state(order_id, selectors, opts={})
  items_transitioned = []
  begin # use a begin/end block so we can do error recovery

    for selector in selectors do
      query = selector.merge({:state => opts[:from]})
      physical_item = @inventory.find_and_modify({
          :query => query,
          :update => {
            '$set' => {
              :state => opts[:to],          # target state
              :ts => Time.now.utc           # get the current client time
            }
          }
        })


    if physical_item.nil?
      raise InventoryFetchFailure
    end

    items_transitioned << physical_item['_id']   # push item into array
    @orders.update_one({:_id => order_id}, {
        '$push' => {
          :item_ids => physical_item['_id']
        }
```

```
        })
      end # of for loop

    rescue Mongo::OperationFailure, InventoryFetchFailure
      rollback(order_id, items_transitioned, opts[:from], opts[:to])
      raise InventoryFetchFailure, "Failed to add #{selector[:sku]}"
    end

    return items_transitioned.size
  end
```

To transition state, each selector gets an extra condition, `{:state => AVAILABLE}`, and the selector is then passed to `findAndModify`, which, if matched, sets a timestamp and the item's new state. The method then saves the list of items transitioned and updates the order with the ID of the item just added.

GRACEFUL FAILURE

If the `findAndModify` command fails and returns `nil`, then you raise an `InventoryFetchFailure` exception. If the command fails because of networking errors, you rescue the inevitable `Mongo::OperationFailure` exception. In both cases, you rescue by rolling back all the items transitioned thus far and then raise an `InventoryFetchFailure`, which includes the SKU of the item that couldn't be added. You can then rescue this exception on the application layer to fail gracefully for the user.

All that now remains is to examine the rollback code:

```
def rollback(order_id, item_ids, old_state, new_state)
  @orders.update_one({"_id" => order_id},
                     {"$pullAll" => {:item_ids => item_ids}})

  item_ids.each do |id|
    @inventory. find_one_and_update({
        :query => {
          "_id" => id,
          :state => new_state
        },
        {
          :update => {
            "$set" => {
              :state => old_state,
              :ts => Time.now.utc
            }
          }
        })
  end
end
```

This topic covers

- Basic indexing concepts and theory
- Practical advice for managing indexes
- Using compound indexes for more complex queries
- Optimizing queries
- All the MongoDB indexing options

Indexes are enormously important. With the right indexes in place, MongoDB can use its hardware efficiently and serve your application's queries quickly. But the wrong indexes produce the opposite result: slow queries, slow writes, and poorly utilized hardware. It stands to reason that anyone wanting to use MongoDB effectively must understand indexing.

A COMPOUND INDEX

This index is good if all you need is a list of recipes for a given ingredient. But if you want to include any other information about the recipe in your search, you still have some scanning to do—once you know the page numbers where cauliflower is referenced, you then need to go to each of those pages to get the name of the recipe and what type of cuisine it is. This is better than paging through the whole book, but you can do better.

What can you do? Happily, there's a solution to the long-lost cauliflower recipe, and its answer lies in the use of compound indexes.

The two indexes you've created so far are single-key indexes: they both order only one key from each recipe. You're going to build yet another index for *The Cookbook Omega*, but this time, instead of using one key per index, you'll use two. Indexes that use more than one key like this are called *compound indexes*.

This compound index uses both ingredients and recipe name, in that order. You'll notate the index like this: `ingredient-name`.

Cashews
Cashew Marinade
1,215
Chicken with Cashews
88
Rosemary-Roasted Cashews
103

Cauliflower
Bacon Cauliflower Salad
875
Lemon-baked Cauliflower
89
Spicy Cauliflower Cheese Soup
47

Currants
Creamed Scones with Currants

Part of this index would look like what you see in figure 8.1.

The value of this index for a human is obvious. You can now search by ingredient and probably find the recipe you want, even if you remember only the initial part of the name. For a machine, it's still valuable for this use case and will keep the database from having to scan every recipe name listed for that ingredient. This compound index would be especially useful if, as with *The Cookbook Omega*, there were several hundred (or thousand) cauliflower recipes. Can you see why?

One thing to notice: with compound indexes, order matters. Imagine the reverse

INDEXING RULES

The goal of this section was to present an extended metaphor to provide you with a better mental model of indexes. From this metaphor, you can derive a few simple concepts:

1. Indexes significantly reduce the amount of work required to fetch documents. Without the proper indexes, the only way to satisfy a query is to scan all documents linearly until the query conditions are met. This frequently means scanning entire collections.
2. Only one single-key index will be used to resolve a query.[1] For queries containing multiple keys (say, `ingredient` and `recipe name`), a compound index containing those keys will best resolve the query.
3. An index on `ingredient` can and should be eliminated if you have a second index on `ingredient-name`. More generally, if you have a compound index on `a-b`, then a second index on `a` alone will be redundant, but not one on `b`.
4. The order of keys in a compound index matters.

Bear in mind that this cookbook analogy can be taken only so far. It's a model for understanding indexes, but it doesn't fully correspond to the way MongoDB's indexes work.

The preceding thought experiment hinted at a number of core indexing concepts. Here and throughout the rest of the chapter, we'll unpack those ideas.

SINGLE-KEY INDEXES

With a single-key index, each entry in the index corresponds to a single value from

each of the documents indexed. The default index on `_id` is a good example of a single-key index. Because this field is indexed, each document's `_id` also lives in an index for fast retrieval by that field.

---

[1] **COMPOUND-KEY INDEXES**

Although when starting with MongoDB 2.6 you can use more than one index for a query, it's best if you use only a single index. But you often need to query on more than one attribute, and you want such a query to be as efficient as possible. For example, imagine that you've built two indexes on the products collection from this book's e-commerce example: one index on `manufacturer` and another on `price`. In this case, you've created two entirely distinct data structures that, when traversed, are ordered like the lists you see in figure 8.2.

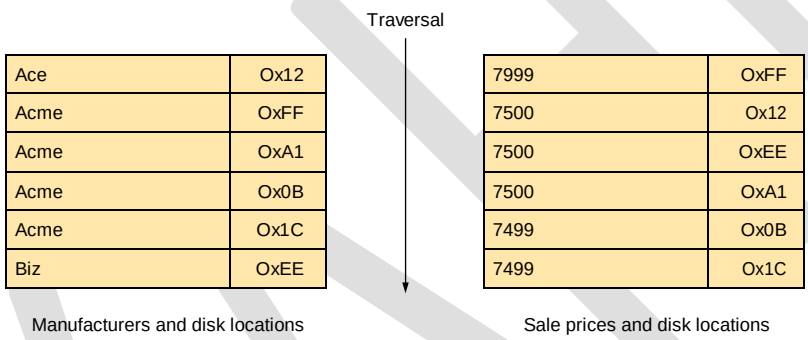


Figure 8.2 Single-key index traversal

Now, imagine your query looks like this:

```
db.products.find({
    'details.manufacturer': 'Acme',
    'pricing.sale': {
      $lt: 7500
    }
  })
```

This query says to find all Acme products costing less than $75.00. If you issue this query with single-key indexes on `manufacturer` and

`price`, only one of them will be used. grab the list of disk locations that match and calculate their intersection.

Ace – 8000	Ox12
Acme – 7999	OxFF
Acme – 7500	OxA1
Acme – 7499	Ox0B
Acme – 7499	Ox1C
Biz – 8999	OxEE

The order of keys in a compound index matters. If that seems clear, the second thing you should understand is why we've chosen the first ordering over the second. This may be obvious from the diagrams, but there's another way to look at the problem. Look again at the query: the two query terms specify different kinds of matches. On `manufacturer`, you want to match the term exactly. But on `price`, you want to match a range of values, beginning with 7500. As a general rule, a query where one term demands an exact match and another specifies a range requires a compound index where the range key comes second. We'll revisit this idea in the section on query optimization.

INDEX EFFICIENCY

Although indexes are essential for good query performance, each new index imposes a small maintenance cost. Whenever you add a document to a collection, each index on that collection must be modified to include the new document. If a particular collection has 10 indexes, that makes 10 separate structures to modify on each insert, in addition to writing the document itself.

For read-intensive applications, the cost of indexes is almost always justified. Just realize that indexes do impose a cost and that they therefore must be chosen with care. This means ensuring that all your indexes are used and that none are redundant. You can do this in part by profiling your application's queries; we'll describe this process later in the chapter.

But there's a second consideration. Even with all the right indexes in place, it's still possible that those indexes won't result in faster queries. This occurs when indexes and a working data set don't fit in RAM.

You may recall from chapter 1 that MongoDB tells the operating system to map all data files to memory using the `mmap()` system call when the MMAPv1 default storage engine is used. As you'll learn in chapter 10, the WiredTiger storage engine works differently. From this point on, the data files, which include all documents, collections, and their indexes, are swapped in and out of RAM by the operating system in 4 KB chunks called pages.[2] Whenever data from a given page is requested, the operating

system must ensure that the page is available in RAM. If it's not, a kind of exception known as a *page fault* is raised, and this tells the memory manager to load the page from disk into RAM.

will eventually be loaded into mem- ory. Whenever that memory is altered, as in the case of a write, those changes will be flushed to disk asynchronously by the OS. The write, however, will be fast because it occurs directly in RAM; thus the number of disk accesses is reduced to a minimum. But if the working data set can't fit into RAM, page faults will start to creep up. This means that the operating system will be going to disk frequently, greatly slowing read and write operations. In the worst case, as data size becomes much larger than available RAM, a situation can occur where for any read or write, data must be paged to and from disk. This is known as *thrashing*, and it causes performance to take a severe dive.

Fortunately, this situation is relatively easy to avoid. At a minimum, you need to make sure that your indexes will fit in RAM. This is one reason why it's important to avoid creating any unneeded indexes. With extra indexes in place, more RAM will be required to maintain those indexes. Along the same lines, each index should have only the keys it needs. A triple-key compound index might be necessary at times, but be aware that it'll use more space than a simple single-key index. One example of where it might be valuable to create an index with more than one or two fields is if you can create a covering index for a frequent query.

Bear in mind that indexes are stored separately in RAM from the data they index and aren't clustered. In a clustered index, the order of the index corresponds directly to the order of the underlying data; if you index recipes by name in a clustered index, then all of the recipes starting with A will be stored together, followed by B, C, and so on. This isn't the case in MongoDB. Every name in the recipe index is essentially duplicated in the index, and the order of these names has no bearing on the order of the data. This is important when you scan through a collection sorted with an index because it means that every document fetched could be anywhere in the data set. There's no guaranteed locality with the previously fetched data.

Ideally, indexes and a working data set fit in RAM. But estimating how much RAM this requires for any given deployment isn't always easy. You can always discover total index size by looking at the results of the `stats` command. The working set is the subset of total data commonly queried and updated, which is different for every application. Suppose you have a million users for which you have data. If only half of them are active (thus half the user documents are queried), then your working set for the

user collection is half the total data size. If these documents are evenly distributed throughout the entire data set, though, it's likely that untouched user documents are also being loaded into memory, which imposes a cost.

B-trees have two overarching traits that make them ideal for database indexes. First, they facilitate a variety of queries, including exact matches, range conditions, sorting, prefix matching, and index-only queries. Second, they're able to remain balanced in spite of the addition and removal of keys.

We'll look at a simple representation of a B-tree and then discuss some principles that you'll want to keep in mind. Imagine that you have a collection of users and that

A *B-tree*, as you might guess, is a tree-like data structure. Each node in the tree can contain multiple keys. You can see in the example that the root node contains two keys, each of which is in the form of a BSON object representing an indexed value from the `users` collection. In reading the contents of the root node, you can see the keys for two documents, indicating last names Edwards and Perry, with ages of 21 and 18, respectively. Each of these keys includes two pointers: one to the data file it belongs to and another to the child node. Additionally, the node itself points to another node with values less than the node's smallest value.

In MongoDB's B-tree implementation, a new node is allocated 8,192 bytes, which means that in practice, each node may contain hundreds of keys. This depends on the average index key size; in this case, that average key size might be around 30 bytes. The maximum key size since MongoDB v2.0 is 1024 bytes. Add to this a per-key overhead of 18 bytes and a per-node overhead of 40 bytes, and this results in about 170 keys per node.[4] One thing to notice is that each node has some empty space (not to scale).

This is relevant because users frequently want to know why index sizes are what they are. You now know that each node is 8 KB, and you can estimate how many keys will fit into each node. To calculate this, keep in mind that B-tree nodes are usually intentionally kept around 60% full by default.

Given this information, you should now see why indexes aren't free, in terms of space or time required to update them. Use this information to help decide when to create indexes on your collections and when to avoid them.

Indexing in practice

With most of the theory behind us, we'll now look at some refinements on our concept of indexing in MongoDB. We'll then proceed to some of the details of index administration.

Index types

MongoDB uses B-trees for indexes and allows you to apply several characteristics to these indexes. This section should give you an overview of your options when creating indexes.

UNIQUE INDEXES

Often you want to ensure that a field in your document, such as `_id` or `username`, is unique to that document. Unique indexes are a way to enforce this characteristic, and in fact are used by MongoDB to ensure that `_id` is a unique primary key.

To create a unique index, specify the `unique` option:

```
db.users.createIndex({username: 1}, {unique: true})
```

Unique indexes enforce uniqueness across all their entries. If you try to insert a document into this book's sample application's `users` collection with an already-indexed username value, the insert will fail with the following exception:

```
E11000 duplicate key error index:
  gardening.users.$username_1 dup key: { : "kbanker" }
```

If using a driver, this exception will be caught only if you perform the insert using your driver's safe mode, which is the default. You may have also encountered this error if you attempted to insert two documents with the same `_id`—every MongoDB collection has a unique index on this field because it's the primary key.

If you need a unique index on a collection, it's usually best to create the index before inserting any data. If you create the index in advance, you guarantee the uniqueness constraint from the start. When creating a unique index on a collection that already contains data, you run the risk of failure because it's possible that duplicate keys may already exist in the collection. When duplicate keys exist, the index creation fails.

If you do find yourself needing to create a unique index on an established collection, you have a couple of options. The first is to repeatedly attempt to create the unique index and use the failure messages to manually remove the documents with duplicate keys. But if the data isn't so important, you can also instruct the database to drop documents with duplicate keys automatically using the `dropDups` option. For example, if your `users` collection already contains data, and if you don't care that documents with duplicate keys are removed, you can issue the index creation command like this:

```
db.users.createIndex({username: 1}, {unique: true, dropDups: true})
```

SPARSE INDEXES

Indexes are dense by default. This means that for every document in an indexed collection, a corresponding entry exists in the index, even if the document lacks the indexed key. For example, recall the products collection from your e-commerce data model, and imagine that you've built an index on the product attribute `category_ids`. Now suppose that a few products haven't been assigned to any categories. For each of these category-less products, there will still exist a null entry in the `category_ids` index. You can query for those null values like this:

```
db.products.find({category_ids: null})
```

Here, when searching for all products lacking a category, the query optimizer will still be able to use the index on `category_ids` to locate the corresponding products.

But in two cases a dense index is undesirable. The first is when you want a unique index on a field that doesn't appear in every document in the collection. For instance, you definitely want a unique index on every product's `sku` field. But suppose that, for some reason, products are entered into the system before a SKU is assigned. If you have a unique index on `sku` and attempt to insert more than one product without a SKU, the first insert will succeed, but all subsequent inserts will fail because there will already be an entry in the index where `sku` is `null`. This is a case where a dense index doesn't serve your purpose. Instead you want a unique and sparse index.

In a sparse index, only those documents having some value for the indexed key will appear. If you want to create a sparse index, all you have to do is specify `{sparse: true}`. For example, you can create a unique sparse index on `sku` like this:

```
db.products.createIndex({sku: 1}, {unique: true, sparse: true})
```

There's another case where a sparse index is desirable: when a large number of documents in a collection don't contain the indexed key. For example, suppose you allowed anonymous reviews on your e-commerce site. In this case, half the reviews might lack a `user_id` field, and if that field were indexed, half the entries in that index would be `null`. This would be inefficient for two reasons.

the size of the index. Second, it'd require updates to the index when adding and removing documents with null `user_id` fields.

If you rarely (or never) expect queries on anonymous reviews, you might elect to

build a sparse index on `user_id`. Again, setting the `sparse` option is simple:

```
db.reviews.createIndex({user_id: 1}, {sparse: true, unique: false})
```

Now only those reviews linked to a user via the `user_id` field will be indexed.

MULTIKEY INDEXES

In earlier chapters you saw several examples of indexing fields whose values are arrays.[5] This is made possible by what's known as a *multikey index*, which allows multiple entries in the index to reference the same document. This makes sense if we look at a simple example. Suppose you have a product document with a few tags like this:

```
{
  name: "Wheelbarrow",
  tags: ["tools", "gardening", "soil"]
}
```

If you create an index on `tags`, then each value in this document's `tags` array will appear in the index. This means that a query on any one of these array values can use the index to locate the document. This is the idea behind a multikey index: multiple index entries, or keys, end up referencing the same document.

Multikey indexes are always enabled in MongoDB, with a few exceptions, such as with hashed indexes. Whenever an indexed field contains an array, each array value will be given its own entry in the index.

The intelligent use of multikey indexes is essential to proper MongoDB schema design. This should be evident from the examples presented in chapters 4 through 6; several more examples are provided in the design patterns section of appendix B. But creating, updating, or deleting multikey indexes is more expensive than creating, updating, or deleting single-key indexes.

HASHED INDEXES

In the previous examples of B-tree indexes, we showed how MongoDB builds the index tree out of the values being indexed. Thus, in an index of recipes, the "Apple Pie" entry is near the "Artichoke Ravioli" entry. This may seem obvious and natural, but MongoDB also supports *hashed indexes* where the entries are first passed through a hash function.[6] This means the hashed values will determine the ordering, so these recipes will likely not be near each other in the index.

Indexes of this kind can be created in MongoDB by passing `'hashed'` as the index sorting direction. For example:

```
db.recipes.createIndex({recipe_name: 'hashed'})
```

Because the indexed value is a hash of the original, these indexes carry some restrictions:

- Equality queries will work much the same, but range queries aren't supported.
- Multikey hashed indexes aren't allowed.

Given these restrictions and peculiarities, you may wonder why anyone would use a hashed index. The answer lies in the fact that the entries in a hashed index are evenly distributed. In other words, when you have a non-uniform distribution of key data, then a hashed index will create uniformity if you can live with its restrictions. Recall that "Apple Pie" and "Artichoke Ravioli" are no longer next to each other in the hashed index; the data locality of the index has changed. This is useful in sharded collections where the shard index determines which shard each document will be assigned to. If your shard index is based on an increasing value, such as a MongoDB OIDs,[7] then new documents created will only be inserted to a single shard—unless the index is hashed.

Let's dig into that statement. Unless explicitly set, a MongoDB document will use an OID as its primary key. Here are a few sequentially generated OIDs:

