

A database for the modern web

This chapter covers

- MongoDB's history, design goals, and key features
- A brief introduction to the shell and drivers
- Use cases and limitations
- Recent changes in MongoDB

If you've built web applications in recent years, you've probably used a relational database as the primary data store. If you're familiar with SQL, you might appreciate the usefulness of a well-normalized¹ data model, the necessity of transactions, and the assurances provided by a durable storage engine. Simply put, the relational database is mature and well-known. When developers start advocating alternative datastores, questions about the viability and utility of these new technologies arise. Are these new datastores replacements for relational database systems? Who's using them in production, and why? What trade-offs are involved in moving

¹ When we mention normalization we're usually talking about reducing redundancy when you store data. For example, in a SQL database you can split parts of your data, such as users and orders, into their own tables to reduce redundant storage of usernames.

to a nonrelational database? The answers to those questions rest on the answer to this one: why are developers interested in MongoDB?

MongoDB is a database management system designed to rapidly develop web applications 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 difficulties 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. Why would you use MongoDB?

Perhaps the biggest reason developers use MongoDB isn't because of its scaling strategy, but because of its intuitive data model. 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.edu'
  ]
}
```

And just like that, you've created an array of email addresses and solved your problem. As a developer, you'll find it extremely useful to be able to store a structured document like this in your database without worrying about fitting a schema or adding more tables when your data changes.

MongoDB's document format is based on JSON, a popular scheme for storing arbitrary 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. For example, suppose you're modeling products for an e-commerce site. With a fully

normalized relational data model, the information for any one product might be divided among dozens of tables. If you want to get a product representation from the database shell, you'll need to write a SQL query full of joins.

With a document model, by contrast, most of a product's information can be represented within a single document. When you open the MongoDB JavaScript shell, you can easily get a comprehensible representation of your product with all its information hierarchically organized in a JSON-like structure. You can also query for it and manipulate it. MongoDB's query capabilities are designed specifically for manipulating structured documents, so users switching from relational databases experience a similar level of query power. In addition, most developers now work with object-oriented languages, and they want a data store that better maps to objects. With MongoDB, an object defined in the programming language can often be persisted as is, removing some of the complexity of object mappers. If you're experienced with relational databases, it can be helpful to approach MongoDB from the perspective of transitioning your existing skills into this new database.

If the distinction between a tabular and object representation of data is new to you, you probably have a lot of questions. Rest assured that by the end of this chapter you'll have a thorough overview of MongoDB's features and design goals. You'll learn the history of MongoDB and take a tour of the database's main features. Next, you'll explore some alternative database solutions in the NoSQL² category and see how MongoDB fits in. Finally, you'll learn where MongoDB works best and where an alternative datastore might be preferable given some of MongoDB's limitations.

MongoDB has been criticized on several fronts, sometimes fairly and sometimes unfairly. Our view is that it's a tool in the developer's toolbox, like any other database, and you should know its limitations and strengths. Some workloads demand relational joins and different memory management than MongoDB provides. On the other hand, the document-based model fits particularly well with some workloads, and the lack of a schema means that MongoDB can be one of the best tools for quickly developing and iterating on an application. Our goal is to give you the information you need to decide if MongoDB is right for you and explain how to use it effectively.

1.1 *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 database, 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

² The umbrella term NoSQL was coined in 2009 to lump together the many nonrelational databases gaining in popularity at the time, one of their commonalities being that they use a query language other than SQL.

want 10gen's new database technology. This led 10gen to concentrate its efforts solely on the database that became MongoDB.

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. Thus, MongoDB, Inc.'s business model isn't unlike that of other well-known open source companies: support the development of an open source product and provide subscription services to end users.

The most important thing to remember from its history is that MongoDB was intended to be an extremely simple, yet flexible, part of a web-application stack. These kinds of use cases have driven the choices made in MongoDB's development and help explain its features.

1.2 **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. This section also explores operations, focusing on MongoDB's flavor of replication and its strategy for scaling horizontally.

1.2.1 **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 example. 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'
  },
},
```

Annotations for Listing 1.1:

- id field, primary key (points to `_id`)
- Tags stored as array of strings (points to `tags`)
- Attribute pointing to another document (points to `image`)

```
comments: [  
  {  
    user: 'bjones',  
    text: 'Interesting article.'  
  },  
  {  
    user: 'sverch',  
    text: 'Color me skeptical!'  
  }  
]  
}
```



← **Comments stored
as array of
comment objects**

3

This listing shows a JSON document representing an article on a social news site (think Reddit or Twitter). As you can see, a *document* is essentially a set of property names and their values. The values can be simple data types, such as strings, numbers, and dates. But these values can also be arrays and even other JSON documents ❷. These latter constructs permit documents to represent a variety of rich data structures. You'll see that the sample document has a property, `tags` ❶, which stores the article's tags in an array. But even more interesting is the `comments` property ❸, which is an array of comment documents.