```
5247ae72defd45a1daba9da9
5247ae73defd45a1daba9daa
5247ae73defd45a1daba9dab
```

Notice how similar the values are; the most significant bits are based on the time when they were generated. When new documents are inserted with these IDs, their index entries are likely to be near each other. If the index using these IDs is being used to decide which shard (and thus machine) a document should reside on, these documents are also likely to be inserted on to the same machine. This can be detrimental if a collection is receiving heavy write load, because only a single machine is being used. Hashed indexes solve this issue by distributing these documents evenly in a namespace, and thus across shards and machines. To fully understand this example, wait until you read chapter 12.

For now, the important thing to remember is that hashed indexes change the locality

of index entries, which can be useful in sharded collections.

GEOSPATIAL INDEXES

Another useful query capability is to find documents "close" to a given location, based on latitude and longitude values stored in each document. If you store a directory of restaurants in a MongoDB collection, for example, users are probably most eager to find restaurants located near their home. One answer to this is to run a query to find every restaurant within a 10-mile radius. Executing this query requires an index that can efficiently calculate geographic distances, including the curvature of the earth. Geospatial indexes can handle this and other types of queries.

Index administration

We've discussed simple index administration, such as creating indexes, in this and in previous chapters. When you use indexes in real-world applications, however, it's useful to understand this topic in greater depth. Here we'll see index creation and deletion in detail and address questions surrounding compaction and backups.

CREATING AND DELETING INDEXES

By now you've created quite a few indexes, so this should be easy. Simply call `createIndex()` either in the shell or with your language of choice. Please note that in MongoDB v3.0, `ensureIndex()`, which was previously used for creating indexes, has been replaced by the `createIndex()` command and shouldn't be used anymore. What you may not know is that this method works by creating a document defining the new index and putting it into the special `system.indexes` collection.

Though it's usually easier to use a helper method to create an index, you can also insert an index specification manually (this is what the helper methods do). You need to be sure you've specified the minimum set of keys: `ns`, `key`, and `name`. `ns` is the namespace, `key` is the field or combination of fields to index, and `name` is a name used to refer to the index. Any additional options, like `sparse`, can also be specified here. For example, let's create a sparse index on the `users` collection:

```
use green
spec = {ns: "green.users", key: {'addresses.zip': 1}, name: 'zip'}
db.system.indexes.insert(spec, true)
```

If no errors are returned on insert, the index now exists, and you can query the `system.indexes` collection to prove it:

```
db.system.indexes.find().pretty()
{
  "ns" : "green.users",
```

```
    "key" : {
      "addresses.zip" : 1
    },
    "name" : "zip",
    "v" : 1
  }
```

The `v` field was added in MongoDB v2.0 to store the version of the index. This version field allows for future changes in the internal index format but should be of little concern to application developers.

To delete an index, you might think that all you need to do is remove the index document from `system.indexes`, but this operation is prohibited. Instead, you must delete indexes using the database command `deleteIndexes`. As with index creation, there are helpers for deleting indexes, but if you want to run the command itself, you can do that, too. The command takes as its argument a document containing the collection name and either the name of the index to drop or `*` to drop all indexes. To manually drop the index you created, issue the command like this:

```
use green
db.runCommand({deleteIndexes: "users", index: "zip"})
```

In most cases, you'll use the shell's helpers to create and drop indexes:

```
use green
db.users.createIndex({zip: 1})
```

You can then check the index specifications with the `getIndexSpecs()` method:

```
> db.users.getIndexes()
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "ns" : "green.users",
    "name" : "_id_"
  },
  {
    "v" : 1,
    "key" : {
      "zip" : 1
    },
    "ns" : "green.users",
    "name" : "zip_1"
  }
]
```

Finally, you can drop the index using the `dropIndex()` method. Note that you must supply the index's name as specified in the spec:

```
use green
db.users.dropIndex("zip_1")
```

You can also supply your own name while creating an index using the `name` parameter.

Those are the basics of creating and deleting indexes. For what to expect when an index is created, read on.

BUILDING INDEXES

Most of the time, you'll want to declare your indexes before putting your application into production. This allows indexes to be built incrementally, as the data is inserted. But there are two cases where you might choose to build an index after the fact. The first case occurs when you need to import a lot of data before switching into production. For instance, you might be migrating an application to MongoDB and need to seed the database with user information from a data warehouse. You could create the indexes on your user data in advance, but doing so after you've imported the data will ensure an ideally balanced and compacted index from the start. It'll also minimize the net time to build the index.

The second (and more obvious) case for creating indexes on existing data sets is when you have to optimize for new queries. This occurs when you add or change functionality in your application, and it happens more than you might think. Suppose you allow users to log in using their username, so you index that field. Then you modify your application to also allow your users to log in using their email; now you probably need a second index on the `email` field. Watch out for cases like these because they require rethinking your indexing.

Regardless of why you're creating new indexes, the process isn't always pleasing. For large data sets, building an index can take hours, even days. But you can monitor the progress of an index build from the MongoDB logs. Let's take an example from a data set that we'll use in the next section. First, you declare an index to be built:

```
db.values.createIndex({open: 1, close: 1})
```

The index builds in two steps. In the first step, the values to be indexed are sorted. A sorted data set makes for a much more efficient insertion into the B-tree. If you look at the MongoDB server log, you'll see the progress printed for long index builds. Note that the progress of the sort is indicated by the ratio of the number of documents

sorted to the total number of documents:

```
[conn1] building new index on { open: 1.0, close: 1.0 } for stocks.values
    1000000/4308303  23%
    2000000/4308303  46%
    3000000/4308303  69%
    4000000/4308303  92%
    Tue Jan  4 09:59:13 [conn1]    external sort used : 5 files in 55 secs
```

For step two, the sorted values are inserted into the index. Progress is indicated in the same way, and when complete, the time it took to complete the index build is indicated as the insert time into `system.indexes`:

```
1200300/4308303  27%
    2227900/4308303  51%
    2837100/4308303  65%
    3278100/4308303  76%
    3783300/4308303  87%
    4075500/4308303  94%
Tue Jan 4 10:00:16 [conn1] done building bottom layer, going to commit
Tue Jan 4 10:00:16 [conn1] done for 4308303 records 118.942secs
Tue Jan  4 10:00:16 [conn1] insert stocks.system.indexes 118942ms
```

In addition to examining the MongoDB log, you can check the index build progress by running the shell's `currentOp()` method. This command's output varies from version to version, but it will probably look something like the next listing.[8]

Listing 8.1 Checking the index build process with the shell `currentOP()` method

```
> db.currentOp()
{
```

```
                                "inprog" : [
                                  {
            "opid" : 83695, "active" : true, "secs_running" : 55, "op" : "insert",
            "ns" : "stocks.system.indexes", "insert" : {
                                      "v" : 1,
                                      "key" : {
                                        "desc" : 1
                                      },
              "ns" : "stocks.values",
              "name" : "desc_1"
                                    },
            "client" : "127.0.0.1:56391",
            "desc" : "conn12", "threadId" : "0x10f20c000", "connectionId" : 12, "locks" :
            {
                                      "^" : "w",
              "^stocks" : "W"

  },
            "waitingForLock" : false,
            "msg" : "index: (1/3) external sort Index: (1/3)
                     External Sort Progress: 3999999/4308303 92%",
```

8

```
              "progress" : {
                "done" : 3999999,
                "total" : 4308303
              },
              "numYields" : 0,
              "lockStats" : {
                "timeLockedMicros" : {},
                "timeAcquiringMicros" : {
                  "r" : NumberLong(0),
                  "w" : NumberLong(723)
                }
              }
            }
          ]
        }
```

The `msg` field describes the build's progress. Note also the `locks` element, which indicates that the index build takes a write lock on the `stocks` database. This means that no other client can read or write from the database at this time. If you're running in production, this is obviously a bad thing, and it's the reason why long index builds can be so vexing. Let's explore two possible solutions to this problem.

BACKGROUND INDEXING

If you're running in production and can't afford to halt access to the database, you can specify that an index be built in the background. Although the index build will

still take a write lock, the job will yield to allow other readers and writers to access the database. If your application typically exerts a heavy load on MongoDB, a background index build will degrade performance, but this may be acceptable under certain circumstances. For example, if you know that the index can be built within a time window where application traffic is at a minimum, background indexing in this case might be a good choice.

To build an index in the background, specify `{background: true}` when you declare the index. The previous index can be built in the background like this:

```
db.values.createIndex({open: 1, close: 1}, {background: true})
```

Offline indexing

Building an index in the background may still put an unacceptable amount of load on a production server. If this is the case, you may need to index the data offline. This will usually involve taking a replica node offline, building the index on that node by itself, and then allowing the node to catch up with the master replica. Once it's caught up, you can promote the node to primary and then take another secondary offline and build its version of the index. This tactic presumes that your replication oplog is large enough to prevent the offline node from becoming stale during the index build. Chapter 10 covers replication in detail and should help you plan for a migration such as this.

Backups

Because indexes are hard to build, you may want to back them up. Unfortunately, not all backup methods include indexes. For instance, you might be tempted to use `mongodump` and `mongorestore`, but these utilities preserve collections and index declarations only. This means that when you run `mongorestore`, all the indexes declared for any collections you've backed up will be re-created. As always, if your data set is large, the time it takes to build these indexes may be unacceptable.

Consequently, if you want your backups to include indexes, you'll want to opt for backing up the MongoDB data files themselves. More details about this, as well as general instructions for backups, can be found in chapter 13.

Defragmenting

If your application heavily updates existing data or performs a lot of large deletions, you may end up with a highly fragmented index. B-trees will coalesce on their own somewhat, but this isn't always sufficient to offset a high delete volume. The primary symptom of a fragmented index is an index size much larger than you'd expect for the given data size. This fragmented state can result in indexes using more RAM than necessary. In these cases, you may want to consider rebuilding one or more indexes. You can do this by dropping and re-creating individual indexes or by running the `reIndex` command, which will rebuild all indexes for a given collection:

```
db.values.reIndex();
```

Be careful about reindexing: the command will take out a write lock for the duration of the rebuild, temporarily rendering your MongoDB instance unusable. Reindexing is best done offline, as described earlier for building indexes on a secondary. Note

that the `compact` command, discussed in chapter 10, will also rebuild indexes for the collection on which it's run.

We've discussed how to create and manage your indexes, but despite this knowledge, you may still find yourself in a situation where your queries aren't fast enough. This can occur as you add data, traffic, or new queries. Let's learn how to identify these queries that could be faster and improve the situation.

Query optimization

Query optimization is the process of identifying slow queries, discovering why they're slow, and then taking steps to speed them up. In this section, we'll look at each step of the query optimization process in turn so that by the time you finish reading, you'll have a framework for addressing problematic queries on any MongoDB installation.

Before diving in, we must warn you that the techniques presented here can't be used to solve every query performance problem. The causes of slow queries vary too much. Poor application design, inappropriate data models, and insufficient physical hardware are all common culprits, and their remedies require a significant time investment. Here we'll look at ways to optimize queries by restructuring the queries themselves and by building the most useful indexes. We'll also describe other avenues for investigation when these techniques fail to deliver.

`ed` is `0`.

UNIT-IV
SYLLABUS

Replication: Overview – Replica sets – Master-slave replication – Drivers and replication. Shading: Overview – A sample shard cluster – Querying and indexing a shard cluster – Choosing a shard key – sharding in production.

Replication overview

Replication is the distribution and maintenance of data across multiple MongoDB servers (nodes). MongoDB can copy your data to one or more nodes and constantly keep them in sync when changes occur. This type of replication is provided through a mechanism called *replica sets*, in which a group of nodes are configured to automatically synchronize their data and fail over when a node disappears. MongoDB also supports an older method of replication called *master-slave*, which is now considered deprecated, but master-slave replication is still supported and can be used in MongoDB v3.0. For both methods, a single primary node receives all writes, and then all secondary nodes read and apply those writes to themselves asynchronously.

Why replication matters

- The network connection between the application and the database is lost.
- Planned downtime prevents the server from coming back online as expected. Most hosting providers must schedule occasional downtime, and the results of this downtime aren't always easy to predict. A simple reboot will keep a database server offline for at least a few minutes. Then there's the question of what happens when the reboot is complete. For example, newly installed software or

---

hardware can prevent MongoDB or even the operating system from starting up properly.

- There's a loss of power. Although most modern datacenters feature redundant power supplies, nothing prevents user error within the datacenter itself or an extended brownout or blackout from shutting down your database server.
- A hard drive fails on the database server. Hard drives have a mean time to failure of a few years and fail more often than you might think.[2] Even if it's acceptable to have occasional downtime for your MongoDB, it's probably not acceptable to lose your data if a hard drive fails. It's a good idea to have at least one copy of your data, which replication provides.

Replication use cases and limitations

In addition to providing redundancy and failover, replication simplifies maintenance, usually by allowing you to run expensive operations on a node other than the primary. For example, it's common practice to run backups on a secondary node to

keep unnecessary load off the primary and to avoid downtime. Building large indexes is another example. Because index builds are expensive, you may opt to build on a secondary node first, swap the secondary with the existing primary, and then build again on the new secondary.

Finally, replication allows you to balance reads across replicas. For applications whose workloads are overwhelmingly read-heavy, this is the easiest, or if you prefer, the most naïve, way to scale MongoDB. But for all its promise, a replica set doesn't help much if any of the following apply:

- The allotted hardware can't process the given workload. As an example, we mentioned working sets in the previous chapter. If your working data set is much larger than the available RAM, then sending random reads to the secondaries likely won't improve your performance as much as you might hope. In this scenario, performance becomes constrained by the number of I/O operations per second (IOPS) your disk can handle—generally around 80–100 for non-SSD hard drives. Reading from a replica increases your total IOPS, but going from 100 to 200 IOPS may not solve your performance problems, especially if writes are occurring at the same time and consuming a portion of that number. In this case, sharding may be a better option.
- The ratio of writes to reads exceeds 50%. This is an admittedly arbitrary ratio, but it's a reasonable place to start. The issue here is that every write to the primary must eventually be written to all the secondaries as well. Therefore, directing reads to secondaries that are already processing a lot of writes can sometimes slow the replication process and may not result in increased read throughput.

Replica sets

Replica sets are the recommended MongoDB replication strategy. We'll start by configuring a sample replica set. We'll then describe how replication works because this knowledge is incredibly important for diagnosing production issues. We'll end by discussing advanced configuration details, failover and recovery, and best deployment practices.

Setup

The minimum recommended replica set configuration consists of three nodes, because in a replica set with only two nodes you can't have a majority in case the primary server goes down. A three-member replica set can have either three members that hold data or two members that hold data and an arbiter. The primary is the only member in the set that can accept write operations. Replica set members go through a process in which they "elect" a new master by voting. If a primary becomes unavailable, elections allow the set to recover normal operations without manual intervention. Unfortunately, if a majority of the replica set is inaccessible or unavailable, the replica set cannot accept writes and all remaining members become read-only.

You may consider adding an arbiter to a replica set if it has an equal number of nodes in two places where network partitions between the places are possible. In such cases, the arbiter will break the tie between the two facilities and allow the set to elect a new primary.

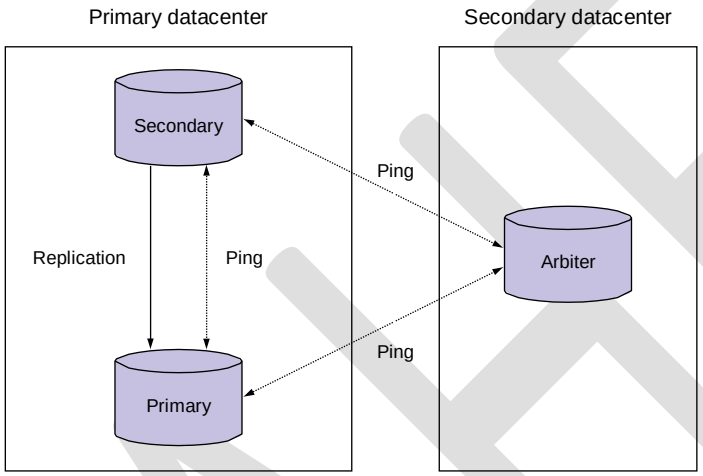


Figure 11.1 A basic replica set consisting of a primary, a secondary, and an arbiter

Begin by creating a data directory for each replica set member:

```
mkdir ~/node1
mkdir ~/node2
mkdir ~/arbiter
```

Next, start each member as a separate `mongod`. Because you'll run each process on the same machine, it's easiest to start each `mongod` in a separate terminal window:

```
mongod --replSet myapp --dbpath ~/node1 --port 40000
mongod --replSet myapp --dbpath ~/node2 --port 40001
mongod --replSet myapp --dbpath ~/arbiter --port 40002
```

Note how we tell each `mongod` that it will be a member of the `myapp` replica set and that we start each `mongod` on a separate port. If you examine the `mongod` log output, the first thing you'll notice are error messages saying that the configuration can't be found. This is completely normal:

```
[rsStart] replSet info you may need to run replSetInitiate
    -- rs.initiate() in the shell -- if that is not already done
[rsStart] replSet can't get local.system.replset config from self
    or any seed (EMPTYCONFIG)
```

On MongoDB v3.0 the log message will be similar to the following:

```
2015-09-15T16:27:21.088+0300 I REPL [initandlisten] Did not find local
replica set configuration document at startup; NoMatchingDocument Did not
find replica set configuration document in local.system.replset
```

To proceed, you need to configure the replica set. Do so by first connecting to one of the non-arbiter `mongod`s just started. These instances aren't running on MongoDB's default port, so connect to one by running

```
mongo --port 40000
```

These examples were produced running these `mongod` processes locally, so you'll see the name of the example machine, `iron`, pop up frequently; substitute your own hostname.

Connect, and then run the `rs.initiate()` command:[3]

```
> rs.initiate()
{
    "info2" : "no configuration explicitly specified -- making one",
    "me" : "iron.local:40000",
    "info" : "Config now saved locally. Should come online in about a
     minute.",
    "ok" : 1
}
```

On MongoDB v3.0 the output will be similar to the following:

```
{
    "info2" : "no configuration explicitly specified -- making one",
    "me" : "iron.local:40000",
    "ok" : 1
}
```

Within a minute or so, you'll have a one-member replica set. You can now add the other two members using `rs.add()`:

```
> rs.add("iron.local:40001")
{ "ok" : 1 }
> rs.add("iron.local:40002", {arbiterOnly: true})
{ "ok" : 1 }
```

On MongoDB v3.0 you can also add an arbiter with the following command:

```
> rs.addArb("iron.local:40002")
{ "ok" : 1 }
```

Note that for the second node, you specify the `arbiterOnly` option to create an arbiter. Within a minute, all members should be online. To get a brief summary of the replica set status, run the `db.isMaster()` command:

```
> db.isMaster()
{
  "setName" : "myapp",
  "ismaster" : true,
  "secondary" : false,
  "hosts" : [
```

```
    "iron.local:40001",
    "iron.local:40000"
  ],
```

```
  "arbiters" : [
    "iron.local:40002"
  ],
  "primary" : "iron.local:40000",
  "me" : "iron.local:40000",
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
  "localTime" : ISODate("2013-11-06T05:53:25.538Z"),
  "ok" : 1
}
```

The same command produces the following output on a MongoDB v3.0 machine:

```
myapp:PRIMARY> db.isMaster()
{
    "setName" : "myapp",
    "setVersion" : 5,
    "ismaster" : true,
    "secondary" : false,
    "hosts" : [
        "iron.local:40000",
        "iron.local:40001"
    ],
    "arbiters" : [
        "iron.local:40002"
    ],
    "primary" : "iron.local:40000",
    "me" : "iron.local:40000",
    "electionId" : ObjectId("55f81dd44a50a01e0e3b4ede"),
    "maxBsonObjectSize" : 16777216,
    "maxMessageSizeBytes" : 48000000,
    "maxWriteBatchSize" : 1000,
    "localTime" : ISODate("2015-09-15T13:37:13.798Z"),
    "maxWireVersion" : 3,
    "minWireVersion" : 0,
    "ok" : 1
}
```

A more detailed view of the system is provided by the `rs.status()` method. You'll see state information for each node. Here's the complete status listing:

```
> rs.status()
{
  "set" : "myapp",
  "date" : ISODate("2013-11-07T17:01:29Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "iron.local:40000",
```

```
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 1099,

    "optime" : Timestamp(1383842561, 1),
    "optimeDate" : ISODate("2013-11-07T16:42:41Z"),
    "self" : true
  },
  {
    "_id" : 1,
    "name" : "iron.local:40001",
    "health" : 1,
    "state" : 2,
    "stateStr" : "SECONDARY",
    "uptime" : 1091,
    "optime" : Timestamp(1383842561, 1),
    "optimeDate" : ISODate("2013-11-07T16:42:41Z"),
    "lastHeartbeat" : ISODate("2013-11-07T17:01:29Z"),
    "lastHeartbeatRecv" : ISODate("2013-11-07T17:01:29Z"),
    "pingMs" : 0,

    "lastHeartbeatMessage" : "syncing to: iron.1ocal:40000",
    "syncingTo" : "iron.local:40000"
  },
  {
    "_id" : 2,
    "name" : "iron.local:40002",
    "health" : 1,
    "state" : 7,
    "stateStr" : "ARBITER",
    "uptime" : 1089,
    "lastHeartbeat" : ISODate("2013-11-07T17:01:29Z"),
    "lastHeartbeatRecv" : ISODate("2013-11-07T17:01:29Z"),
    "pingMs" : 0
  }
],
    "ok" : 1
}
```

The `rs.status()` command produces a slightly different output on a MongoDB v3.0 server:

```
{
    "set" : "myapp",
    "date" : ISODate("2015-09-15T13:41:58.772Z"),
    "myState" : 1,
    "members" : [
        {
            "_id" : 0,
            "name" : "iron.local:40000",
            "health" : 1,
            "state" : 1,
            "stateStr" : "PRIMARY",
```

```
            "uptime" : 878,
            "optime" : Timestamp(1442324156, 1),
            "optimeDate" : ISODate("2015-09-15T13:35:56Z"),
            "electionTime" : Timestamp(1442323924, 2),
            "electionDate" : ISODate("2015-09-15T13:32:04Z"),
            "configVersion" : 5,


            "self" : true
        },
        {
            "_id" : 1,
            "name" : "iron.local:40001",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 473,
            "optime" : Timestamp(1442324156, 1),
            "optimeDate" : ISODate("2015-09-15T13:35:56Z"),
            "lastHeartbeat" : ISODate("2015-09-15T13:41:56.819Z"),
            "lastHeartbeatRecv" : ISODate("2015-09-15T13:41:57.396Z"),
            "pingMs" : 0,
            "syncingTo" : "iron.local:40000",
            "configVersion" : 5
        },
        {
            "_id" : 2,
            "name" : "iron.local:40002",
            "health" : 1,
            "state" : 7,
            "stateStr" : "ARBITER",
            "uptime" : 360,
            "lastHeartbeat" : ISODate("2015-09-15T13:41:57.676Z"),
            "lastHeartbeatRecv" : ISODate("2015-09-15T13:41:57.676Z"),
            "pingMs" : 10,
            "configVersion" : 5
        }
    ],
    "ok" : 1
}
```

Unless your MongoDB database contains a lot of data, the replica set should come online within 30 seconds. During this time, the `stateStr` field of each node should transition from `RECOVERING` to `PRIMARY`, `SECONDARY`, or `ARBITER`.

Now even if the replica set status claims that replication is working, you may want to see some empirical evidence of this. Go ahead and connect to the primary node with the shell and insert a document:

```
$ mongo --port 40000
myapp:PRIMARY> use bookstore
switched to db bookstore
myapp:PRIMARY> db.books.insert({title: "Oliver Twist"})
myapp:PRIMARY> show dbs
bookstore 0.203125GB
```

```
local 0.203125GB
```

Notice how the MongoDB shell prints out the replica set membership status of the instance it's connected to.

Initial replication of your data should occur almost immediately. In another terminal window, open a new shell instance, but this, time point it to the secondary node. Query for the document just inserted; it should have arrived:

```
$ mongo --port 40001
myapp:SECONDARY> show dbs
bookstore 0.203125GB
local 0.203125GB
myapp:SECONDARY> use bookstore
switched to db bookstore
myapp:SECONDARY> rs.slaveOk()
myapp:SECONDARY> db.books.find()
{ "_id" : ObjectId("4d42ebf28e3c0c32c06bdf20"), "title" : "Oliver Twist" }
```

`~/node1` and run `kill -3 <processid>`. You can also connect to the primary using the shell and run commands to shut down the server:

```
$ mongo --port 40000

PRIMARY> use admin
PRIMARY> db.shutdownServer()
```

Once you've killed the primary, note that the secondary detects the lapse in the primary's heartbeat. The secondary then elects itself primary. This election is possible because a majority of the original nodes (the arbiter and the original secondary) are still able to ping each other. Here's an excerpt from the secondary node's log:

```
Thu Nov 7 09:23:23.091 [rsHealthPoll] replset info iron.local:40000
    heartbeat failed, retrying
Thu Nov 7 09:23:23.091 [rsHealthPoll] replSet info iron.local:40000
    is down (or slow to respond):
Thu Nov 7 09:23:23.091 [rsHealthPoll] replSet member iron.local:40000
    is now in state DOWN
Thu Nov 7 09:23:23.092 [rsMgr] replSet info electSelf 1
Thu Nov 7 09:23:23.202 [rsMgr] replSet PRIMARY
```

If you connect to the new primary node and check the replica set status, you'll see that the old primary is unreachable:

```
$ mongo --port 40001

> rs.status()
...
  {
    "_id" : 0,
    "name" : "iron.local:40000",
    "health" : 0,
    "state" : 8,
```

```
      "stateStr" : "(not reachable/healthy)",
      "uptime" : 0,
      "optime" : Timestamp(1383844267, 1),
      "optimeDate" : ISODate("2013-11-07T17:11:07Z"),
      "lastHeartbeat" : ISODate("2013-11-07T17:30:00Z"),
      "lastHeartbeatRecv" : ISODate("2013-11-07T17:23:21Z"),
      "pingMs" : 0
    },
...
```

Post-failover, the replica set consists of only two nodes. Because the arbiter has no data, your application will continue to function as long as it communicates with the primary node only.[4] Even so, replication isn't happening, and there's now no possibility of failover. The old primary must be restored. Assuming that the old primary was shut down cleanly, you can bring it back online, and it'll automatically rejoin the replica set as a secondary. Go ahead and try that now by restarting the old primary node.

That's a quick overview of replica sets. Some of the details are, unsurprisingly, messier. In the next two sections, you'll see how replica sets work and look at deployment, advanced configuration, and how to handle tricky scenarios that may arise in production.

To better see how this works, let's look more closely at a real oplog and at the operations recorded in it. First connect with the shell to the primary node started in the previous section and switch to the `local` database:

```
myapp:PRIMARY> use local
switched to db local
```

The `local` database stores all the replica set metadata and the oplog. Naturally, this database isn't replicated itself. Thus it lives up to its name; data in the `local` database is supposed to be unique to the local node and therefore shouldn't be replicated.

If you examine the `local` database, you'll see a collection called `oplog.rs`, which is where every replica set stores its oplog. You'll also see a few system collections. Here's the complete output:

```
myapp:PRIMARY> show collections
me
oplog.rs
replset.minvalid
slaves
startup_log
system.indexes
system.replset
```

`replset.minvalid` contains information for the initial sync of a given replica set member, and `system.replset` stores the replica set config document.

Not all of your mongod servers will have the `replset.minvalid` collection. `me` and `slaves` are used to implement write concerns, described at the end of this chapter, and `system.indexes` is the standard index spec container.

First we'll focus on the oplog. Let's query for the oplog entry corresponding to the book document you added in the previous section. To do so, enter the following query. The resulting document will have four fields, and we'll discuss each in turn:

```
> db.oplog.rs.findOne({op: "i"})
{
  "ts" : Timestamp(1383844267, 1),
  "h" : NumberLong("-3057344637426023 23"),
  "v" : 2,
  "op" : "i",
  "ns" : "bookstore.books",
  "o" : {
    "_id" : ObjectId("527bc9aac2595f18349e4154"),
    "title" : "Oliver Twist"
  }
}
```

5

The first field, `ts`, stores the entry's BSON timestamp. The timestamp includes two numbers; the first representing the seconds since epoch and the second representing a counter value—1 in this case. To query with a timestamp, you need to explicitly construct a timestamp object. All the drivers have their own BSON timestamp constructors, and so does JavaScript. Here's how to use it:

```
db.oplog.rs.findOne({ts: Timestamp(1383844267, 1)})
```

Returning to the oplog entry, the `op` field specifies the opcode. This tells the secondary node which operation the oplog entry represents. Here you see an `i`, indicating an insert. After `op` comes `ns` to signify the relevant namespace (database and collection) and then the lowercase letter `o`, which for insert operations contains a copy of the inserted document.

```
myapp:PRIMARY> use bookstore
myapp:PRIMARY> db.books.insert({title: "A Tale of Two Cities"})
myapp:PRIMARY> db.books.insert({title: "Great Expectations"})
```

Now with four books in the collection, let's issue a multi-update to set the author's name:

```
myapp:PRIMARY> db.books.update({}, {$set: {author: "Dickens {multi:true})
```

How does this appear in the oplog?

```
myapp:PRIMARY> use local
myapp:PRIMARY> db.oplog.rs.find({op: "u"})
{
  "ts" : Timestamp(1384128758, 1),
  "h" : NumberLong("5431582342821118204"),
  "v" : 2,
  "op" : "u",
  "ns" : "bookstore.books",
  "o2" : {
```

```
      "_id" : ObjectId("527bc9aac2595f18349e4154")
    },
    "o" : {
      "$set" : {
        "author" : "Dickens"
      }
    }
  }
  {
    "ts" : Timestamp(1384128758, 2),
    "h" : NumberLong("3897436474689294423"),
    "v" : 2,
    "op" : "u",
    "ns" : "bookstore.books",


  "o2" : {
    "_id" : ObjectId("528020a9f3f61863aba207e7")
  },
  "o" : {
    "$set" : {
      "author" : "Dickens"
    }
  }
}
{
  "ts" : Timestamp(1384128758, 3),
  "h" : NumberLong("2241781384783113"),
  "v" : 2,
  "op" : "u",
  "ns" : "bookstore.books",
  "o2" : {
    "_id" : ObjectId("528020a9f3f61863aba207e8")
  },
  "o" : {
    "$set" : {
      "author" : "Dickens"
    }
  }
}
```

As you can see, each updated document gets its own oplog entry. This normalization is done as part of the more general strategy of ensuring that secondaries always end up with the same data as the primary.

To guarantee this, every applied operation must be *idempotent*—it can't matter how many times a given oplog entry is applied. The result must always be the same. But the secondaries must apply the oplog entries in the same order as they were generated for the oplog. Other multidocument operations, like deletes, will exhibit the same behavior. You can try different operations and see how they ultimately appear in the oplog.

To get some basic information about the oplog's current status, you can run the shell's `db.getReplicationInfo()` method:

```
myapp:PRIMARY> db.getReplicationInfo()
```

```
  {
   "logSizeMB" : 192,
  "usedMB" : 0.01,
  "timeDiff" : 286197,
  "timeDiffHours" : 79.5,
  "tFirst" : "Thu Nov 07 2013 08:42:41 GMT-0800 (PST)",
  "tLast" : "Sun Nov 10 2013 16:12:38 GMT-0800 (PST)",
  "now" : "Sun Nov 10 2013 16:19:49 GMT-0800 (PST)"
}
```

Here you see the timestamps of the first and last entries in this oplog. You can find these oplog entries manually by using the `$natural` sort modifier. For example, the following query fetches the latest entry:

```
db.oplog.rs.find().sort({$natural: -1}) .limit(1)
```

---

want to avoid having to completely resync any node, and increasing the oplog size will buy you time in the event of network failures and the like.

If you want to change the default oplog size, you must do so the first time you start each member node using `mongod`'s `--oplogSize` option. The value is in megabytes. Thus you can start `mongod` with a 1 GB oplog like this:[7]

```
mongod --replSet myapp --oplogSize 1024
```

CONFIGURATION DETAILS

Here we'll present the `mongod` startup options pertaining to replica sets, and we'll describe the structure of the replica set configuration document.

Replication options

Earlier, you learned how to initiate a replica set using the shell's `rs.initiate()` and `rs.add()` methods. These methods are convenient, but they hide certain replica set configuration options. Let's look at how to use a configuration document to initiate and update a replica set's configuration.

A configuration document specifies the configuration of the replica set. To create one, first add a value for `_id` that matches the name you passed to the `--replSet` parameter:

```
> config = {_id: "myapp", members: []}
{ "_id" : "myapp", "members" : [ ] }
```

The individual `members` can be defined as part of the configuration document as follows:

```
config.members.push({_id: 0, host: 'iron.local:40000'})
config.members.push({_id: 1, host: 'iron.local:40001'})
config.members.push({_id: 2, host: 'iron.local:40002', arbiterOnly: true})
```

As noted earlier, `iron` is the name of our test machine; substitute your own hostname as necessary. Your configuration document should now look like this:

```
> config
{
```

```
  "_id" : "myapp",
  "members" : [
    {
      "_id" : 0,
      "host" : "iron.local:40000"
    },
    {
      "_id" : 1,
      "host" : "iron.local:40001"
    },
    {
      "_id" : 2,
      "host" : "iron.local:40002",
      "arbiterOnly" : true
    }
  ]
}
```

You can then pass the document as the first argument to `rs.initiate()` to initiate the replica set.

configuration parameters, plus the optional `arbiterOnly` setting. Please keep in mind that although a replica set can have up to 50 members, it can only have up to 7 voting members.

The document requires an `_id` that matches the replica set's name. The initiation command will verify that each member node has been started with the `--replSet` option with that name. Each replica set `member` requires an `_id` consisting of increasing integers starting from 0. Also, `members` require a `host` field with a hostname and optional port.

Here you initiate the replica set using the `rs.initiate()` method. This is a simple wrapper for the `replSetInitiate` command. Thus you could have started the replica set like this:

```
db.runCommand({replSetInitiate: config});
```

`config` is a variable holding your configuration document. Once initiated, each set member stores a copy of this configuration document in the `local` database's `system.replset` collection. If you query the collection, you'll see that the document now has a version number. Whenever you modify the replica set's configuration, you must also increment this version number. The easiest way to access the current configuration document is to run `rs.conf()`.

To modify a replica set's configuration, there's a separate command, `replSetReconfig`, which takes a new configuration document. Alternatively, you can use `rs.reconfig()` which also uses `replSetReconfig`. The new document can specify the addition or removal of set members along with alterations to both member-specific and global configuration options. The process of modifying a configuration document, incrementing the version number, and passing it as part of the `replSetReconfig` can be laborious, so a number of shell helpers exist to ease the way. To see a list of them

all, enter `rs.help()` at the shell.

Bear in mind that whenever a replica set reconfiguration results in the election of a new primary node, all client connections will be closed. This is done to ensure that clients will no longer attempt to send writes to a secondary node unless they're aware of the reconfiguration.

If you're interested in configuring a replica set from one of the drivers, you can see how by examining the implementation of `rs.add()`. Enter `rs.add` (the method without the parentheses) at the shell prompt to see how the method works.

Configuration document options

Until now, we've limited ourselves to the simplest replica set configuration document. But these documents support several options for both replica set members and for the replica set as a whole. We'll begin with the member options. You've seen `_id`, `host`, and `arbiterOnly`. Here are these plus the rest, in all their gritty detail:

- `_id` *(required)*—A unique incrementing integer representing the member's ID. These `_id` values begin at 0 and must be incremented by one for each member added.

- `ost` (required)—A string storing the hostname of this member along with an optional port number. If the port is provided, it should be separated from the hostname by a colon (for example, `iron:30000`). If no port number is specified, the default port, 27017, will be used. We've seen it before, but here's a simple document with a replica set `_id` and `host`:

```
{
  "_id" : 0,
  "host" : "iron:40000"
}
```

- `arbiterOnly`—A Boolean value, `true` or `false`, indicating whether this member is an arbiter. Arbiters store configuration data only. They're lightweight members that participate in primary election but not in the replication itself. Here's an example of using the `arbiterOnly` setting:

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "arbiterOnly": true
}
```

- `priority`—A decimal number from 0 to 1000 that helps to determine the relative eligibility that this node will be elected primary. For both replica set initiation and failover, the set will attempt to elect as primary the node with the highest priority, as long as it's up to date. This might be useful if you have a replica set where some nodes are more powerful than the others; it makes sense to prefer the biggest machine as the primary.