Internally, MongoDB stores documents in a format called Binary JSON, or BSON. BSON has a similar structure but is intended for storing many documents. When you query MongoDB and get results back, these will be translated into an easy-to-read data structure. The MongoDB shell uses JavaScript and gets documents in JSON, which is what we'll use for most of our examples. We'll discuss the BSON format extensively in later chapters.

Where relational databases have tables, MongoDB has *collections*. In other words, MySQL (a popular relational database) keeps its data in tables of rows, while MongoDB keeps its data in collections of documents, which you can think of as a group of documents. Collections are an important concept in MongoDB. The data in a collection is stored to disk, and most queries require you to specify which collection you'd like to target.

Let's take a moment to compare MongoDB collections to a standard relational database representation of the same data. Figure 1.1 shows a likely relational analog. Because tables are essentially flat, representing the various one-to-many relationships in your post document requires multiple tables. You start with a `posts` table containing the core information for each post. Then you create three other tables, each of which includes a field, `post_id`, referencing the original post. The technique of separating an object's data into multiple tables like this is known as *normalization*. A normalized data set, among other things, ensures that each unit of data is represented in one place only.

But strict normalization isn't without its costs. Notably, some assembly is required. To display the post you just referenced, you'll need to perform a join between the `post` and `comments` tables. Ultimately, the question of whether strict normalization is required depends on the kind of data you're modeling, and chapter 4 will have much more to say about the topic. What's important to note here is that a document-oriented data model naturally represents data in an aggregate form, allowing you to work with an object holistically: all the data representing a post, from comments to tags, can be fitted into a single database object.

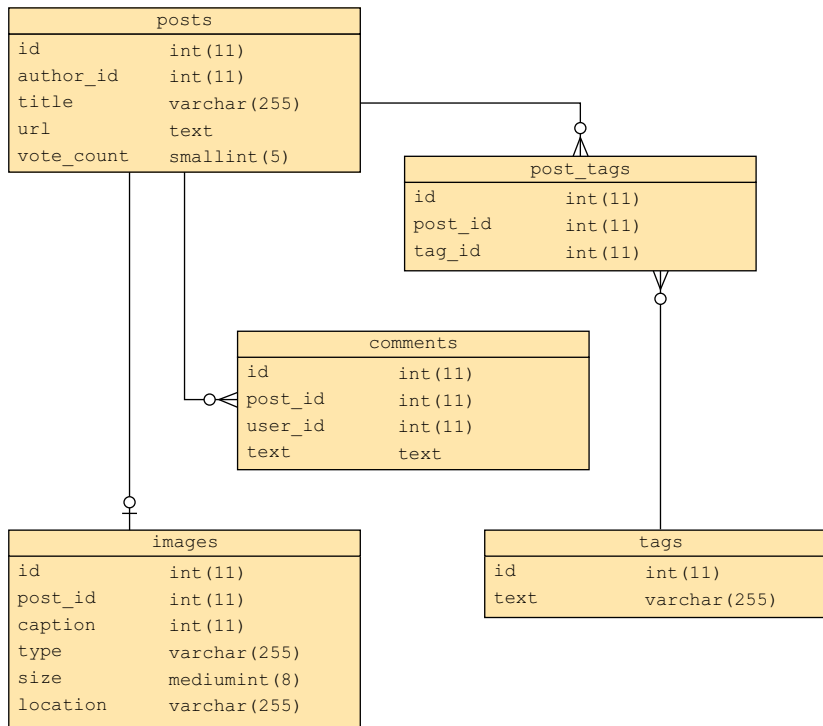


Figure 1.1 A basic relational data model for entries on a social news site. The line terminator that looks like a cross represents a one-to-one relationship, so there's only one row from the **images** table associated with a row from the **posts** table. The line terminator that branches apart represents a one-to-many relationship, so there can be many rows in the **comments** table associated with a row from the **posts** table.

You've probably noticed that in addition to providing a richness of structure, documents needn't conform to a prespecified schema. With a relational database, you store rows in a table. Each table has a strictly defined schema specifying which columns and types are permitted. If any row in a table needs an extra field, you have to alter the table explicitly. MongoDB groups documents into collections, containers that don't impose any sort of schema. In theory, each document in a collection can have a completely different structure; in practice, a collection's document will be relatively uniform. For instance, every document in the **posts** collection will have fields for the title, tags, comments, and so forth.

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 application development when the schema is changing frequently.