  There are also cases where you might want a node never to be primary (say, a disaster recovery node residing in a secondary data center). In those cases, set

the priority to 0. Nodes with a priority of 0 will be marked as passive in the results to the `isMaster()` command and will never be elected primary. Here's an example of setting the member's priority:

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "priority" : 500
}
```

- `votes`—All replica set members get one vote by default. The `votes` setting allows you to give more than one vote to an individual member.

  This option should be used with extreme care, if at all. For one thing, it's difficult to reason about replica set failover behavior when not all members have the same number of votes. Moreover, the vast majority of production deployments will be perfectly well served with one vote per member. If you do choose to alter the number of votes for a given member, be sure to think through and

simulate the various failure scenarios carefully. This member has an increased number of votes:

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "votes" : 2
}
```

- `hidden`—A Boolean value that, when `true`, will keep this member from showing up in the responses generated by the `isMaster` command. Because the MongoDB drivers rely on `isMaster` for knowledge of the replica set topology, hiding a member keeps the drivers from automatically accessing it. This setting can be used in conjunction with `buildIndexes` and must be used with `slaveDelay`. This member is configured to be hidden:

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "hidden" : true
}
```

- `buildIndexes`—A Boolean value, defaulting to `true`, that determines whether this member will build indexes. You'll want to set this value to `false` only on members that will never become primary (those with a priority of 0).

  This option was designed for nodes used solely as backups. If backing up indexes is important, you shouldn't use this option. Here's a member configured not to build indexes:

```
{
  "_id" : 0,
```

```
  "host" : "iron:40000",
  "buildIndexes" : false
}
```

- `slaveDelay`—The number of seconds that a given secondary should lag behind the primary. This option can be used only with nodes that will never become primary. To specify a `slaveDelay` greater than 0, be sure to also set a priority of 0.

  You can use a delayed slave as insurance against certain kinds of user errors. For example, if you have a secondary delayed by 30 minutes and an administrator accidentally drops a database, you have 30 minutes to react to this event before it's propagated. This member has been configured with a `slaveDelay` of one hour:

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "slaveDelay" : 3600
}
```

- `tags`—A document containing a set of key-value pairs, usually used to identify this member's location in a particular datacenter or server rack. Tags are used for specifying granular write concern and read settings, and they're discussed in section 11.3.4. In the tag document, the values entered must be strings. Here's a member with two tags:

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "tags" : {
    "datacenter" : "NY",
    "rack" : "B"
}
```

That sums up the options for individual replica set members. There are also two global replica set configuration parameters scoped under a `settings` key. In the replica set configuration document, they appear like this:

```
{
  _id: "myapp",
  members: [ ... ],
  settings: {
    getLastErrorDefaults: {
      w: 1
    },
    getLastErrorModes: {
      multiDC: {
        dc: 2
      }
    }
  }
}
```

- `getLastErrorDefaults`—A document specifying the default arguments to be

used when the client calls `getLastError` with no arguments. This option should be treated with care because it's also possible to set global defaults for `getLastError` within the drivers, and you can imagine a situation where application developers call `getLastError` not realizing that an administrator has specified a default on the server.

For more details on `getLastError`, see its documentation at http://docs.mongodb.org/manual/reference/command/getLastError. Briefly, to specify that all writes are replicated to at least two members with a timeout of 500 ms, you'd specify this value in the config like this:

```
settings: {
  getLastErrorDefaults: {
    w: 2,
    wtimeout: 500
  }
}
```

- `etLastErrorModes`—A document defining extra modes for the `getLastError` command. This feature is dependent on replica set tagging and is described in detail in section 11.3.4.

Table 11.1 Replica set states

State	State string	Notes
0	STARTUP	Indicates that the replica set is negotiating with other nodes by pinging all set members and sharing config data.
1	PRIMARY	This is the primary node. A replica set will always have at most one primary node.
2	SECONDARY	This is a secondary, read-only node. This node may become a primary in the event of a failover if and only if its priority is greater than 0 and it's not marked as hidden.
3	RECOVERING	This node is unavailable for reading and writing. You usually see this state after a failover or upon adding a new node. While recovering, a data file sync is often in progress; you can verify this by examining the recovering node's logs.
4	FATAL	A network connection is still established, but the node isn't responding to pings. This usually indicates a fatal error on the machine hosting the node marked FATAL.
5	STARTUP2	An initial data file sync is in progress.
6	UNKNOWN	A network connection has yet to be made.
7	ARBITER	This node is an arbiter.
8	DOWN	The node was accessible and stable at some point but isn't currently responding to heartbeat pings.
9	ROLLBACK	A rollback is in progress.
10	REMOVED	The node was once a member of the replica set but has since been removed.

FAILOVER AND RECOVERY

In the sample replica set you saw a couple examples of failover. Here we summarize the rules of failover and provide some suggestions on handling recovery.

A replica set will come online when all members specified in the configuration can communicate with one another. Each node is given one vote by default, and those votes are used to form a majority and elect a primary. This means that a replica set can be started with as few as two nodes (and votes). But the initial number of votes also decides what constitutes a majority in the event of a failure.

Let's assume that you've configured a replica set of three complete replicas (no arbiters) and thus have the recommended minimum for automated failover. If the primary fails, and the remaining secondaries can see each other, then a new primary can be elected. As for deciding which one, the secondary with the most up-to-date oplog with the higher priority will be elected primary.

Failure modes and recovery

Recovery is the process of restoring the replica set to its original state following a failure. There are two overarching failure categories to be handled. The first is called *clean failure*, where a given node's data files can still be assumed to be intact. One example of this is a network partition. If a node loses its connections to the rest of the set, you need only wait for connectivity to be restored, and the partitioned node will resume as a set member. A similar situation occurs when a given node's `mongod` process is terminated for any reason but can be brought back online cleanly.[9] Again, once the process is restarted, it can rejoin the set.

The second type is called *categorical failure*, where a node's data files either no longer exist or must be presumed corrupted. Unclean shutdowns of the `mongod` process without journaling enabled and hard drive crashes are both examples of this kind of failure. The only ways to recover a categorically failed node are to completely replace the data files via a resync or to restore from a recent backup. Let's look at both strategies in turn.

To completely resync, start a `mongod` with an empty data directory on the failed node. As long as the host and port haven't changed, the new `mongod` will rejoin the replica set and then resync all the existing data. If either the host or port has changed, then after bringing the `mongod` back online you'll also have to reconfigure the replica set. As an example, suppose the node at `iron:40001` is rendered unrecoverable and you bring up a new node at `foobar:40000`. You can reconfigure the replica set by grabbing the configuration document, modifying the host for the second node, and then passing that to the `rs.reconfig()` method:

```
> config = rs.conf()
{
  "_id" : "myapp",
  "version" : 1,
  "members" : [
```

```
    {
      "_id" : 0,
      "host" : "iron:40000"
    },
    {
      "_id" : 1,
      "host" : "iron:40001"
    },
    {
      "_id" : 2,
      "host" : "iron:40002",
      "arbiterOnly" : true
    }
  ]
}

> config.members[1].host = "foobar:40000"
foobar:40000
> rs.reconfig(config)
```

Drivers and replication

If you're building an application using MongoDB's replication, you need to know about several application-specific topics. The first is related to connections and failover. Next comes the write concern, which allows you to decide to what degree a given write should be replicated before the application continues. The next topic, read scaling, allows an application to distribute reads across replicas. Finally, we'll discuss tagging, a way to configure more complex replica set reads and writes.

Connections and failover

The MongoDB drivers present a relatively uniform interface for connecting to replica sets.

SINGLE-NODE CONNECTIONS

You'll always have the option of connecting to a single node in a replica set. There's no difference between connecting to a node designated as a replica set primary and connecting to one of the vanilla stand-alone nodes we've used for the examples throughout the book. In both cases, the driver will initiate a TCP socket connection and then run the `isMaster` command. For a stand-alone node, this command returns a document like the following:

```
{
  "ismaster" : true,
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
  "localTime" : ISODate("2013-11-12T05:22:54.317Z"),
  "ok" : 1
}
```

What's most important to the driver is that the `isMaster` field be set to `true`, which indicates that the given node is a stand-alone, a master running master-slave replica-

tion, or a replica set primary.[11] In all of these cases, the node can be written to, and the user of the driver can perform any CRUD operation.

But when connecting directly to a replica set secondary, you must indicate that you know you're connecting to such a node (for most drivers, at least). In the Ruby driver, you accomplish this with the `:read` parameter. To connect directly to the first secondary you created earlier in the chapter, the Ruby code would look like this:

```
@con = Mongo::Client.new(['iron: 40001'], {:read => {:mode => :secondary}})
```

REPLICA SET CONNECTIONS

You can connect to any replica set member individually, but you'll normally want to connect to the replica set as a whole. This allows the driver to figure out which node is primary and, in the case of failover, reconnect to whichever node becomes the new primary.

Most of the officially supported drivers provide ways of connecting to a replica set. In Ruby, you connect by creating a new instance of `Mongo::Client`, passing in a list of seed nodes as well as the name of the replica set:

```
Mongo::Client.new(['iron:40000', 'iron:40001'], :replica_set => 'myapp')
```

Internally, the driver will attempt to connect to each seed node and then call the `isMaster` command. Issuing this command to a replica set returns a number of important set details:

```
> db.isMaster()
{
  "setName" : "myapp",
  "ismaster" : false,
  "secondary" : true,
  "hosts" : [
    "iron:40001",
    "iron:40000"
  ],
  "arbiters" : [
    "iron:40002"
  ],
  "me" : "iron:40000",
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
  "localTime" : ISODate("2013-11-12T05:14:42.009Z"),
  "ok" : 1
}
```

Read scaling

Replicated databases are great for read scaling. If a single server can't handle the application's read load, you have the option to route queries to more than one replica. Most of the drivers have built-in support for sending queries to secondary nodes through a read preference configuration. With the Ruby driver, this is provided as an

option on the `Mongo::Client` constructor:

```
Mongo::Client.new(
    ['iron:40000', 'iron:40001'],
    {:read => {:mode => :secondary})
```

Note in the connection code that we configure which nodes the new client will read from. When the `:read` argument is set to {:mode => `:secondary}`, the connection object will choose a random, nearby secondary to read from. This configuration is called the read preference, and it can be used to direct your driver to read from certain nodes. Most MongoDB drivers have these available read preferences:

- *primary*—This is the default setting and indicates that reads will always be from the replica set primary and thus will always be consistent. If the replica set is experiencing problems and there's no secondary available, an error will be thrown.
- *primaryPreferred*—Drivers with this setting will read from the primary unless for some reason it's unavailable or there's no primary, in which case reads will go to a secondary. This means that reads aren't guaranteed to be consistent.
- *secondary*—This setting indicates the driver should always read from the secondary. This is useful in cases where you want to be sure that your reads will have no impact on the writes that occur on the primary. If no secondaries are available, the read will throw an exception.
- *secondaryPreferred*—This is a more relaxed version of the previous setting. Reads will go to secondaries, unless no secondaries are available, in which case reads will go to the primary.
- *nearest*—A driver configured with this setting will attempt to read from the nearest member of the replica set, as measured by network latency. This could be either a primary or a secondary. Thus, reads will go to the member that the driver believes it can communicate with the quickest.

For the Ruby driver, this configuration might look like this:

```
Mongo::Client.new(
    ['iron:40000', 'iron:40001'],
    :read => {:mode => :secondary},:local_threshold => '0.0015')
```

The `:local_threshold` option specifies the maximum latency in seconds as a float.

Tagging

If you're using either write concerns or read scaling, you may find yourself wanting more granular control over exactly which secondaries receive writes or reads. For example, suppose you've deployed a five-node replica set across two data geographically separate centers, *NY* and *FR*. The primary datacenter, NY, contains three nodes, and the secondary datacenter, FR, contains the remaining two. Let's say that you want to use a write concern to block until a certain write has been replicated to at least one

Here's an example:

```
{
  "_id" : "myapp",
  "version" : 1,
  "members" : [
    {
      "_id" : 0,
      "host" : "ny1.myapp.com:30000",
      "tags": { "dc": "NY", "rackNY": "A" }
    },
    {
      "_id" : 1,
      "host" : "ny2.myapp.com:30000",
      "tags": { "dc": "NY", "rackNY": "A" }
    },
    {
      "_id" : 2,
      "host" : "ny3.myapp.com:30000",
      "tags": { "dc": "NY", "rackNY": "B" }
    },
    {
      "_id" : 3,
      "host" : "fr1.myapp.com:30000",
      "tags": { "dc": "FR", "rackFR": "A" }
    },
```

```
    {
      "_id" : 4,
      "host" : "fr2.myapp.com:30000",
      "tags": { "dc": "FR", "rackFR": "B" }
    }
  ],
  settings: {
    getLastErrorModes: {
      multiDC: { dc: 2 } },
      multiRack: { rackNY: 2 } },
    }
  }
}
```

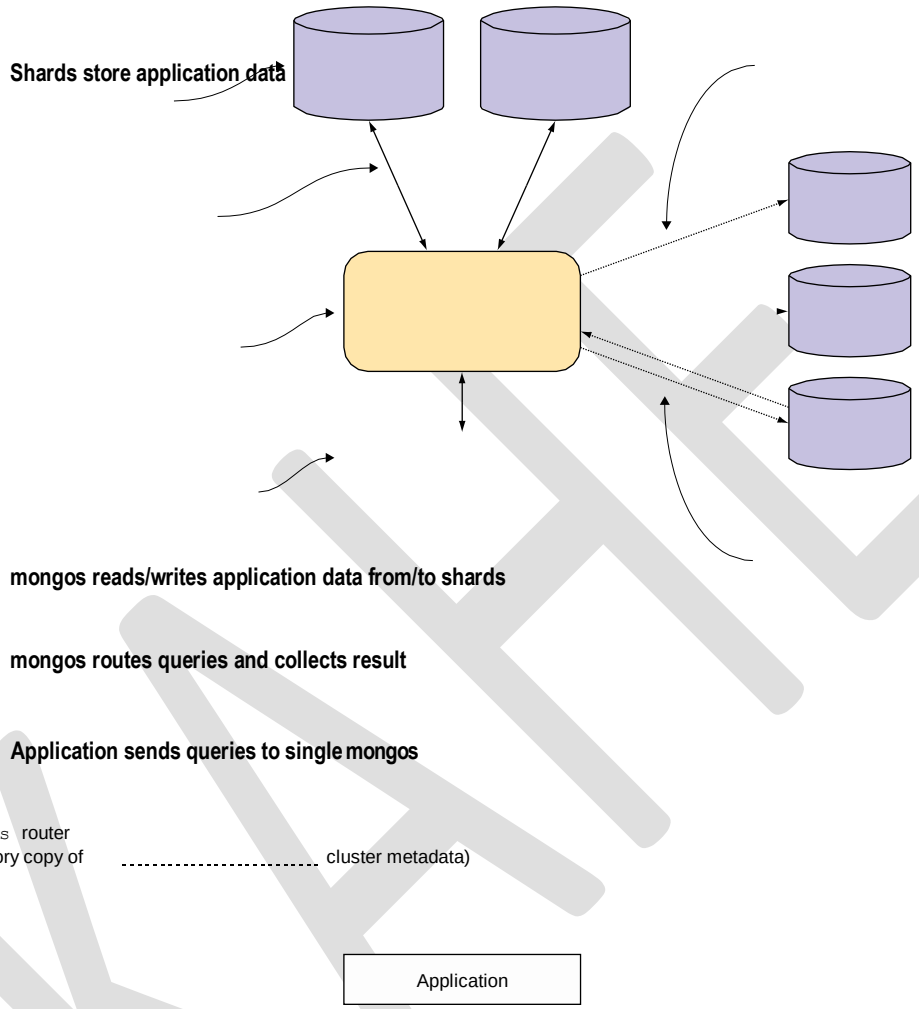


Figure 12.1 Components in a MongoDB shard cluster

A sharded cluster consists of shards, `mongos` routers, and config servers, as shown in figure 12.1.

Let's examine each component in figure 12.1:

- *Shards* (upper left) store the application data. In a sharded cluster, only the `mongos` routers or system administrators should be connecting directly to the shards. Like an unsharded deployment, each shard can be a single node for development and testing, but should be a replica set in production.

- `mongos` *routers* (center) cache the cluster metadata and use it to route operations to the correct shard or shards.
- *Config servers* (upper right) persistently store metadata about the cluster, including which shard has what subset of the data.

Now, let's discuss in more detail the role each of these components plays in the cluster as a whole.

replica set, but if you try to run operations on that shard directly, you'll see only a portion of the cluster's total data.

Mongos router: router of operations

Because each shard contains only part of the cluster's data, you need something to route operations to the appropriate shards. That's where `mongos` comes in. The `mongos` process, shown in the center of figure 12.1, is a router that directs all reads, writes, and commands to the appropriate shard. In this way, `mongos` provides clients with a single point of contact with the cluster, which is what enables a sharded cluster to present the same interface as an unsharded one.

`mongos` processes are lightweight and nonpersistent.[1] Because of this, they're often deployed on the same machines as the application servers, ensuring that only one network hop is required for requests to any given shard. In other words, the application connects locally to a `mongos`, and the `mongos` manages connections to the individual shards.

f a predetermined field or set of fields called a *shard key.* It's the user's responsibility to choose the shard key, and we'll cover how to do this in section 12.8.

For example, consider the following document from a spreadsheet management application:

```
{
  _id: ObjectId("4d6e9b89b600c2c196442c21")
  filename: "spreadsheet-1",
  updated_at: ISODate("2011-03-02T19:22:54.845Z"),
  username: "banks",
  data: "raw document data"
}
```

If all the documents in our collection have this format, we can, for example, choose a shard key of the `_id` field and the `username` field. MongoDB will then use that information in each document to determine what *chunk* the document belongs to.

How does MongoDB make this determination? At its core, MongoDB's sharding is *range-based;* this means that each "chunk" represents a range of shard keys. When MongoDB looks at a document to determine what chunk it belongs to, it first extracts the values for the shard key and then finds the chunk whose shard key range contains the given shard key values.

To see a concrete example of what this looks like, imagine that we chose a shard key of `username` for this `spreadsheets` collection, and we have two shards, "A" and "B." Our chunk distribution may look something like table 12.1.

Table 12.1 Chunks and shards

Start	End	Shard
-∞	Abbot	B
Abbot	Dayton	A
Dayton	Harris	B
Harris	Norris	A
Norris	∞	B

Looking at the table, it becomes a bit clearer what purpose chunks serve in a sharded cluster. If we gave you a document with a `username` field of `"Babbage"`, you'd immediately know that it should be on shard A, just by looking at the table above. In fact, if we gave you any document that had a `username` field, which in this case is our shard key, you'd be able to use table 12.1 to determine which chunk the document belonged to, and from there determine which shard it should be sent to.

Building a sample shard cluster

The best way to get a handle on sharding is to see how it works in action. Fortunately, it's possible to set up a sharded cluster on a single machine, and that's exactly what we'll do now.[3]

The full process of setting up a sharded cluster involves three phases:

1. *Starting the* `mongod` *and* `mongos` *servers*—The first step is to spawn all the individual `mongod` and `mongos` processes that make up the cluster. In the cluster we're setting up in this chapter, we'll spawn nine `mongod` servers and one `mongos` server.

2 *Configuring the cluster*—The next step is to update the configuration so that the replica sets are initialized and the shards are added to the cluster. After this, the nodes will all be able to communicate with each other.

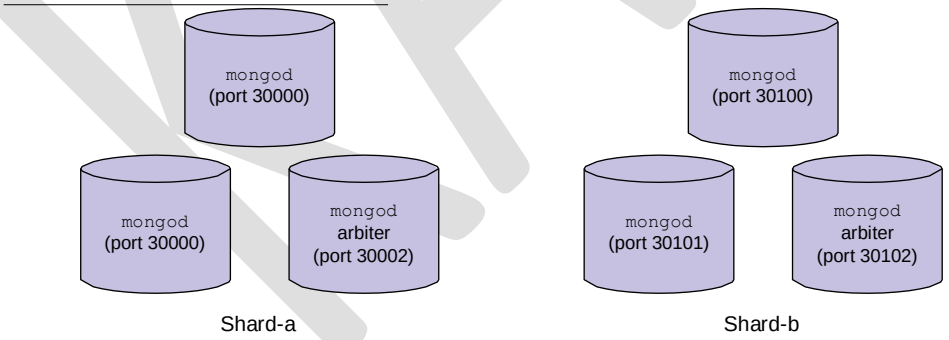
3 *Sharding collections*—The last step is to shard a collection so that it can be spread across multiple shards. The reason this exists as a separate step is because MongoDB can have both sharded and unsharded collections in the same cluster, so you must explicitly tell it which ones you want to shard. In this chapter, we'll shard our only collection, which is the `spreadsheets` collection of the `cloud-docs` database.

We'll cover each of these steps in detail in the next three sections. We'll then simulate the behavior of the sample cloud-based spreadsheet application described in the previous sections. Throughout the chapter we'll examine the global shard configuration, and in the last section, we'll use this to see how data is partitioned based on the shard key.

Starting the mongod and mongos servers

The first step in setting up a sharded cluster is to start all the required `mongod` and `mongos` processes. The shard cluster you'll build will consist of two shards and three config servers. You'll also start a single `mongos` to communicate with the cluster. Figure 12.3 shows a map of all the processes that you'll launch, with their port numbers in parentheses.

You'll runa bunch of commands to bring the cluster online, so if you find yourself unable to see the forest because of the trees, refer back to this figure.



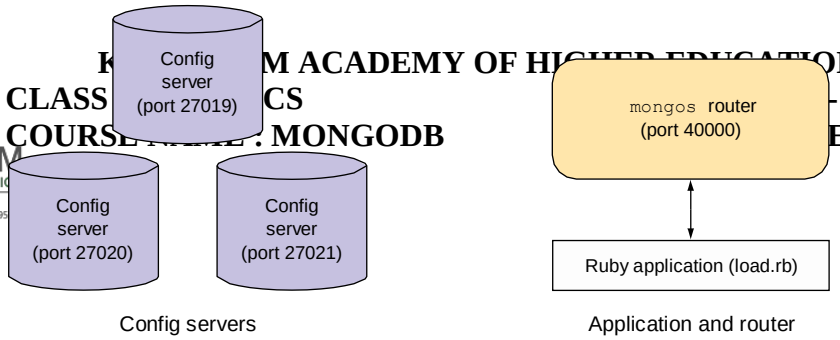


Figure 12.3 A map of processes comprising the sample shard cluster

STARTING THE SHARDING COMPONENTS

Let's start by creating the data directories for the two replica sets that will serve as our shards:

```
$ mkdir /data/rs-a-1
$ mkdir /data/rs-a-2
$ mkdir /data/rs-a-3
$ mkdir /data/rs-b-1
$ mkdir /data/rs-b-2
$ mkdir /data/rs-b-3
```

Next, start each `mongod`. Because you're running so many processes, you'll use the `--fork` option to run them in the background.[4] The commands for starting the first replica set are as follows:

```
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-1 \
  --port 30000 --logpath /data/rs-a-1.log --fork
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-2 \
  --port 30001 --logpath /data/rs-a-2.log --fork
```

```
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-3 \
  --port 30002 --logpath /data/rs-a-3.log --fork
```

Here are the commands for the second replica set:

```
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-1 \
  --port 30100 --logpath /data/rs-b-1.log --fork
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-2 \
  --port 30101 --logpath /data/rs-b-2.log --fork
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-3 \
  --port 30102 --logpath /data/rs-b-3.log --fork
```

s usual, you now need to initiate these replica sets. Connect to each one individually, run `rs.initiate()`, and then add the remaining nodes. The first should look like this:

```
$ mongo localhost:30000
> rs.initiate()
```

You'll have to wait a minute or so before the initial node becomes primary. During the process, the prompt will change from `shard-a:SECONDARY>` to `shard-a:PRIMARY`. Using the `rs.status()` command will also reveal more information about what's going on behind the scenes. Once it does, you can add the remaining nodes:

```
> rs.add("localhost:30001")
> rs.addArb("localhost:30002")
```

`localhost` as the machine name might cause problems in the long run because it only works if you're going to run all processes on a single machine. If you know your hostname, use it to get out of trouble. On a Mac, your hostname should look something like `MacBook-Pro.local`. If you don't know your hostname, make sure that you use `localhost` everywhere!

Configuring a replica set that you'll use as a shard is exactly the same as configuring a replica set that you'll use on its own, so refer back to chapter 10 if any of this replica set setup looks unfamiliar to you.

Initiating the second replica set is similar. Again, wait a minute after running `rs.initiate()`:

```
$ mongo localhost:30100
> rs.initiate()
> rs.add("localhost:30100")
> rs.addArb("localhost:30101")
```

Finally, verify that both replica sets are online by running the `rs.status()` command from the shell on each one. If everything checks out, you're ready to start the config

servers.[5] Now you create each config server's data directory and then start a `mongod` for each one using the `configsvr` option:

```
$ mkdir /data/config-1
$ mongod --configsvr --dbpath /data/config-1 --port 27019 \
  --logpath /data/config-1.log --fork --nojournal
$ mkdir /data/config-2
$ mongod --configsvr --dbpath /data/config-2 --port 27020 \
  --logpath /data/config-2.log --fork --nojournal
$ mkdir /data/config-3
$ mongod --configsvr --dbpath /data/config-3 --port 27021 \
  --logpath /data/config-3.log --fork --nojournal
```

Ensure that each config server is up and running by connecting with the shell, or by tailing the log file (`tail -f <log_file_path>`) and verifying that each process is listening on the configured port. Looking at the logs for any one config server, you should see something like this:

```
Wed Mar 2 15:43:28 [initandlisten] waiting for connections on port 27020
Wed Mar 2 15:43:28 [websvr] web admin interface listening on port 28020
```

If each config server is running, you can go ahead and start the `mongos`. The `mongos` must be started with the `configdb` option, which takes a comma-separated list of config database addresses:[6]

```
$ mongos --configdb localhost:27019,localhost:27020,localhost:27021 \
  --logpath /data/mongos.log --fork --port 40000
```

Configuring the cluster

Now that you've started all the `mongod` and `mongos` processes that we'll need for this cluster (see figure 12.2), it's time to configure the cluster. Start by connecting to the `mongos`. To simplify the task, you'll use the sharding helper methods. These are methods run on the global `sh` object. To see a list of all available helper methods, run `sh.help()`.

You'll enter a series of configuration commands beginning with the `addShard` command. The helper for this command is `sh.addShard()`. This method takes a string consisting of the name of a replica set, followed by the addresses of two or more seed

---

nodes for connecting. Here you specify the two replica sets you created along with the addresses of the two non-arbiter members of each set:

```
$ mongo localhost:40000
> sh.addShard("shard-a/localhost:30000,localhost:30001")
  { "shardAdded" : "shard-a", "ok" : 1 }
> sh.addShard("shard-b/localhost:30100,localhost:30101")
  { "shardAdded" : "shard-b", "ok" : 1 }
```

If successful, the command response will include the name of the shard just added. You can examine the config database's `shards` collection to see the effect of your work. Instead of using the `use` command, you'll use the `getSiblingDB()` method to switch databases:

```
> db.getSiblingDB("config").shards.find()
{ "_id" : "shard-a", "host" : "shard-a/localhost:30000,localhost:30001" }
{ "_id" : "shard-b", "host" : "shard-b/localhost:30100,localhost:30101" }
```

As a shortcut, the `listshards` command returns the same information:

```
> use admin
> db.runCommand({listshards: 1})
```

While we're on the topic of reporting on sharding configuration, the shell's `sh.status()` method nicely summarizes the cluster. Go ahead and try running it now.

Sharding collections

The next configuration step is to enable sharding on a database. This doesn't do anything on its own, but it's a prerequisite for sharding any collection within a database. Your application's database will be called `cloud-docs`, so you enable sharding like this:

```
> sh.enableSharding("cloud-docs")
```

As before, you can check the config data to see the change you just made. The config database holds a collection called `databases` that contains a list of databases. Each document specifies the database's primary shard location and whether it's partitioned (whether sharding is enabled):

```
> db.getSiblingDB("config").databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "cloud-docs", "partitioned" : true, "primary" : "shard-a" }
```

Now all you need to do is shard the `spreadsheets` collection. When you shard a collection, you define a shard key. Here you'll use the compound shard key `{username: 1, _id: 1}` because it's good for distributing data and makes it easy to view and comprehend chunk ranges:

```
> sh.shardCollection("cloud-docs.spreadsheets", {username: 1, _id: 1})
```

Again, you can verify the configuration by checking the config database for sharded collections:

```
> db.getSiblingDB("config").collections.findOne()
{
  "_id" : "cloud-docs.spreadsheets",
  "lastmod" : ISODate("1970-01-16T00:50:07.268Z"),
  "dropped" : false,
  "key" : {
    "username" : 1,
    "_id" : 1
  },
  "unique" : false
}
```

Don't worry too much about understanding all the fields in this document. This is internal metadata that MongoDB uses to track collections, and it isn't meant to be accessed directly by users.

SHARDING AN EMPTY COLLECTION

This sharded collection definition may remind you of something: it looks a bit like an index definition, especially with its `unique` key. When you shard an empty collection, MongoDB creates an index corresponding to the shard key on each shard.[7] Verify this for yourself by connecting directly to a shard and running the `getIndexes()` method.

Here you connect to your first shard, and the output contains the shard key index, as expected:

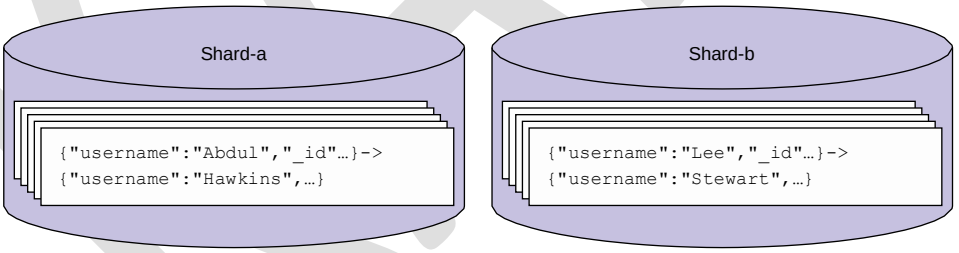
```
$ mongo localhost:30000
> use cloud-docs
> db.spreadsheets.getIndexes()
[

  {
    "name" : "_id_",
    "ns" : "cloud-docs.spreadsheets", "key" : {
                                    "_id" : 1
```

SEEING DATA ON MULTIPLE SHARDS

The picture has definitely changed. As you can see in figure 12.4, you now have 10 chunks. Naturally, each chunk represents a contiguous range of data.

You can see in figure 12.4 that shard-a has a chunk that ranges from one of Abdul's documents to one of Buettner's documents, just as you saw in our output. This means that all the documents with a shard key that lies between these two values will either be inserted into, or found on, shard-a.[8] You can also see in the figure that shard-b has



Now the split threshold will increase. You can see how the splitting slows down, and how chunks start to grow toward their max size, by doing a more massive insert. Try adding another 800 MB to the cluster. Once again, we'll use the Ruby script, remem- bering that it inserts about 1 MB on each iteration:

```
$ ruby load.rb 800
```

This will take a lot of time to run, so you may want to step away and grab a snack after starting this load process. By the time it's done, you'll have increased the total data

size by a factor of 8. But if you check the chunking status, you'll see that there are only around twice as many chunks:

```
> use config
> db.chunks.count()
21
```

Given that there are more chunks, the average chunk ranges will be smaller, but each chunk will include more data. For example, the first chunk in the collection spans from Abbott to Bender but it's already nearly 60 MB in size. Because the max chunk size is currently 64 MB by default, you'd soon see this chunk split if you were to continue inserting data.

Another thing to note is that the distribution still looks pretty even, as it did before:

```
> db.chunks.count({"shard": "shard-a"})
11
> db.chunks.count({"shard": "shard-b"})
10
```

Although the number of chunks has increased during the last 800 MB insert round, you can probably assume that no migrations occurred; a likely scenario is that each of the original chunks split in two, with a single extra split somewhere in the mix. You can verify this by querying the config database's `changelog` collection:

```
> db.changelog.count({what: "split"})
20
> db.changelog.find({what: "moveChunk.commit"}).count()
6
```

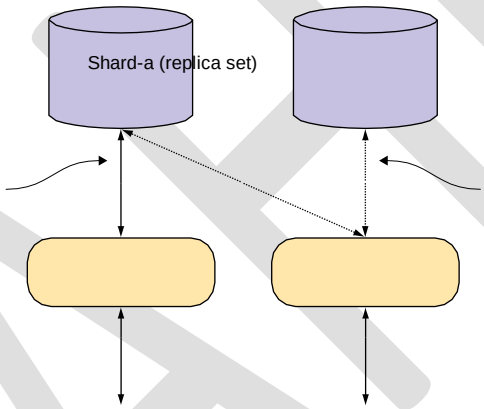
This is in line with these assumptions. A total of 20 splits have occurred, yielding 20 chunks, but only 6 migrations have taken place. For an extra-deep look at what's going on here, you can scan the changelog entries. For instance, here's the entry recording the first chunk move:

```
> db.changelog.findOne({what: "moveChunk.commit"})
{
  "_id" : "localhost-2011-09-01T20:40:59-2",
  "server" : "localhost",
  "clientAddr" : "127.0.0.1:55749",
  "time" : ISODate("2011-03-01T20:40:59.035Z"),
  "what" : "moveChunk.commit",
  "ns" : "cloud-docs.spreadsheets",
```

```
                    "details" : {
"min" : {
  "username" : { $minKey : 1 },
  "_id" : { $minKey : 1 }
},

"max" : {
  "username" : "Abbott",
  "_id" : ObjectId("4d6d57f61d41c851ee000092")
},
"from" : "shard-a",


"to" : "shard-b"
                                    }
                                  }
```



Queries only the shard with the chunk containing documents with "Abbott" as the shard key

```
find({username:"Abbott"})                    find({filename:"sheet-1"})
```

in an unsharded deployment, indexing is an important part of optimizing performance. There are only a few key points to keep in mind about indexing that are specific to a sharded cluster:

- Each shard maintains its own indexes. When you declare an index on a sharded collection, each shard builds a separate index for its portion of the collection. For example, when you issue the `db.spreadsheets.createIndex()` command while connected to a `mongos` router, each shard processes the index creation command individually.

- It follows that the sharded collections on each shard should have the same indexes. If this ever isn't the case, you'll see inconsistent query performance.
- Sharded collections permit unique indexes on the `_id` field and on the shard key only. Unique indexes are prohibited elsewhere because enforcing them would require inter-shard communication, which is against the fundamental design of sharding in MongoDB.

Once you understand how queries are routed and how indexing works, you should be in a good position to write smart queries and indexes for your sharded cluster. Most of the advice on indexing and query optimization from chapter 8 will apply.

In the next section, we'll cover the powerful `explain()` tool, which you can use to see exactly what path is taken by a query against your cluster.

The explain() tool in a sharded cluster

The `explain()` tool is your primary way to troubleshoot and optimize queries. It can show you exactly how your query would be executed, including whether it can be targeted and whether it can use an index. The following listing shows an example of what this output might look like.

Listing 12.1 Index and query to return latest documents updated by a user

```
mongos> db.spreadsheets.createIndex({username:1, updated_at:-1})
{
    "raw" : {
        "shard-a/localhost:30000,localhost:30001" : {
            "createdCollectionAutomatically" : false,
            "numIndexesBefore" : 3,
            "numIndexesAfter" : 4,
            "ok" : 1
        },
        "shard-b/localhost:30100,localhost:30101" : {
            "createdCollectionAutomatically" : false,
            "numIndexesBefore" : 3,
            "numIndexesAfter" : 4,
            "ok" : 1
        }
    },
    "ok" : 1
}

mongos> db.spreadsheets.find({username: "Wallace"}).sort({updated_at:-
    1}).explain()
{
    "clusteredType" : "ParallelSort",
    "shards" : {
        "shard-b/localhost:30100,localhost:30101" : [
            {
```

```
                "cursor" : "BtreeCursor username_1_updated_at_-1", "isMultiKey" : false,
                           "n" : 100,
                "nscannedObjects" : 100,
                "nscanned" : 100,
                "nscannedObjectsAllPlans" : 200,
                "nscannedAllPlans" : 200, "scanAndOrder" : false, "indexOnly" :
                false, "nYields" : 1,
                "nChunkSkips" : 0,
                "millis" : 3, "indexBounds" : {
                    "username" : [ [
                            "Wallace",
                            "Wallace"
                        ]
                    ],
                    "updated_at" : [
                        [
                            {
                                "$maxElement" : 1
                            },
                            {
                                "$minElement" : 1
                            }
                        ]
                    ]
                },
                "server" : "localhost:30100",
                "filterSet" : false
            }
        ]
    },
    "cursor" : "BtreeCursor username_1_updated_at_-1",
    "n" : 100,
    "nChunkSkips" : 0,
    "nYields" : 1,
    "nscanned" : 100,
    "nscannedAllPlans" : 200,
    "nscannedObjects" : 100,
    "nscannedObjectsAllPlans" : 200,