Second, and more significantly, a schema-less model allows you to represent data with truly variable properties. For example, imagine you're building an e-commerce

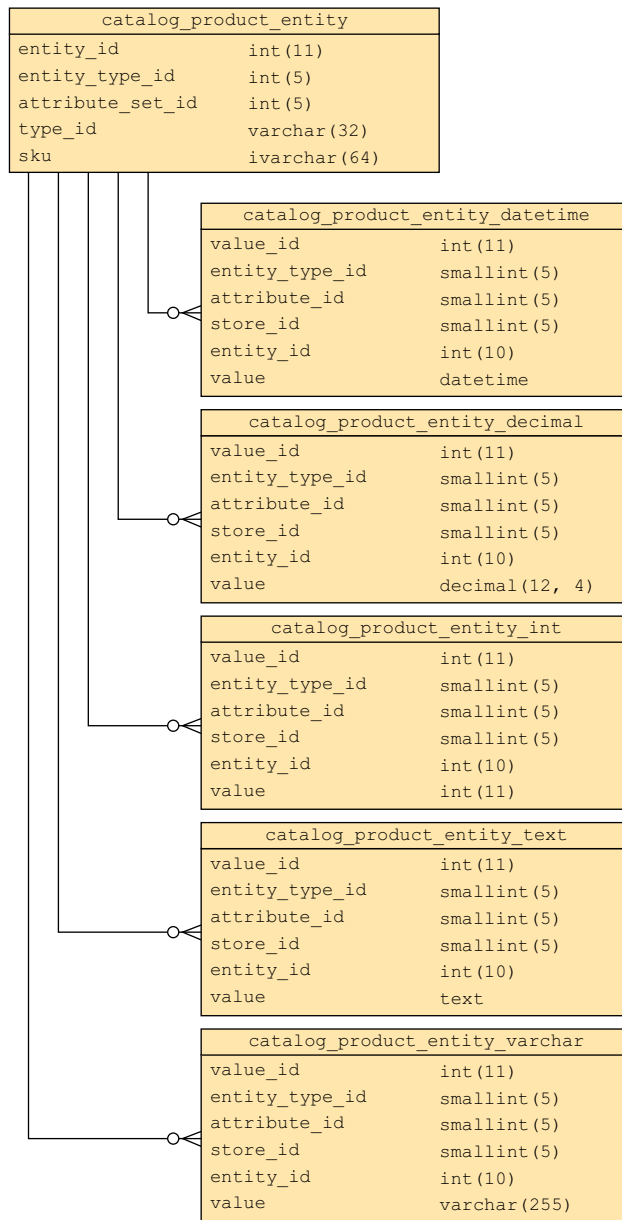


Figure 1.2 A portion of the schema for an e-commerce application. These tables facilitate dynamic attribute creation for products.

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. The traditional way of handling this in a fixed-schema database is to use the entity-attribute-value pattern,³ shown in figure 1.2.

³ For more information see http://en.wikipedia.org/wiki/Entity-attribute-value_model.

What you're seeing is one section of the data model for an e-commerce framework. Note the series of tables that are all essentially the same, except for a single attribute, value, that varies only by data type. This structure allows an administrator to define additional product types and their attributes, but the result is significant complexity. Think about firing up the MySQL shell to examine or update a product modeled in this way; the SQL joins required to assemble the product would be enormously complex. Modeled as a document, no join is required, and new attributes can be added dynamically. Not all relational models are this complex, but the point is that when you're developing a MongoDB application you don't need to worry as much about what data fields you'll need in the future.

1.2.2 *Ad hoc queries*

To say that a system supports *ad hoc queries* is to say that it isn't necessary to define in advance what sorts of queries the system will accept. Relational databases have this property; they'll faithfully execute any well-formed SQL query with any number of conditions. 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. Like many other systems, key-value stores sacrifice rich query power in exchange for a simple scalability model. One of MongoDB's design goals is to preserve most of the query power that's been so fundamental to the relational database world.

To see how MongoDB's query language works, let's take a simple example involving posts and comments. Suppose you want to find all posts tagged with the term *politics* having more than 10 votes. 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 `$gt` key indicates the greater-than condition:

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

Note that the two queries assume a different data model. The SQL query relies on a strictly normalized model, where posts and tags are stored in distinct tables, whereas the MongoDB query assumes that tags are stored within each post document. But both queries demonstrate an ability to query on arbitrary combinations of attributes, which is the essence of ad hoc query ability.

1.2.3 *Indexes*

A critical element of ad hoc queries is that they search for values that you don't know when you create the database. As you add more and more documents to your

database, searching for a value becomes increasingly expensive; it's a needle in an ever-expanding haystack. Thus, you need a way to efficiently search through your data. The solution to this is an index.

The best way to understand database indexes is by analogy: many books have indexes matching keywords to page numbers. Suppose you have a cookbook and want to find all recipes calling for pears (maybe you have a lot of pears and don't want them to go bad). The time-consuming approach would be to page through every recipe, checking each ingredient list for pears. Most people would prefer to check the book's index for the pears entry, which would give a list of all the recipes containing pears. Database indexes are data structures that provide this same service.

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, including 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 production release.

Most databases give each document or row a *primary key*, a unique identifier for that datum. The primary key is generally indexed automatically so that each datum can be efficiently accessed using its unique key, and MongoDB is no different. But not every database allows you to also index the data inside that row or document. These are called *secondary indexes*. Many NoSQL databases, such as HBase, are considered *key-value stores* because they don't allow any secondary indexes. This is a significant feature in MongoDB; by permitting multiple secondary indexes MongoDB allows users to optimize for a wide variety of queries.

With MongoDB, you can create up to 64 indexes per collection. The kinds of indexes supported include all the ones you'd find in an RDBMS; ascending, descending, unique, compound-key, hashed, text, and even geospatial indexes⁴ are supported. Because MongoDB and most RDBMSs use the same data structure for their indexes, advice for managing indexes in both of these systems is similar. You'll begin looking at indexes in the next chapter, and because an understanding of indexing is so crucial to efficiently operating a database, chapter 8 is devoted to the topic.

1.2.4 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.

⁴ Geospatial indexes allow you to efficiently query for latitude and longitude points; they're discussed later in this book.

Replica sets consist of many MongoDB servers, usually with each server on a separate physical machine; we'll call these nodes. At any given time, one node serves as the replica set primary node and one or more nodes serve as secondaries. Like the master-slave replication that you may be familiar with from other databases, a replica set's primary node can accept both reads and writes, but the secondary nodes are read-only. What makes replica sets unique is their support for automated failover: if the primary node fails, the cluster will pick a secondary node and automatically promote it to primary. When the former primary comes back online, it'll do so as a secondary. An illustration of this process is provided in figure 1.3.

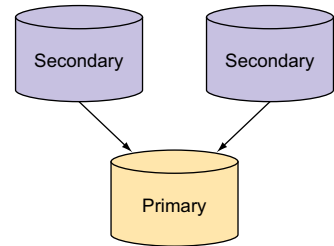
Replication is one of MongoDB's most useful features and we'll cover it in depth later in the book.

1.2.5 Speed and durability

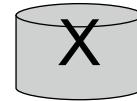
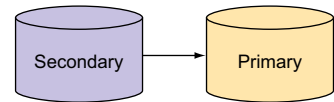
To understand MongoDB's approach to durability, it pays to consider a few ideas first. In the realm of database systems there exists an inverse relationship between write speed and durability. *Write speed* can be understood as the volume of inserts, updates, and deletes that a database can process in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

For instance, suppose you write 100 records of 50 KB each to a database and then immediately cut the power on the server. Will those records be recoverable when you bring the machine back online? The answer depends on your database system, its configuration, and the hardware hosting it. Most databases enable good durability by default, so you're safe if this happens. For some applications, like storing log lines, it might make more sense to have faster writes, even if you risk data loss. The problem is that writing to a magnetic hard drive is orders of magnitude slower than writing to RAM. Certain databases, such as Memcached, write exclusively to RAM, which makes them extremely fast but completely volatile. On the other hand, few databases write exclusively to disk because the low performance of such an operation is unacceptable. Therefore, database designers often need to make compromises to provide the best balance of speed and durability.

1. A working replica set



2. Original primary node fails and a secondary is promoted to primary.



3. Original primary comes back online as a secondary.

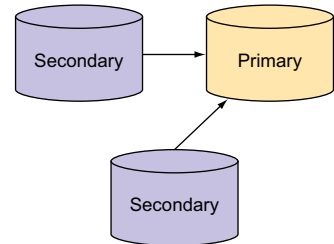


Figure 1.3 Automated failover with a replica set

Transaction logging

One compromise between speed and durability can be seen in MySQL's InnoDB. InnoDB is a transactional storage engine, which by definition must guarantee durability. It accomplishes this by writing its updates in two places: once to a transaction log and again to an in-memory buffer pool. The transaction log is synced to disk immediately, whereas the buffer pool is only eventually synced by a background thread. The reason for this dual write is because generally speaking, random I/O is much slower than sequential I/O. Because writes to the main data files constitute random I/O, it's faster to write these changes to RAM first, allowing the sync to disk to happen later. But some sort of write to disk is necessary to guarantee durability and it's important that the write be sequential, and thus fast; this is what the transaction log provides. In the event of an unclean shutdown, InnoDB can replay its transaction log and update the main data files accordingly. This provides an acceptable level of performance while guaranteeing a high level of durability.

In MongoDB's case, users control the speed and durability trade-off by choosing write semantics and deciding whether to enable journaling. Journaling is enabled by default since MongoDB v2.0. In the drivers released after November 2012 MongoDB safely guarantees that a write has been written to RAM before returning to the user, though this characteristic is configurable. You can configure MongoDB to *fire-and-forget*, sending off a write to the server without waiting for an acknowledgment. You can also configure MongoDB to guarantee that a write has gone to multiple replicas before considering it committed. For high-volume, low-value data (like clickstreams and logs), fire-and-forget-style writes can be ideal. For important data, a safe mode setting is necessary. It's important to know that in MongoDB versions older than 2.0, the unsafe fire-and-forget strategy was set as the default, because when 10gen started the development of MongoDB, it was focusing solely on that data tier and it was believed that the application tier would handle such errors. But as MongoDB was used for more and more use cases and not solely for the web tier, it was deemed that it was too unsafe for any data you didn't want to lose.