    "millisShardTotal" : 3,
    "millisShardAvg" : 3,
    "numQueries" : 1,
    "numShards" : 1, "indexBounds" : {
```

UNIT-V
SYLLABUS

Deployment and administration: Deployment – Monitoring and diagnostics – Maintenance – Performance troubleshooting

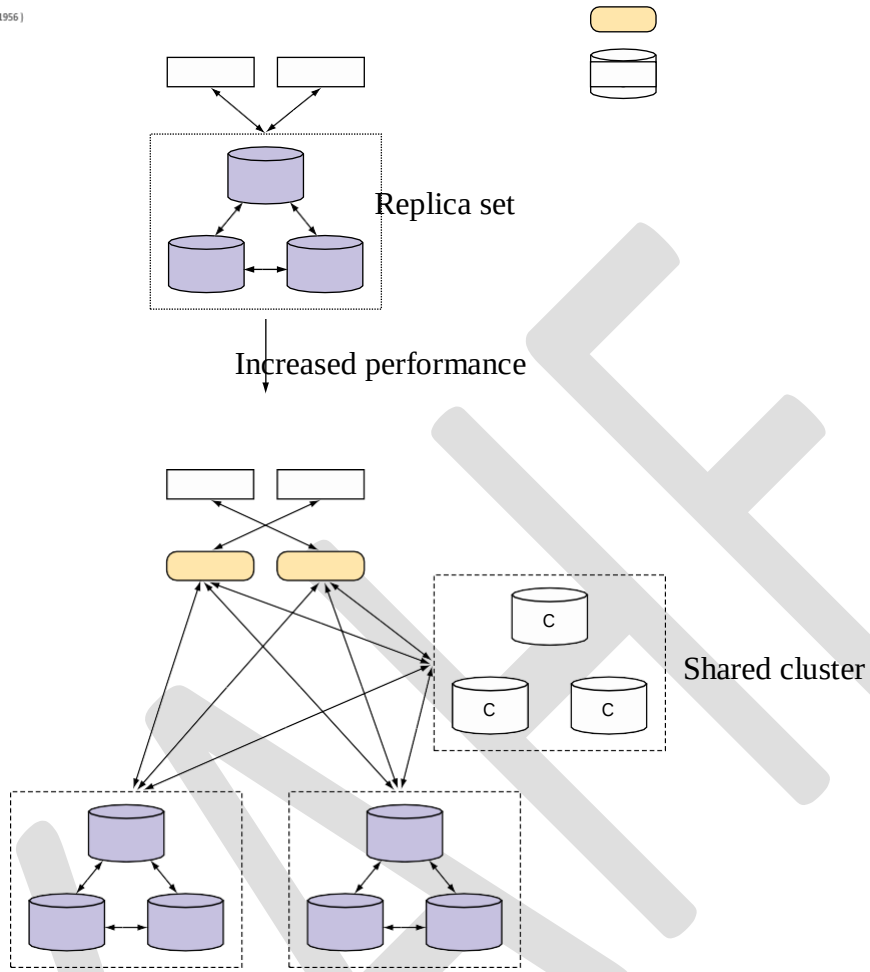
Cluster topology

In total, there are three different types of clusters in MongoDB:

- *Single node*—As you can see at the top of figure 13.1, MongoDB can be run as a single server to support testing and staging environments. But for production deployments, a single server isn't recommended, even if journaling is enabled. Having only one machine complicates backup and recovery, and when there's a server failure, there's nothing to fail over to. That said, if you don't need reliability and have a small enough data set, this is always an option.
- *Replica set*—As shown in the middle of figure 13.1, the minimum recommended deployment topology for a replica set is three nodes, at least two of which should be data-storing nodes rather than arbiters. A replica set is necessary for automatic failover, easier backups, and not having a single point of failure. Refer to chapter 10 for more details on replica sets.

- *Sharded cluster*—As you can see at the bottom of figure 13.1, the minimum recommended deployment for a sharded cluster has two shards because deploying a sharded cluster with only one shard would add additional over- head without any of the benefits of sharding. Each shard should also be a replica set and there should be three config servers to ensure that there's no single point of failure. Note that there are also two mongos processes. Loss of all mongos processes doesn't lead to any data loss, but it does lead to down- time, so we have two here as part of the minimum production topology to ensure high availability.

- A sharded cluster is necessary when you want to scale up the capacity of your cluster by pooling together the capacity of a number of less powerful commodity servers.



Deployment environment

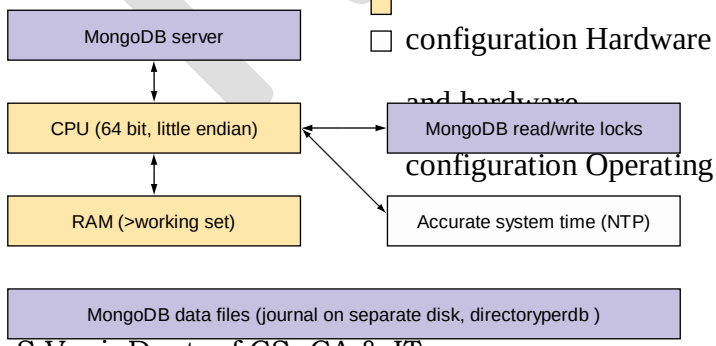
Here we'll present considerations for choosing good deployment environments for MongoDB. We'll discuss specific hardware requirements, such as CPU, RAM, and disks, and provide recommendations for optimizing the operating system environment.

ongoDB internals or

configuration Hardware

and hardware

configuration Operating



MongoDB isn't particularly CPU-intensive; database operations are rarely CPU-bound, so this isn't the first place to look when diagnosing a performance issue. Your first pri- ority when optimizing for MongoDB is to ensure operations aren't I/O-bound (we'll discuss I/O-bound issues more in the next two sections on RAM and disks).

But once your indexes and working set fit entirely in RAM, you may see some CPU- boundedness. If you have a single MongoDB instance serving tens (or hundreds) of thousands of queries per second, you can realize performance increases by providing more CPU cores.

If you do happen to see CPU saturation, check your logs for slow query warnings. There may be some types of queries that are inherently more CPU-intensive, or you may have an index issue that the logs will help you diagnose. But if that's not the case and you're still seeing CPU saturation, it's likely due to lock contention, which we'll briefly touch on here.

RAM

As with any database, MongoDB performs best with sufficient RAM. Be sure to select hardware with enough RAM to contain your frequently used indexes, plus your work- ing data set. Then as your data grows, keep a close eye on the ratio of RAM-to–working set size. If you allow working set size to grow beyond RAM, you may start to see signifi- cant performance degradation. Paging from disk in and of itself isn't a problem—it's a necessary step in loading data into memory.

DISKS

When choosing disks, you need to consider cost, IOPS (input/output operations per second), seek time, and storage capacity. The differences between running on a single consumer-grade hard drive, running in the cloud in a virtual disk (say, EBS), and run-ning against a high-performance SAN can't be overemphasized.

perform acceptably against a single network-attached EBS volume, but demanding applications will require something more.

Disk performance is important for a few reasons:

- *High write workloads*—As you're writing to MongoDB, the server must flush the data back to disk.

- With a write-intensive app and a slow disk, the flushing may be slow enough to negatively affect overall system performance.

- *A fast disk allows a quicker server warm-up*—Any time you need to restart a server, you also have to load your data set into RAM. This happens lazily; each succes- sive read or write to MongoDB will load a new virtual memory page into RAM until the physical memory is full. A fast disk will make this process much faster, which will increase MongoDB's performance following a cold restart.

- *A fast disk alters the required ratio of working set size to RAM for your application*— Using, say, a solid-state drive, you may be able to run with much less RAM (or much greater capacity) than you would otherwise.

Regardless of the type of disk used, serious deployments will generally use, not a single disk, but a redundant array of disks (RAID) instead. Users typically manage a RAID cluster using Linux's logical volume manager, LVM, with a RAID level of 10. RAID 10 provides redundancy while maintaining acceptable performance, and it's commonly used in MongoDB deployments.[4] Note that this is more expensive than a single disk, which illustrates the tradeoff between cost and performance. Even more advanced deployments will use a high-performance self-managed SAN, where the disks are all virtual and the idea of RAID may not even apply.

LOCKS

MongoDB's locking model is a topic unto itself. We won't discuss all the nuances of concurrency and performance in MongoDB here, but we'll cover the basic concur- rency models MongoDB supports. In practice, ensuring that you don't have lock con- tention will require careful monitoring or benchmarking because every workload is different and may have completely different points of contention.

In the early days, MongoDB took a global lock on the entire server. This was soon updated to a global lock on each database and support was added to release the lock before disk operations.

FILESYSTEMS

You'll get the best performance from MongoDB if you run it on the right filesystem. Two in particular, ext4 and xfs, feature fast, contiguous disk allocation. Using these filesystems will speed up MongoDB's frequent preallocations.[6]

Once you mount your fast filesystem, you can achieve another performance gain by disabling updates to files' last access time: atime. Normally, the operating system will update a file's atime every time the file is read or written. In a database environment, this amounts to a lot of unnecessary work. Disabling atime on Linux is relatively easy:

1 First, make a backup of the filesystem config file:

```
sudo cp /etc/fstab /etc/fstab.bak
```

2 Open the original file in your favorite editor:

```
sudo vim /etc/fstab
```

3 For each mounted volume you'll find inside /etc/fstab, you'll see a list of settings aligned by column. Under the options column, add the noatime directive:

```
# file-system        mount  type    options dump pass
UUID=8309beda-bf62-43  /ssd     ext4
                       noatime  0       2
```

4 Save your work. The new settings should take effect immediately.[7]

You can see the list of all mounted filesystems with the help of the findmnt command, which exists on Linux machines:

```
$ findmnt -s
TARGET SOURCE    FSTYPE OPTIONS
/proc  proc      proc   defaults
/      /dev/xvda ext3   noatime,errors=remount-
ro none                 /dev/xvdb swap       sw
```

The –soption makes findmnt get its data from the /etc/fstab file. Running findmnt without any command-line parameters shows more details yet more busy output.

FILE DESCRIPTORS

Some Linux systems cap the number of allowable open file descriptors at 1024. This is occasionally too low for MongoDB and may result in warning messages or even errors

when opening connections (which you'll see clearly in the logs). Naturally, MongoDB requires a file descriptor for each open file and network connection.

Assuming you store your data files in a folder with the word "*data"* in it, you can see the number of data file descriptors using lsof and a few well-placed pipes:

```
lsof | grep mongo | grep data | wc -l
```

Counting the number of network connection descriptors is just as easy:

```
lsof | grep mongo | grep TCP | wc -l
```

When it comes to file descriptors, the best strategy is to start with a high limit so that you never run out in production. You can check the current limit temporarily with the ulimit command:

```
ulimit -Hn
```

To raise the limit permanently, open your limits.conf file with your editor of choice:

```
sudo vim /etc/security/limits.conf
```

Then set the soft and hard limits. These are specified on a per-user basis. This exam- ple assumes that the mongodb user will run the mongod process:

```
mongodb soft nofile
2048 mongodb hard
nofile 10240
```

The new settings will take effect when that user logs in again.[8]

CLOCKS

It turns out that replication is susceptible to "clock skew," which can occur if the clocks on the machines hosting the various nodes of a replica set get out of sync. Replication

depends heavily on time comparisons, so if different machines in the same replica set disagree on the current time, that can cause serious problems. This isn't ideal, but for-

tunately there's a solution. You need to ensure that each of your servers uses NTP (Network Time Protocol) or some other synchronization protocol to keep their clocks synchronized:

- On Unix variants, this means running the ntpddaemon.
- On Windows, the Windows Time Services fulfills this role.

JOURNALING

MongoDB v1.8 introduced journaling, and since v2.0 MongoDB enables journaling by default. When journaling is enabled, MongoDB will commit all writes to a journal before writing to the core data files. This allows the MongoDB server to come back online quickly and cleanly in the event of an unclean shutdown.

Journaling obviates the need for database repairs because MongoDB can use the journal to restore the data files to a consistent state. In MongoDB v2.0 as well as v3.0, journaling is enabled by default, but you can disable it with the --nojournalflag:

```
$ mongod --nojournal
```

When enabled, the journal files will be kept in a directory called journal, located just below the main data path.

If you run your MongoDB server with journaling enabled, keep a of couple points in mind:

- First, journaling adds some additional overhead to writes.
- One way to mitigate this is to allocate a separate disk for the journal, and then either create a symlink[10] between the journal directory and the auxiliary volume or simply mount this disk where the journal directory should be. The auxiliary volume needn't be large; a 120 GB disk is more than sufficient, and a solid-state drive (SSD) of this size is affordable. Mounting a separate SSD for the journal files will ensure that journaling runs with the smallest possible per- formance penalty.
- Second, journaling by itself doesn't guarantee that no write will be lost. It guarantees only that MongoDB will always come back online in a consistent state. Journaling works by syncing a write buffer to disk every 100 ms, so an unclean shutdown can result in the loss of up to the last 100 ms of writes. If this isn't acceptable for any part of your application, you can change the write concern of operations through any client driver. You'd run this as a safe mode option (just

like w and wtimeout). For example, in the Ruby driver, you might use the j option like this in order to have safe mode enabled all the time for one of the servers:

```
client = Mongo::Client.new( ['127.0.0.1:27017'], :write => {:j
=> true}, :database => 'garden')
```

Logging

Logging is the first level of monitoring; as such, you should plan on keeping logs for all your deployments. This usually isn't a problem because MongoDB requires that you specify the --logpath option when running it in the background. But there are a few extra settings to be aware of. To enable verbose logging, start the mongod process with the -vvvvv option (the more vs, the more verbose the output). This is handy if, for instance, you need to debug some code and want to log every query. But do be aware that verbose logging will make your logs quite large and may affect server performance. If your logs become too unwieldy, remember that you can always store your logs on a different partition.

Next you can start mongod with the --logappend option. This will append to an existing log rather than moving it and appending a timestamp to the filename, which is the default behavior.

Finally, if you have a long-running MongoDB process, you may want to write a script that periodically rotates the log files. MongoDB provides the logrotate command for this purpose. Here's how to run it from the shell:

```
> use admin
> db.runCommand({logrotate: 1})
```

Sending the SIGUSR1[15] signal to the process also runs the logrotate command. Here's how to send that signal to process number 12345:

```
$ kill -SIGUSR1 12345
```

You can find the process ID of the process you want to send the signal to using the ps command, like this:

```
$ ps –ef | grep mongo
```

Note that the kill command isn't always as dire as it sounds. It only sends a signal to a running process, but was named in the days when most or all signals ended the pro-

cess.[16] But running kill with the -9 command-line option will end a process in a brutal way and should be avoided as much as possible on production systems.

MongoDB diagnostic commands

MongoDB has a number of database commands used to report internal state. These underlie all MongoDB monitoring applications. Here's a quick reference for a few of the commands that you might find useful:

- Global server statistics: db.serverStatus()
- Stats for currently running operation: db.currentOp(
- Include stats for idle system operations: db.currentOp(true)
- Per database counters and activity stats: db.runCommand({top:1})
- Memory and disk usage statistics: db.stats()

The output for all of these commands improves with each MongoDB release, so documenting it in a semi-permanent medium like this book isn't always helpful. Consult the documentation for your version of MongoDB to find out what each field in the output means.

MongoDB diagnostic tools

In addition to the diagnostic commands listed previously, MongoDB ships with a few handy diagnostic tools. Most of these are built on the previous commands and could be easily implemented using a driver or the mongo shell.

Here's a quick introduction to what we'll cover in this section:

- mongostat—Global system statistics
- mongotop—Global operation statistics
- mongosniff(advanced)—Dump MongoDB network traffic
- bsondump—Display BSON files as JSON

MONGOSTAT

The db.currentOp() method shows only the operations queued or in progress at a particular moment in time. Similarly, the serverStatus command provides a point-in-time snapshot of various system fields and counters. But sometimes you need a view of the system's real-time activity, and that's where mongostat comes in. Modeled after iostat and other similar tools, mongostat polls the server at a fixed interval and displays an array of statistics, from the number of inserts per second to the amount of resident memory, to the frequency of B-tree page misses.

You can invoke the mongostatcommand on localhost, and the polling will occur once a second:

$ mongostat

Similar to the way mongostat is the external tool for the db.currentOp() and server-Status commands, mongotop is the external tool for the top command. You can run this in exactly the same way as mongostat, assuming you have a server running on the local machine and listening on the default port:

$ mongotop

As with mongostat, you can run this command with –helpto see a number of useful configuration options.

MONGOSNIFF

The next command we'll cover is mongosniff, which sniffs packets from a client to the MongoDB server and prints them intelligibly. If you happen to be writing a driver or debugging an errant connection, then this is your tool. You can start it up like this to listen on the local network interface at the default port:

sudo mongosniff --source NET I0

Then when you connect with any client—say, the MongoDB shell—you'll get an easy- to-read stream of network chatter:

```
127.0.0.1:58022 -->> 127.0.0.1:27017 test.$cmd 61 bytes id:89ac9c1d
2309790749 query: { isMaster: 1.0 } ntoreturn: -1
127.0.0.1:27017  <<--  127.0.0.1:58022   87 bytes
reply n:1 cursorId: 0 { ismaster: true, ok: 1.0 }
```

Here you can see a client running the isMaster command, which is represented as a query for {isMaster:1.0}against the special test.$cmdcollection. You can also see that the response document contains ismaster: true, indicating that the node that this command was sent to was in fact the primary. You can see all the mongosniff options by running it with --help.

BSONDUMP

Another useful command is bsondump, which allows you to examine raw BSON files. BSON files are generated by the mongodump command (discussed in section 13.3) and by replica set rollbacks.[17] For instance, let's say you've dumped a collection with a sin- gle document. If that collection ends up in a file called users.bson, then can examine the contents easily:

```
$ bsondump users.bson
{ "_id" : ObjectId( "4d82836dc3efdb9915012b91" ), "name" : "Kyle" }
```

As you can see, bsondump prints the BSON as JSON by default. If you're doing serious debugging, you'll want to see the real composition of BSON types and sizes. For that, run the command in debug mode:

```
$ bsondump --type=debug users.bson
--- new object ---
size : 37
_id
type: 7 size: 17
name
type: 2 size: 15
```

This gives you the total size of the object (37 bytes), the types of the two fields (7 and 2), and those fields' sizes.

THE WEB CONSOLE

Finally, MongoDB provides some access to statistics via a web interface and a REST server. As of v3.0, these systems are old and under active development. On top of that, they report the same information available via the other tools or database commands presented earlier. If you want to use these systems, be sure to look at the currentdocu- mentation and carefully consider the security implications.

MongoDB Monitoring Service

MongoDB, Inc. provides MMS Monitoring for free, which not only allows you to view dashboards to help you understand your system, but also provides an easy way to share your system information with MongoDB support, which is indispensable if

you ever need help with your system. MMS Monitoring can also be licensed as a self-hosted ver- sion for large enterprises with paid contracts. To get started, all you need to do is cre- ate an account on the MMS Monitoring website at https://mms.mongodb.com. Once you create an account, you'll see instructions to walk you through the process of set- ting up MMS, which we won't cover here.

External monitoring applications

Most serious deployments will require an external monitoring application. Nagios and Munin are two popular open source monitoring systems used to keep an eye on many MongoDB deployments. You can use each of these with MongoDB by installing a sim- ple open source plug-in.

Writing a plug-in for any arbitrary monitoring application isn't difficult. It mostly involves running various statistics commands against a live MongoDB database. The serverStatus, dbstats, and collstats commands usually provide all the informa- tion you might need, and you can get all of them straight from the HTTP REST inter-face, avoiding the need for a driver.

Finally, don't forget the wealth of tools available for low-level system monitoring. For example, the iostat command can be helpful in diagnosing MongoDB perfor- mance issues. Most of the performance issues in MongoDB deployments can be traced to a single source: the hard disk.

In the following example, we use the -xoption to show extended statistics and specify 2to display those stats at two-second intervals:

```
$ iostat -x 2
Device:  rsec/s  wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdb      0.00   3101.12   10.09    32.83  101.39   1.34  29.36
Device:  rsec/s  wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdb      0.00   2933.93    9.87    23.72  125.23   1.47  34.13
```

or a detailed description of each of these fields, or for details on your specific version of iostat, consult your system's man[18] pages. For a quick diagnostic, you'll be most interested in two of the columns shown:

- The awaitcolumn indicates the average time in milliseconds for serving I/O requests. This average includes time spent in the I/O queue and time spent actually servicing I/Orequests.

%utilis the percentage of CPU during which I/O requests were issued to the device, which essentially translates to the bandwidth use of the device.

Backups

Part of running a production database deployment is being prepared for disasters. Backups play an important role in this. When disaster strikes, a good backup can save the day, and in these cases, you'll never regret having invested time and diligence in a regular backup policy. Yet some users still decide that they can live without backups. These users have only themselves to blame when they can't recover their databases. Don't be one of these users.

Three general strategies for backing up a MongoDB database are as follows:

- Using mongodumpand mongorestore
- Copying the raw data files
- Using MMS Backups

We'll go over each of these strategies in the next three sections.

mongodump and mongorestore

mongodumpwrites the contents of a database as BSON files. mongorestorereads these files and restores them. These tools are useful for backing up individual collections and databases as well as the whole server. They can be run against a live server (you don't have to lock or shut down the server), or you can point them to a set of data files,

---

but only when the server is locked or shut down. The simplest way to run mongodump is like this:[19]