Since MongoDB v2.0, journaling is enabled by default. With *journaling*, every write is flushed to the journal file every 100 ms. If the server is ever shut down uncleanly (say, in a power outage), the journal will be used to ensure that MongoDB's data files are restored to a consistent state when you restart the server. This is the safest way to run MongoDB.

It's possible to run the server without journaling as a way of increasing performance for some write loads. The downside is that the data files may be corrupted after an unclean shutdown. As a consequence, anyone planning to disable journaling should run with replication, preferably to a second datacenter, to increase the likelihood that a pristine copy of the data will still exist even if there's a failure.

MongoDB was designed to give you options in the speed-durability tradeoff, but we highly recommend safe settings for essential data. The topics of replication and durability are vast; you'll see a detailed exploration of them in chapter 11.

1.2.6 Scaling

The easiest way to scale most databases is to upgrade the hardware. If your application is running on a single node, it's usually possible to add some combination of faster disks, more memory, and a beefier CPU to ease any database bottlenecks. The technique of augmenting a single node's hardware for scale is known as *vertical scaling*, or *scaling up*. Vertical scaling has the advantages of being simple, reliable, and cost-effective up to a certain point, but eventually you reach a point where it's no longer feasible to move to a better machine.

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 database across multiple machines. A horizontally scaled architecture can run on many smaller, less expensive machines, often reducing your hosting costs. What's more, the distribution of data across machines mitigates the consequences of failure. Machines will unavoidably fail from time to time. If you've scaled vertically and the machine fails, then you need to deal with the failure of a machine on which most of your system depends. This may not be an issue if a copy of the data exists on a replicated slave, but it's still the case that only a single server need fail to bring down the entire system. Contrast that with failure inside a horizontally scaled architecture. This may be less catastrophic because a single machine represents a much smaller percentage of the system as a whole.

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

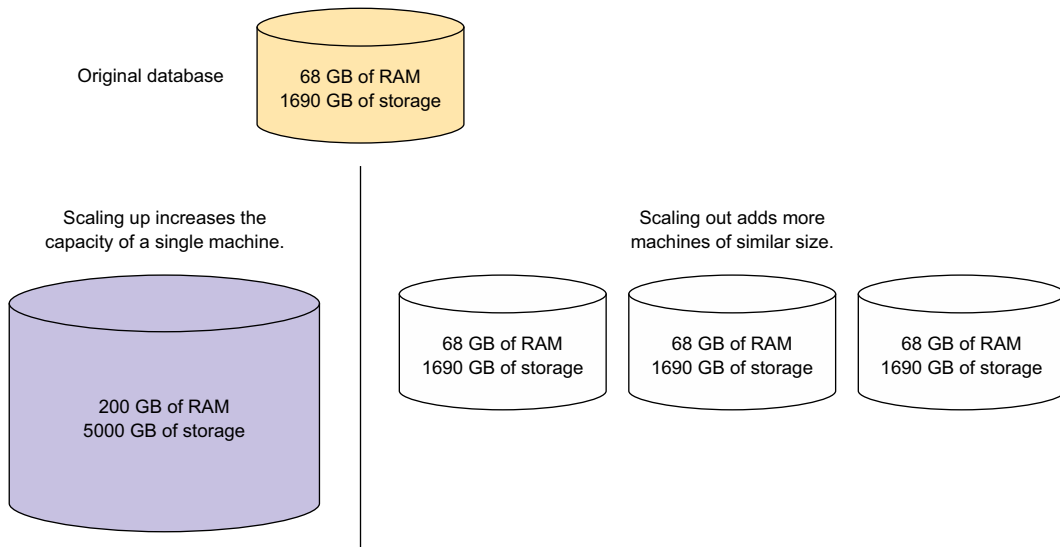


Figure 1.4 Horizontal versus vertical scaling

the distribution of data across nodes. There's also a hash- and tag-based sharding mechanism, but it's just another form of the range-based sharding mechanism.

The sharding system handles the addition of shard nodes, and it also facilitates automatic failover. Individual shards are made up of a replica set consisting of at least two nodes, ensuring automatic recovery with no single point of failure. All this means that no application code has to handle these logistics; your application code communicates with a sharded cluster just as it speaks to a single node. Chapter 12 explores sharding in detail.

You've seen a lot of MongoDB's most compelling features; in chapter 2, you'll begin to see how some of them work in practice. But at this point, let's take a more pragmatic look at the database. In the next section, you'll look at MongoDB in its environment, the tools that ship with the core server, and a few ways of getting data in and out.