```
$ mongodump -h localhost --port 27017
```

This will dump each database and collection from the server at localhost to a directory called dump.[20] The dump directory will include all the documents from each collec- tion, including the system collections that define users and indexes. But significantly, the indexes themselves won't be included in the dump. This means that when you restore, any indexes will have to be rebuilt. If you have an especially large dataset, or a large number of indexes, this will take time.

RESTORING BSON FILES

To restore BSON files, run mongorestoreand point it at the dump folder:

$ mongorestore -h localhost --port 27017 dump

Note that when restoring, mongorestorewon't drop data by default, so if you're restoring to an existing database, be sure to run with the --dropflag.

Data file–based backups

Most users opt for a file-based backup, where the raw data files are copied to a new location. This approach is often faster than mongodump because the backups and restorations require no transformation of the data.

The only potential problem with a file-based backup is that it requires locking the database, but generally you'll lock a secondary node and thus should be able to keep your application online for the duration of the backup.

To safely copy the data files, you first need to make sure that they're in a consistent state, so you either have to shut down the database or lock it. Because shutting down the database might be too involved for some deployments, most users opt for the locking approach. Here's the command for syncing and locking:

```
> use admin
> db.fsyncLock()
```

Security

Security is an extremely important, and often overlooked, aspect of deploying a production database. In this section, we'll cover the main types of security, including secure environments, network encryption, authentication, and authorization.

We'll end with a brief discussion of which security features are only available in the enterprise edition of MongoDB. Perhaps more than for any other topic, it's vital to stay up to date with the current security tools and best practices, so treat this section as an overview of what to consider when thinking about security, but consult the most recent documentation at https://docs.mongodb.org/manual/security when putting it into production.

Secure environments

MongoDB, like all databases, should be run in a secure environment. Production users of MongoDB must take advantage of the security features of modern operating systems to ensure the safety of their data. Probably the most important of these features is the firewall.

The only potential difficulty in using a firewall with MongoDB is knowing which machines need to communicate with each other. Fortunately, the communication rules are simple:

- With a replica set, each node must be able to reach every other node.
- All database clients must be able to connect with every replica set node that the client might conceivably talk to.
- All communication is done using the TCP protocol.
- For a node to be reachable, it means that it's reachable on the port that it was configured to listen on. For example, mongod listens on TCP port 27017 by default, so to be reachable it must be reachable on that port.

Network encryption

Perhaps the most fundamental aspect of securing your system is ensuring your network traffic is encrypted. Unless your system is completely isolated and no one you don't trust can even see your traffic (for example, if all your traffic is already encrypted over a virtual private network, or your network routing rules are set up such that no traffic can be sent to your machines from outside your trusted network[23]), you should probably use MongoDB with encryption.

Fortunately, as of v2.4, MongoDB ships with a library that handles this encryption—called the Secure Sockets Layer (SSL)—built in.

Here's what the beginning of the output looks like on our machine:

```
$ ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
        ...
        ...
```

For us, the loopback interface is lo. Now we can use the appropriate tcpdumpcommand to dump all traffic on this interface:

```
$ sudo tcpdump -i lo –X
```

NOTE Reading network traffic using tcpdumprequires root permissions, so if you can't run this command, just read along with the example that follows.

In another terminal on the same machine, start a mongodserver without SSL enabled (change the data path as appropriate):

```
$ mongod --dbpath /data/db/
```

Then, connect to the database and insert a single document:

```
$ mongo
...
> db.test.insert({ "message" : "plaintext" }) >exit
bye
```

Now, if you look at the tcpdumpoutput in the terminal, you'll see a number of packets output, one of which looks something like this:

```
16:05:10.507867 IP localhost.localdomain.50891 >
    ➥ localhost.localdomain.27017 …
    0x0000:  4500 007f aa4a 4000 4006 922c 7f00 0001   E....J@.@..,....
    0x0010:  7f00 0001 c6cb 6989 cf17 1d67 d7e6 c88f  . ....i....g....
    0x0020:  8018 0156 fe73 0000 0101 080a 0018 062e  ...V.s. ........
```

There's our message, right in the clear B! This shows how important network encryption is. Now, let's run MongoDB with SSL and see what happens.

RUN MONGODB WITH SSL

First, generate the key for the server:

```
openssl req -newkey rsa:2048 -new -x509 -days 365 -nodes -out mongodbcert.crt
-keyout mongodb-cert.key
cat mongodb-cert.key mongodb-cert.crt > mongodb.pem
```

Then, run the mongodserver with SSL, using the --sslPEMKeyFileand --sslMode options:

$ mongod --sslMode requireSSL --sslPEMKeyFile mongodb.pem

Now, connect the client with SSL and do exactly the same operation:

```
$ mongo --ssl
...
> db.test.insert({ "message" : "plaintext" }) >exit
bye
```

If you now go back to the window with tcpdump, you'll see something completely incomprehensible where the message used to be:

```
16:09:26.269944 IP localhost.localdomain.50899 >
    localhost.localdomain.27017: …
      0x0000:  4500 009c 52c3 4000 4006 e996 7f00 0001  E...R.@.@. .....
      0x0010:  7f00 0001 c6d3 6989 c46a 4267 7ac5 5202  . ....i..jBgz.R.
      0x0020:  8018 0173 fe90 0000 0101 080a 001b ed40  ...s.....@
```

SERVICE AUTHENTICATION

The first stage of authentication is verifying that the program on the other end of the connection is trusted. Why is this important? The main attack that this is meant to prevent is the *man-in-the-middle attack*, where the attacker masquerades as both the client and the server to intercept all traffic between them.

As you can see in the figure, a man-in-the-middle attack is exactly what it sounds like:

- A malicious attacker poses as a server, creating a connection with the client, and then poses as the client and creates a connection with the server.
- After that, it can not only decrypt and encrypt all the traffic between the client and server, but it can send arbitrary messages to both the client and the server.

Once you have a certificate, you can use it in MongoDB like this

```
mongod --clusterAuthMode x509 --sslMode requireSSL --sslPEMKeyFile server.pem
--sslCAFile ca.pem
mongo --ssl --sslPEMKeyFile client.pem
```

where ca.pem contains the root certificate chain from the CA and client.pem is signed by that CA. The server will use the contents of ca.pem to verify that client.pem was indeed signed by the CA and is therefore trusted.

Taking these steps will ensure that no malicious program can establish a connection to your database. In the next section, you'll see how to make this more fine-grained and authenticate individual users in a single database.

straight into an example of how to set up basic authentication for a single mongod.

SETTING UP BASIC AUTHENTICATION

First, you should start a mongod node with auth enabled. Note that if this node is in a sharded cluster or a replica set, you also need to pass options to allow it to authenticate with other servers. But for a single node, enabling authentication requires only one flag:

```
$ mongod --auth
```

Now, the first time you connect to the server, you want to add an administrative user account:

```
> use admin
> db.createUser(
  {
    user: "boss",
    pwd: "supersecretpassword",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  }
)
```

In our example, we gave this user a role of userAdminAnyDatabase, which essentially gives the user complete access to the system, including the ability to add and remove new users, as well as change user privileges.

This is essentially the superuser of MongoDB.

Now that we've created our admin user, we can log in as this user:

```
> use admin
> db.auth("boss", "supersecretpassword")
```

We can now create users for individual databases. Once again we use the createUser method. The main differences here are the roles:

```
> use stocks
> db.createUser(
  {
    user: "trader",
    pwd: "youlikemoneytoo",
    roles: [ { role: "readWrite", db: "stocks" } ]
  }
)
> db.createUser(
  {
    user: "read-only-trader",
    pwd: "weshouldtotallyhangout",
    roles: [ { role: "read", db: "stocks" } ]
  }
)
```

Now the traderuser has the readWriterole on the stocksdatabase, whereas the read-only-traderonly has the readrole. This essentially means that the first user can read and write stock data, and the second can only read it. Note that because we added these users to the stocksdatabase, we need to authenticate using that database as well:

```
> use stocks
> db.auth("trader", "youlikemoneytoo")
```

REMOVING A USER

To remove a user, use the dropUserhelper on the database it was added to:

```
> use stocks
> db.dropUser("trader")
```

This is a bit heavyweight, so note that you can also revoke user access without completely dropping them from the system using the revokeRolesFromUser helper, and grant them roles again using the grantRolesToUserhelper.

To close the session you don't need to explicitly log out; terminating the connection (closing the shell) will accomplish that just fine. But there's a helper for logging out if you need it:

```
> db.logout()
```

Naturally, you can use all the authentication logic we've explored here using the drivers. Check your driver's API for the details.

Replica set authentication

Replica sets support the same authentication API just described, but enabling authentication for a replica set requires extra configuration, because not only do clients need to be able to authenticate with the replica set, but replica set nodes also need to be able to authenticate with each other.

Internal replica set authentication can be done via two separate mechanisms:

- Key file authentication
- X509 authentication

In both cases, each replica set node authenticates itself with the others as a special internal user that has enough privileges to make replication work properly.

KEY FILE AUTHENTICATION

The simpler and less secure authentication mechanism is key file authentication. This essentially involves creating a "key file" for each node that contains the password that replica set node will use to authenticate with the other nodes in the replica set. The upside of this is that it's easy to set up, but the downside is that if an attacker compromises just one machine, you'll have to change the password for every node in the cluster, which unfortunately can't be done without downtime.

To start, create the file containing your secret. The contents of the file will serve as the password that each replica set member uses to authenticate with the others. As an example, you might create a file called secret.txt and fill it with the following (don't actually use this password in a real cluster):

tOps3cr3tpa55word

Place the file on each replica set member's machine and adjust the permissions so that it's accessible only by the owner:

sudo chmod 600 /home/mongodb/secret.txt

Finally, start each replica set member by specifying the location of the password file using the --keyFileoption:

mongod --keyFile /home/mongodb/secret.txt

Authentication will now be enabled for the set. You'll want to create an admin user in advance, as you did in the previous section.

X509 AUTHENTICATION

X509 certificate authentication is built into OpenSSL, the library MongoDB uses to encrypt network traffic. As we mentioned earlier, obtaining signed certificates is outside the scope of this book. However, once you have them, you can start each node like this

mongod --replSet myReplSet --sslMode requireSSL --clusterAuthMode x509 -- sslClusterFile --sslPEMKeyFile server.pem --sslCAFile ca.pem

where server.pemis a key signed by the certificate authority that ca.pemcorresponds to.

There's a way to upgrade a system using key file authentication to use X509 certificates with no downtime. See the MongoDB docs for the details on how to do this, or check in the latest MMS documentation to see whether support has been added to MMS automation.

Sharding authentication

Sharding authentication is an extension of replica set authentication. Each replica set in the cluster is secured as described in the previous section. In addition, all the config servers and every mongosinstance can be set up to authenticate with the rest of the clusterinexactlythesameway,

using either a shared key file or using X509 certificate authentication. Once you've done this, the whole cluster can use authentication.

Enterprise security features

Some security features exist only in MongoDB's paid enterprise plug-in. For example, the authentication and authorization mechanisms that allow MongoDB to interact with Kerberos and LDAP are enterprise. In addition, the enterprise module adds auditing support so that security-related events get tracked and logged. The MongoDB docs will explicitly mention if a particular feature is enterprise only.

Administrative tasks

In this section, we'll cover some basic administrative tasks, including importing and exporting data, dealing with disk fragmentation, and upgrading your system.

Compaction and repair

MongoDB includes a built-in tool for repairing a database. You can initiate it from the command line to repair all databases on the server:

```
$ mongod --repair
```

Or you can run the repairDatabase command to repair a single database:

```
> use cloud-docs
> db.runCommand({repairDatabase: 1})
```

To rebuild indexes, use the reIndex() method:

```
> use cloud-docs
> db.spreadsheets.reIndex()
```

This might be useful, but generally speaking, index space is efficiently reused. The data file space is what can be a problem, so the compact command is usually a better choice. compact will rewrite the data files and rebuild all indexes for one collection. Here's how you run it from the shell:

```
> db.runCommand({ compact: "spreadsheets" })
```

This command has been designed to be run on a live secondary, obviating the need for downtime. Once you've finished compacting all the secondaries in a replica set, you can step down the primary and then compact that node. If you must run the compact

command on the primary, you can do so by adding {force: true} to the command object. Note that if you go this route, the command will write lock the system:

```
> db.runCommand({ compact: "spreadsheets", force: true })
```

On WiredTiger databases, the compact() command will release unneeded disk space to the operating system. Also note that the paddingFactor field, which is applicable for the MMAPv1 storage engine, has no effect when used with the WiredTiger stor- age engine.

Upgrading

As with any software project, you should keep your MongoDB deployment as up to date as possible, because newer versions contain many important bug fixes and improvements.

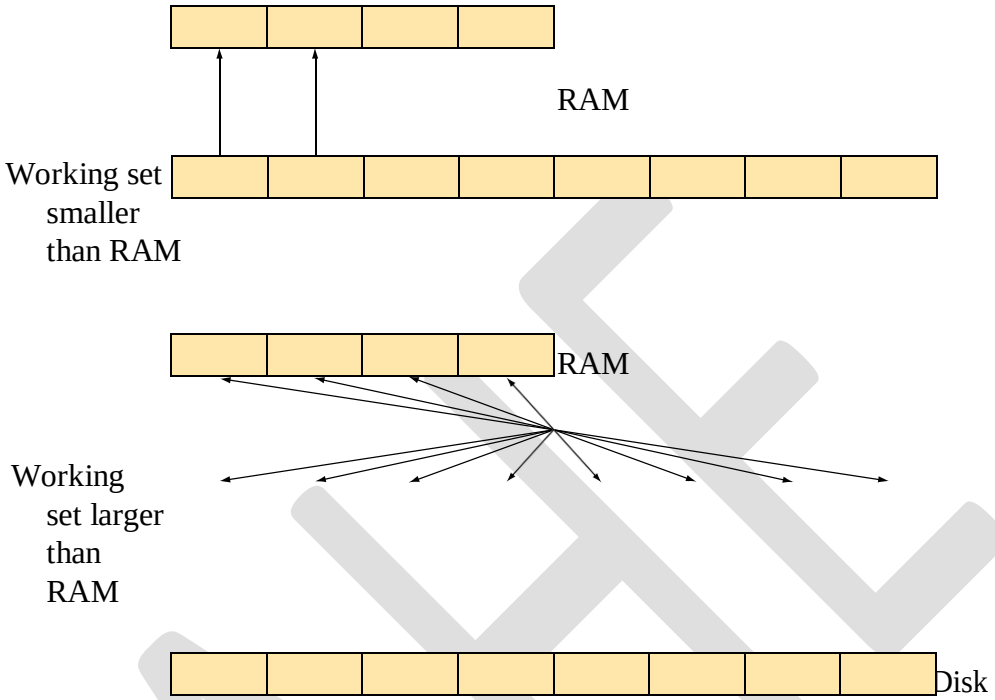
One of the core design principles behind MongoDB is to always ensure an upgrade is possible with no downtime. For a replica set, this means a rolling upgrade, and for a sharded cluster, this means that mongosrouters can still function against mixed clusters.

Working set

We've covered the idea of the working set in various parts of this book, but we'll define it here again with a focus on your production deployment.

Imagine you have a machine with 8 GB of RAM, running a database with an on-disk size of 16 GB, not including indexes. Your *working set* is how much data you're access- ing in a specified time interval. In this example, if your queries are all full collection scans, your "working set" will be 16 GB because to answer those queries your entire database must be paged into memory.

But if your queries are properly indexed, and you're only querying for the most recent quarter of the data, most of your database can stay on disk, and only the 2 GB that you need, plus some extra space for indexes, needs to be in memory.



Query interactions

Another side effect of the fact that MongoDB doesn't enforce resource limits is that one badly behaving query can affect the performance of all other queries on the system.

It's the same drill as before. Assume you have a working set of 2 GB, with a 64 GB database. Everything may be going well, until someone runs a query that performs a full collection scan. This query will not only place a huge amount of load on the disk, but may also page out the data that was being used for the other queries on the system, causing slowdown there as well. Figure 13.4 from earlier illustrates this issue, where the top represents normal query load and the bottom represents the load after the bad query.

This is actually another reason why access controls are important. Even if you get everything else right, one table scan by an intern can hose your system. Make sure everyone who has the ability to query your database understands the consequences of a

The sources of performance degradations are manifold and frequently idiosyncratic. Anything from poor schema design to sneaky server bugs can negatively affect performance.

If you think you've tried every possible remedy and still can't get results, con- sider allowing someone experienced in the ways of MongoDB to audit your system.

A book can take you far, but an experienced human being can make all the difference in the world. When you're at a loss for ideas and in doubt, seek professional assistance. The solutions to performance issues are sometimes entirely unintuitive.

When or if you seek help, be sure to provide all the information you have about your system when the issue occurred. This is when the monitoring will pay off. The official standard used by MongoDB is MMS Monitoring, so if you're using MongoDB support, being set up with MMS Monitoring will speed up the process significantly.

Deployment checklist

We've covered a lot of topics in this chapter. It may seem overwhelming at first, but as long as you have the main areas covered, your system will keep running smoothly. This section is a quick reference for making sure you've got the important points covered:

- Hardware
  - *RAM*—Enough to handle the expected working set.
  - *Disk space*—Enough space to handle all your data, indexes, and MongoDB internal metadata.
  - *Disk speed*—Enough to satisfy your latency and throughput requirements. Consider this in conjunction with RAM—less RAM usually means more disk usage.
  - *CPU*—Usually not the bottleneck for MongoDB, but if you're getting low disk utilization but low throughput, you may have a CPU bound workload. Check this as part of careful performance testing.
  - *Network*—Make sure the network is fast and reliable enough to satisfy your performance requirements. MongoDB nodes communicate with each other internally, so be sure to test every connection, not just the ones from your clients to the mongos or mongod servers.
- Security
  - *Protection of network traffic*—Either run in a completely isolated environment or make sure your traffic is encrypted using MongoDB's built-in SSL support or some external method such as a VPN to prevent man-in-the-

middle attacks.

- *Access control* —Make sure only trusted users and clients programs can oper- ate on the database. Make sure your interns don't have the "root" privilege.

- Monitoring
  - *Hardware usage (disks, CPU, network, RAM)*—Make sure you have some kind of monitoring setup for all your hardware resources that will not only keep track of the usage, but also alert you if it goes above a certain threshold.
  - *Health checks*—Periodically make sure your servers are up and responsive, and will alert you if anyone stops calling back.
  - *MMS Monitoring* —Monitor your services using MMS Monitoring. Not only does this provide monitoring, health checks, and alerts, but it's what the MongoDB support team will use to help you if you run into trouble. Histori- cally it's been free to use, so don't hesitate to add this to your deployment.
  - *Client performance monitoring* —Periodically run automated end-to-end tests as a client to ensure that you're still performing as well as you expect. The last thing you want is for a client to be the first one to tell you that your applica- tion is slow.
- Disaster recovery
  - *Evaluate risk*—Imagine that you've lost all your data. How sad do you feel? In all seriousness, losing your data may be worse in some applications than oth- ers. If you're analyzing Twitter trends, losing your data may cost a week's worth of time, whereas if you're storing bank data, losing that may cost quite a bit more. When you do this evaluation, assume that a disaster of some kind will happen, and plan accordingly.
  - *Have a plan*—Create a concrete plan for how you'll recover in each failure case. Depending on how your system fails, you may react completely differently.
  - *Test your plan*—Be sure to test your plan. The biggest mistake people make with backups and disaster recovery is assuming that having a backup or a plan is enough. It's not enough. Maybe the backup is getting corrupted. Maybe it's in a format that's impossible to reimport into your production sys- tems. As in a production system, many things can go wrong, so it's important to make sure your recovery strategy works.

- *Have a backup plan*—Your first disaster recovery plan might fail. When it does, have a last resort available. This doesn't have to be an appealing option, but you'll be happy it's there if you get desperate.

- Performance
  - *Load testing*—Make sure you load test your application with a realistic work- load. In the end, this is the only way to be sure that your performance is what you expect.

MongoDB on Mac OS X

If you're using Mac OS X, you have three options for installing MongoDB. You can download the precompiled binaries directly from the mongodb.org website, use a package manager, or compile manually from source. We'll discuss the first two options in the next sections, and then provide a few notes on compiling later in the appendix.

Precompiled binaries

First navigate to www.mongodb.org/downloads. There you'll see a grid with all the latest downloadable MongoDB binaries. Select the download URL for the latest stable version for your architecture. The following example uses MongoDB v3.0.6 compiled for a 64-bit system.

Download the archive using your web browser or the curl utility. You should check on the downloads page for the most recent release. Then expand the archive using tar:

```
$ curl https://fastdl.mongodb.org/osx/mongodb-osx-x86_64-
3.0.6.tgz > mongo.tgz
$ tar xzvf mongo.tgz
```

To run MongoDB, you'll need a data directory. By default, the mongod daemon will store its data files in /data/db. Go ahead and create that directory:

```
$ sudo mkdir -p /data/db/
$ sudo chown `id -u` /data/db
```

You're now ready to start the server. Just change to the MongoDB bin directory and launch the mongod executable:

```
$ cd mongodb-osx-x86_64-3.0.6/bin
```

```
$ ./mongod
```

If all goes well, you should see something like the following abridged startup log. The first time you start the server it may allocate journal files, which takes several minutes, before being ready for connections. Note the last lines, confirming that the server is listening on the default port of 27017:

```
2015-09-19T08:51:40.214+0300 I CONTROL [initandlisten] MongoDB
    starting : pid=41310 port=27017 dbpath=/data/db 64-bit
    host=iron.local
2015-09-19T08:51:40.214+0300 I CONTROL [initandlisten] db version v3.0.6
...
2015-09-19T08:51:40.215+0300 I INDEX    [initandlisten] allocating
    new ns file /data/db/local.ns, filling with zeroes...
2015-09-19T08:51:40.240+0300 I STORAGE [FileAllocator]
    allocating new datafile /data/db/local.0, filling with zeroes...
2015-09-19T08:51:40.240+0300 I STORAGE [FileAllocator] creating
    directory / data/db/_tmp
2015-09-19T08:51:40.317+0300 I STORAGE [FileAllocator] done
    allocating datafile /data/db/local.0, size: 64MB, took 0.077 secs
2015-09-19T08:51:40.344+0300 I NETWORK [initandlisten] waiting
    for connections on port 27017
```

You should now be able to connect to the MongoDB server using the JavaScript console by running ./mongo. If the server terminates unexpectedly, refer to section A.6.

Using a package manager

MacPorts (http://macports.org) and Homebrew (http://brew.sh/) are two package managers for Mac OS X known to maintain up-to-date versions of MongoDB. To install via MacPorts, run the following:

```
sudo port install mongodb
```

Note that MacPorts will build MongoDB and all its dependencies from scratch. If you go this route, be prepared for a lengthy compile.

Homebrew, rather than compiling, merely downloads the latest binaries, so it's much faster than MacPorts. You can install MongoDB through Homebrew as follows:

```
$ brew update
$ brew install mongodb
```

After installing, Homebrew will provide instructions on how to start MongoDB using the Mac OS X launch agent.

MongoDB on Windows

If you're using Windows, you have two ways to install MongoDB. The easier, pre- ferred way is to download the precompiled binaries directly from the mongodb.org website. You can also compile from source, but this option is recommended only for developers and advanced users. You can read about compiling from source in the next section.

Precompiled binaries

First navigate to www.mongodb.org/downloads. There you'll see a grid with all the latest downloadable MongoDB binaries. Select the download URL for the latest sta- ble version for your architecture. Here we'll install MongoDB v2.6 compiled for 64-bit Windows.

Alternatively, you can use the command line. First navigate to your Downloads directory. Then use the unziputility to extract the archive:

```
C:\> cd \Users\kyle\Downloads


C:\> unzip mongodb-win32-x86_64-2.6.7.zip
```

To run MongoDB, you'll need a data folder. By default, the mongod daemon will store its data files in C:\data\db. Open the Windows command prompt and create the folder like this:

```
C:\> mkdir \data
C:\> mkdir
```

\data\db

You're now ready to start the server. Change to the MongoDB bin directory and launch the mongodexecutable:

```
C:\> cd \Users\kyle\Downloads
C:\Users\kyle\Downloads> cd mongodb-win32-x86_64-2.6.7\bin
C:\Users\kyle\Downloads\mongodb-win32-x86_64-2.6.7\bin>
mongod.exe
```

If all goes well, you should see something like the following abridged startup log. The first time you start the server it may allocate journal files, which takes several minutes, before being ready for connections. Note the last lines, confirming that the server is listening on the default port of 27017:

```
Thu Mar 10 11:28:51 [initandlisten] MongoDB
  starting : pid=1773 port=27017 dbpath=/data/db/ 64-
  bit host=iron
Thu Mar 10 11:28:51 [initandlisten] db version v2.6.7
...
Thu Mar 10 11:28:51 [websvr] web admin console waiting for connections on
  port 28017
Thu Mar 10 11:28:51 [initandlisten] waiting for connections on port 27017
```

If the server terminates unexpectedly, refer to section A.6.

Finally, you'll want to start the MongoDB shell. To do that, open a second terminal window, and then launch mongo.exe:

```
C:\> cd \Users\kyle\Downloads\mongodb-win32-x86_64-
2.6.7\bin C:\Users\kyle\Downloads\mongodb-win32-x86_64-
2.6.7\bin> mongo.exe
```

Compiling MongoDB from source

Compiling MongoDB from source is recommended only for advanced users and developers. If all you want to do is operate on the bleeding edge, without having to compile, you can always download the nightly binaries for the latest revisions from the mongodb.org website.

That said, you may want to compile yourself. The trickiest part about compiling

MongoDB is managing the various dependencies. The latest compilation instructions for each platform can be found at www.mongodb.org/about/contributors/tutorial/ build-mongodb-from-source.

Troubleshooting

MongoDB is easy to install, but users occasionally experience minor problems. These usually manifest as error messages generated when trying to start the mongod daemon. Here we provide a list of the most common of these errors along with their resolutions.

Wrong architecture

If you try to run a binary compiled for a 64-bit system on a 32-bit machine, you'll see an error like the following:

```
bash: ./mongod: cannot execute binary file
```

On Windows 7, the message is more helpful:

```
This version of
C:\Users\kyle\Downloads\mongodb-win32-x86_64-
2.6.7\bin\mongod.exe is not compatible with the version of
Windows you're running.
Check your computer's system information to see whether you
need a x86 (32-bit) or x64 (64-bit) version of the program, and
then contact the software publisher.
```

The solution in both cases is to download and then run the 32-bit binary instead. Bina- ries for both architectures are available on the MongoDB download site (www.mongodb .org/downloads).

Nonexistent data directory

MongoDB requires a directory for storing its data files. If the directory doesn't exist, you'll see an error like the following:

```
dbpath (/data/db/) does not exist, terminating
```

The solution is to create this directory. To see how, consult the preceding instructions

for your OS.

Lack of permissions

If you're running on a Unix variant, you'll need to make sure that the user running the mongodexecutable has permissions to write to the data directory. Otherwise, you'll see this error

Permission denied: "/data/db/mongod.lock", terminating

or possibly this one:

Unable to acquire lock for lockfilepath: /data/db/mongod.lock, terminating

In either case, you can solve the problem by opening up permissions in the data directory using chmodor chown.

Unable to bind to port

MongoDB runs by default on port 27017. If another process, or another mongod, is bound to the same port, you'll see this error:

listen(): bind() failed errno:98
Address already in use for socket: 0.0.0.0:27017

This issue has two possible solutions. The first is to find out what other process is running on port 27017 and then terminate it, provided that it isn't being used for some other purpose. One way of finding which process listens to port number 27017 is the following:

sudo lsof -i :27017

The output of the lsofcommand will also reveal the process ID of the process that listens to port number 27017, which can be used for killing the process using the killcommand.

Alternatively, run mongodon a different port using the --portflag, which seems to be a better and easier solution. Here's how to run MongoDB on port 27018:

mongod --port 27018

Basic configuration options

Here's a brief overview of the flags most commonly used when running MongoDB:

- --dbpath—The path to the directory where the data files are to be stored. This defaults to /data/db and is useful if you want to store your MongoDB data elsewhere.

- --logpath—The path to the file where log output should be directed. Log output will be printed to standard output (stdout) by default.
- --port—The port that MongoDB listens on. If not specified, it's set to 27017.
- --rest—This flag enables a simple REST interface that enhances the server's default web console. The web console is always available 1000 port numbers above the port the server listens on. Thus if the server is listening at localhost on port 27017, then the web console will be available at http://localhost:28017. Spend some time exploring the web console and the commands it exposes; you can discover a lot about a live MongoDB server this way.
- --fork—Detaches the process to run as a daemon. Note that fork works only on Unix variants. Windows users seeking similar functionality should look at the instructions for running MongoDB as a proper Windows service. These are available at www.mongodb.org.

Those are the most important of the MongoDB startup flags. Here's an example of their use on the command line:

```
$ mongod --dbpath /var/local/mongodb --logpath /var/log/mongodb.log
--port 27018 --rest --fork
```

Note that it's also possible to specify all of these options in a config file. Create a new text file (we'll call it mongodb.conf) and you can specify the config file equivalent[1] of all the preceding options:

```
storage:
    dbPath:
"/var/local/mongodb"
systemLog:
    destination: file
    path:
"/var/log/mongodb.log" net:
    port:
    27018
    http:
```

```
RESTInterfaceEnabled:
true processManagement:
    fork: true
```

You can then invoke mongodusing the config file with the -foption:

```
$ mongod -f mongodb.conf
```

If you ever find yourself connected to a MongoDB and wondering which options were used at startup, you can get a list of them by running the getCmdLineOptscommand:

```
> use admin
> db.runCommand({getCmdLineOpts: 1})
```

Installing Ruby

A number of the examples in this book are written in Ruby, so to run them yourself, you'll need a working Ruby installation. This means installing the Ruby interpreter as well as Ruby's package manager, RubyGems.

You should use a newer version of Ruby, such as 1.9.3 or preferably 2.2.3, which is the current stable version. Version 1.8.7 is still used by many people, and it works well with MongoDB, but the newer versions of Ruby offer advantages such as better character encoding that make it worthwhile to upgrade.

Linux and Mac OS X

Ruby comes installed by default on Max OS X and on a number of Linux distributions. You may want to check whether you have a recent version by running

```
ruby -v
```

If the command isn't found, or if you're running a version older than 1.8.7, you'll want to install or upgrade. There are detailed instructions for installing Ruby on Mac OS X as well as on a number of Unix variants at https://www.ruby-lang.org/en/downloads/

In addition to the Ruby interpreter, you need the Ruby package manager, RubyGems, to install the MongoDB Ruby driver. Find out whether RubyGems is installed by running the gemcommand:

```
gem -v
```

You can install RubyGems through a package manager, but most users download the latest version and use the included installer. You can find instructions for doing this at https://rubygems.org/pages/download.

Windows

By far, the easiest way to install Ruby and RubyGems on Windows is to use the Windows Ruby Installer. The installer can be found here: http://rubyinstaller.org/downloads. When you run the executable, a wizard will guide you through the installation of both Ruby and RubyGems.

In addition to installing Ruby, you can install the Ruby DevKit, which permits the easy compilation of Ruby C extensions. The MongoDB Ruby driver's BSON library may optionally use these extensions.

Many-to-many

In RDBMSs, you use a join table to represent many-to-many relationships; in MongoDB, you use array keys. You can see a clear example of this technique earlier in the book where we relate products and categories. Each product contains an array of category IDs, and both products and categories get their own collections. If you have two simple category documents

```
{ _id:
  ObjectId("4d6574baa6b804ea563c132a")
  , title: "Epiphytes"
}
{ _id:
  ObjectId("4d6574baa6b804ea563c459d")
  , title: "Greenhouse flowers"
}
```

then a product belonging to both categories will look like this:

```
{ _id:
  ObjectId("4d6574baa6b804ea563ca982")
  , name: "Dragon Orchid",
category_ids: [ ObjectId("4d6574baa6b804ea563c132a"),
                ObjectId("4d6574baa6b804ea563c459d") ]
}
```

For efficient queries, you should index the array of category IDs:

```
db.products.createIndex({category_ids: 1})
```

Then, to find all products in the Epiphytes category, match against the category_id field:

```
db.products.find({category_id: ObjectId("4d6574baa6b804ea563c132a")})
```

And to return all category documents related to the Dragon Orchid product, first get the list of that product's category IDs:

```
product = db.products.findOne({_id: ObjectId("4d6574baa6b804ea563c132a")})
```

Then query the categories collection using the $in operator:

```
db.categories.find({_id: {$in: product['category_ids']}})
```

You'll notice that finding the categories requires two queries, whereas the product search takes just one. This optimizes for the common case, as you're more likely to search for products in a category than the other way around.

The strategy there was to store a snapshot of the category's ancestors within each category document. This denormalization makes updates more complicated but greatly simplifies reads.

△ 5 points by kbanker 1 hour ago

Who was Alexander the Great's teacher?

△ 2 points by asophist 1 hour ago

It was definitely Socrates.

△ 10 points by daletheia 1 hour ago

Oh you sophist...It was actually Aristotle!

△ 1 point by seuclid 2 hours ago

So who really discarded the parallel postulate?

Let'sseehowthesecommentslookasdocumentsorganizedwithamaterializedpath. The first is a root-level comment, so the pathis null:

```
{ _id:
  ObjectId("4d692b5d59e212384d95001")
  , depth: 0,
  path: null,
  created: ISODate("2011-02-26T17:18:01.251Z"),
  username: "plotinus",
  body: "Who was Alexander the Great's
  teacher?", thread_id:
  ObjectId("4d692b5d59e212384d95223a")
}
```

The other root-level question, the one by user seuclid, will have the same structure. More illustrative are the follow-up comments to the question about Alexander the Great's teacher. Examine the first of these, and note that pathcontains the _idof the immediate parent:

```
{ _id:
  ObjectId("4d692b5d59e212384d951002"
  ), depth: 1,
  path: "4d692b5d59e212384d95001",


    created: ISODate("2011-02-26T17:21:01.251Z"),
    username: "asophist",
    body: "It was definitely Socrates.",
    thread_id: ObjectId("4d692b5d59e212384d95223a")
  }
```

The next deeper comment's path contains both the IDs of the original and immediate parents, in that order and separated by a colon:

```
{ _id:
  ObjectId("4d692b5d59e212384d95003")
  , depth: 2,
  path:
```

```
  "4d692b5d59e212384d95001:4d692b5d59e212384d951
  002", created: ISODate("2011-02-26T17:21:01.251Z"),
  username: "daletheia",
  body: "Oh you sophist...It was actually Aristotle!",
  thread_id:
  ObjectId("4d692b5d59e212384d95223a")
}
```

At a minimum, you'll want indexes on the thread_id and path fields, as you'll always query on exactly one of these fields:

```
db.comments.createIndex({thread_id: 1})
db.comments.createIndex({path: 1})
```

Now the question is how you go about querying and displaying the tree. One of the advantages of the materialized path pattern is that you query the database only once, whether you're displaying the entire comment thread or only a subtree within the thread. The query for the first of these is straightforward:

```
db.comments.find({thread_id: ObjectId("4d692b5d59e212384d95223a")})
```

The query for a particular subtree is subtler because it uses a prefix query (discussed in Chapter 5):

```
db.comments.find({path: /^4d692b5d59e212384d95001/})
```

The first method, threaded_list, builds a list of all root-level comments and a map that keys parent IDs to lists of child nodes:

```
def threaded_list(cursor, opts={})
  list        = []
  child_map = {}
  start_depth = opts[:start_depth] || 0
  cursor.each do |comment|
    if comment['depth'] ==
      start_depth list.push(comment)
    else
      matches =
      comment['path'].match(/([d|w]+)$/)
```

```
      immediate_parent_id = matches[1]
      if immediate_parent_id
        child_map[immediate_parent_id] ||= []
        child_map[immediate_parent_id] <<
        comment
      en
    d
    end
  end
  assemble(list,
child_map) end
```

The assemblemethod takes the list of root nodes and the child map and then builds a new list in display order:

```
def assemble(comments,
  map) list = []
  comments.each do
    |comment|
    list.push(comment)
    child_comments =
    map[comment['_id'].to_s] if
    child_comments
      list.concat(assemble(child_comments,
    map)) end
  end
  list
end
```

To print the comments, you merely iterate over the list, indenting appropriately for each comment's depth:

```
def print_threaded_list(cursor, opts={})
  threaded_list(cursor, opts).each do |item|
    indent = " " * item['depth']
    puts indent + item['body'] + " #{item['path']}"
  end
end
```

Querying for the comments and printing them is then straightforward:

```
cursor =
@comments.find.sort("created")
print_threaded_list(cursor)
```

Worker queues

You can implement worker queues in MongoDB using either standard or capped collections (discussed in chapter 4). In both cases, the findAndModify command will permit you to process queue entries atomically.

A queue entry requires a state and a timestamp plus any remaining fields to contain the payload. The state can be encoded as a string, but an integer is more space-efficient. We'll use 0 and 1 to indicate *processed* and *unprocessed*, respectively. The timestamp is the standard BSON date. And the payload here is a simple plaintext mes- sage but could be anything in principle:

```
{ state: 0,
  created: ISODate("2011-02-24T16:29:36.697Z"),
  message: "hello world" }
```

You'll need to declare an index that allows you to efficiently fetch the oldest unprocessed entry (FIFO). A compound index on stateand createdfits the bill:

```
db.queue.createIndex({state: 1, created: 1})
```

You then use findAndModifyto return the next entry and mark it as processed:

```
q = {state: 0}
s = {created: 1}
u = {$set: {state: 1}}
db.queue.findAndModify({query: q, sort: s, update: u})
```