1.3 *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). The source code is freely available on GitHub, and contributions from the community are frequently accepted. But the project is guided by the MongoDB, Inc. core server team, and the overwhelming majority of commits come from this group.

About the GNU-AGPL

The GNU-AGPL is the subject of some controversy. In practice, this licensing means that the source code is freely available and that contributions from the community are encouraged. But GNU-AGPL requires that any modifications made to the source code must be published publicly for the benefit of the community. This can be a concern for companies that want to modify MongoDB but don't want to publish these changes to others. For companies wanting to safeguard their core server enhancements, MongoDB, Inc. provides special commercial licenses.

MongoDB v1.0 was released in November 2009. Major releases appear approximately once every three months, with even point numbers for stable branches and odd numbers for development. As of this writing, the latest stable release is v3.0.⁵

What follows is an overview of the components that ship with MongoDB along with a high-level description of the tools and language drivers for developing applications with the database.

⁵ You should always use the latest stable point release; for example, v3.0.6. Check out the complete installation instructions in appendix A.

1.3.1 Core server

The core database server runs via an executable called `mongod` (`mongod.exe` on Windows). The `mongod` server process receives commands over a network socket using a custom binary protocol. All the data files for a `mongod` process are stored by default in `/data/db` on Unix-like systems and in `c:\data\db` on Windows. Some of the examples in this text may be more Linux-oriented. Most of our MongoDB production servers are run on Linux because of its reliability, wide adoption, and excellent tools.

`mongod` can be run in several modes, such as a standalone server or a member of a replica set. Replication is recommended when you're running MongoDB in production, and you generally see replica set configurations consisting of two replicas plus a `mongod` running in arbiter mode. When you use MongoDB's sharding feature, you'll also run `mongod` in config server mode. Finally, a separate routing server exists called `mongos`, which is used to send requests to the appropriate shard in this kind of setup. Don't worry too much about all these options yet; we'll describe each in detail in the replication (11) and sharding (12) chapters.

Configuring a `mongod` process is relatively simple; it can be accomplished both with command-line arguments and with a text configuration file. Some common configurations to change are setting the port that `mongod` listens on and setting the directory where it stores its data. To see these configurations, you can run `mongod --help`.

1.3.2 JavaScript shell

The MongoDB command shell is a JavaScript⁶-based tool for administering the database and manipulating data. The `mongo` executable loads the shell and connects to a specified `mongod` process, or one running locally by default. The shell was developed to be similar to the MySQL shell; the biggest differences are that it's based on JavaScript and SQL isn't used. For instance, you can pick your database and then insert a simple document into the `users` collection like this:

```
> use my_database
> db.users.insert({name: "Kyle"})
```

The first command, indicating which database you want to use, will be familiar to users of MySQL. The second command is a JavaScript expression that inserts a simple document. To see the results of your insert, you can issue a simple query:

```
> db.users.find()
{ _id: ObjectId("4ba667b0a90578631c9caea0"), name: "Kyle" }
```

⁶ If you'd like an introduction or refresher to JavaScript, a good resource is <http://eloquentjavascript.net>. JavaScript has a syntax similar to languages like C or Java. If you're familiar with either of those, you should be able to understand most of the JavaScript examples.

The `find` method returns the inserted document, with an object ID added. All documents require a primary key stored in the `_id` field. You're allowed to enter a custom `_id` as long as you can guarantee its uniqueness. But if you omit the `_id` altogether, a MongoDB object ID will be inserted automatically.

In addition to allowing you to insert and query for data, the shell permits you to run administrative commands. Some examples include viewing the current database operation, checking the status of replication to a secondary node, and configuring a collection for sharding. As you'll see, the MongoDB shell is indeed a powerful tool that's worth getting to know well.

All that said, the bulk of your work with MongoDB will be done through an application written in a given programming language. To see how that's done, we must say a few things about MongoDB's language drivers.

1.3.3 Database drivers

If the notion of a database driver conjures up nightmares of low-level device hacking, don't fret; the MongoDB drivers are easy to use. The driver is the code used in an application to communicate with a MongoDB server. All drivers have functionality to query, retrieve results, write data, and run database commands. Every effort has been made to provide an API that matches the idioms of the given language while also maintaining relatively uniform interfaces across languages. For instance, all of the drivers implement similar methods for saving a document to a collection, but the representation of the document itself will usually be whatever is most natural to each language. In Ruby, that means using a Ruby hash. In Python, a dictionary is appropriate. And in Java, which lacks any analogous language primitive, you usually represent documents as a `Map` object or something similar. Some developers like using an object-relational mapper to help manage representing their data this way, but in practice, the MongoDB drivers are complete enough that this isn't required.

Language drivers

As of this writing, MongoDB, Inc. officially supports drivers for C, C++, C#, Erlang, Java, Node.js, JavaScript, Perl, PHP, Python, Scala, and Ruby—and the list is always growing. If you need support for another language, there are probably community-supported drivers for it, developed by MongoDB users but not officially managed by MongoDB, Inc., most of which are pretty good. If no community-supported driver exists for your language, specifications for building a new driver are documented at <http://mongodb.org>. Because all of the officially supported drivers are used heavily in production and provided under the Apache license, plenty of good examples are freely available for would-be driver authors.

Beginning in chapter 3, we describe how the drivers work and how to use them to write programs.

1.3.4 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 TSV⁷ 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. We'll cover BSON in much more detail in chapter 2.

1.4 Why MongoDB?

You've seen a few reasons why MongoDB might be a good choice for your projects. Here, we'll make this more explicit, first by considering the overall design objectives of the MongoDB project. According to its creators, MongoDB was designed to combine the best features of key-value stores and relational databases. Because of their simplicity, key-value stores are extremely fast and relatively easy to scale. Relational databases are more difficult to scale, at least horizontally, but have a rich data model and a powerful query language. MongoDB is intended to be a compromise between these two designs, with useful aspects of both. The end goal is for it to be a database that scales easily, stores rich data structures, and provides sophisticated query mechanisms.

⁷ CSV stands for Comma-Separated Values, meaning data split into multiple fields, which are separated by commas. This is a popular format for representing tabular data, since column names and many rows of values can be listed in a readable file. TSV stands for Tab-Separated Values—the same format with tabs used instead of commas.

In terms of use cases, MongoDB is well-suited as a primary datastore for web applications, analytics and logging applications, and any application requiring a medium-grade cache. In addition, because it easily stores schema-less data, MongoDB is also good for capturing data whose structure can't be known in advance.

The preceding claims are bold. To substantiate them, we're going to take a broad look at the varieties of databases currently in use and contrast them with MongoDB. Next, you'll see some specific MongoDB use cases as well as examples of them in production. Then, we'll discuss some important practical considerations for using MongoDB.

1.4.1 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 categories. 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. Memcached can scale across nodes, converting all available RAM into a single, monolithic datastore.	Caching. Web ops.
Sophisticated key-value stores	HBase, Cassandra, Riak KV, Redis, CouchDB	Variable. Cassandra uses a key-value structure known as a <i>column</i> . 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 databases	Oracle Database, IBM DB2, Microsoft SQL Server, MySQL, PostgreSQL	Tables.	Vertical scaling. Limited support for clustering and manual partitioning.	System requiring transactions (banking, finance) or SQL. Normalized data model.

SIMPLE KEY-VALUE STORES

Simple key-value stores do what their name implies: they index values based on a supplied key. A common use case is caching. For instance, suppose you needed to cache an HTML page rendered by your app. The key in this case might be the page's URL, and the value would be the rendered HTML itself. Note that as far as a key-value store

is concerned, the value is an opaque byte array. There's no enforced schema, as you'd find in a relational database, nor is there any concept of data types. This naturally limits the operations permitted by key-value stores: you can insert a new value and then use its key either to retrieve that value or delete it. Systems with such simplicity are generally fast and scalable.

The best-known simple key-value store is *Memcached*, which stores its data in memory only, so it trades durability for speed. It's also distributed; Memcached nodes running across multiple servers can act as a single datastore, eliminating the complexity of maintaining cache state across machines.

Compared with MongoDB, a simple key-value store like Memcached will often allow for faster reads and writes. But unlike MongoDB, these systems can rarely act as primary datastores. Simple key-value stores are best used as adjuncts, either as caching layers atop a more traditional database or as simple persistence layers for ephemeral services like job queues.

SOPHISTICATED KEY-VALUE STORES

It's possible to refine the simple key-value model to handle complicated read/write schemes or to provide a richer data model. In these cases, you end up with what we'll term a sophisticated key-value store. One example is Amazon's Dynamo, described in a widely studied white paper titled "Dynamo: Amazon's Highly Available Key-Value Store" (<http://allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>). The aim of Dynamo is to be a database robust enough to continue functioning in the face of network failures, datacenter outages, and similar disruptions. This requires that the system always be read from and written to, which essentially requires that data be automatically replicated across multiple nodes. If a node fails, a user of the system—perhaps in this case a customer with an Amazon shopping cart—won't experience any interruptions in service. Dynamo provides ways of resolving the inevitable conflicts that arise when a system allows the same data to be written to multiple nodes. At the same time, Dynamo is easily scaled. Because it's masterless—all nodes are equal—it's easy to understand the system as a whole, and nodes can be added easily. Although Dynamo is a proprietary system, the ideas used to build it have inspired many systems falling under the NoSQL umbrella, including Cassandra, HBase, and Riak KV.

By looking at who developed these sophisticated key-value stores, and how they've been used in practice, you can see where these systems shine. Let's take Cassandra, which implements many of Dynamo's scaling properties while providing a column-oriented data model inspired by Google's BigTable. Cassandra is an open source version of a datastore built by Facebook for its inbox search feature. The system scales horizontally to index more than 50 TB of inbox data, allowing for searches on inbox keywords and recipients. Data is indexed by user ID, where each record consists of an array of search terms for keyword searches and an array of recipient IDs for recipient searches.⁸

⁸ See "Cassandra: A Decentralized Structured Storage System," at <http://mng.bz/5321>.

These sophisticated key-value stores were developed by major internet companies such as Amazon, Google, and Facebook to manage cross-sections of systems with extraordinarily large amounts of data. In other words, sophisticated key-value stores manage a relatively self-contained domain that demands significant storage and availability. Because of their masterless architecture, these systems scale easily with the addition of nodes. They opt for eventual consistency, which means that reads don't necessarily reflect the latest write. But what users get in exchange for weaker consistency is the ability to write in the face of any one node's failure.

This contrasts with MongoDB, which provides strong consistency, a rich data model, and secondary indexes. The last two of these attributes go hand in hand; key-value stores can generally store any data structure in the value, but the database is unable to query them unless these values can be indexed. You can fetch them with the primary key, or perhaps scan across all of the keys, but the database is useless for querying these without secondary indexes.

RELATIONAL DATABASES

Much has already been said of relational databases in this introduction, so in the interest of brevity, we need only discuss what RDBMSs (Relational Database Management Systems) have in common with MongoDB and where they diverge. 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 relational databases are both capable of representing a rich data model. Where relational databases use fixed-schema tables, MongoDB has schema-free documents. Most relational databases support secondary indexes and aggregations.

Perhaps the biggest defining feature of relational databases from the user's perspective is the use of SQL as a query language. SQL is a powerful tool for working with data; it's not perfect for every job, but in some cases it's more expressive and easier to work with than MongoDB's query language. Additionally, SQL is fairly portable between databases, though each implementation has its own quirks. One way to think about it is that SQL may be easier for a data scientist or full-time analyst who writes queries to explore data. MongoDB's query language is aimed more at developers, who write a query once to embed it in their application. Both models have their strengths and weaknesses, and sometimes it comes down to personal preference.

There are also many relational databases intended for analytics (or as a "data warehouse") rather than as an application database. Usually data is imported in bulk to these platforms and then queried by analysts to answer business-intelligence questions. This area is dominated by enterprise vendors with HP Vertica or Teradata Database, which both offer horizontally scalable SQL databases.

There is also growing interest in running SQL queries over data stored in Hadoop. Apache Hive is a widely used tool that translates a SQL query into a Map-Reduce job, which offers a scalable way of querying large data sets. These queries use the relational model, but are intended only for slower analytics queries, not for use inside an application.

DOCUMENT DATABASES

Few databases identify themselves as document databases. As of this writing, the closest 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. CouchDB doesn't partition data across machines; rather, each CouchDB node is a complete replica of every other.

1.4.2 Use cases and production deployments

Let's be honest. You're not going to choose a database solely on the basis of its features. You need to know that real businesses are using it successfully. Let's look at a few broadly defined use cases for MongoDB and some examples of its use in production.⁹

WEB APPLICATIONS

MongoDB is well-suited as a primary datastore for web applications. Even a simple web application will require numerous data models for managing users, sessions, app-specific data, uploads, and permissions, to say nothing of the overarching domain. Just as this aligns well with the tabular approach provided by relational databases, so too it benefits from MongoDB's collection and document model. And because documents can represent rich data structures, the number of collections needed will usually be less than the number of tables required to model the same data using a fully normalized relational model. In addition, dynamic queries and secondary indexes allow for the easy implementation of most queries familiar to SQL developers. Finally, as a web application grows, MongoDB provides a clear path for scale.

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. What's interesting in this case is that in addition to handling the site's main content (posts, comments, users, and so on), MongoDB processes and stores real-time analytics data. These analytics are used by TBI to generate dynamic heat maps indicating click-through rates for the various news stories.

AGILE DEVELOPMENT

Regardless of what you may think about the agile development movement, it's hard to deny the desirability of building an application quickly. A number of development teams, including those from Shutterfly and The New York Times, have chosen MongoDB in part because they can develop applications much more quickly on it than on relational databases. One obvious reason for this is that MongoDB has no fixed schema, so all the time spent committing, communicating, and applying schema changes is saved.

⁹ For an up-to-date list of MongoDB production deployments, see <http://mng.bz/z2CH>.

In addition, less time need be spent shoehorning the relational representation of data into an object-oriented data model or dealing with the vagaries and optimizing the SQL produced by object-relational mapping (ORM) technology. Thus, MongoDB often complements projects with shorter development cycles and agile, mid-sized teams.

ANALYTICS AND LOGGING

We alluded earlier to the idea that MongoDB works well for analytics and logging, and the number of applications using MongoDB for these is growing. Often, a well-established company will begin its forays into the MongoDB world with special apps dedicated to analytics. Some of these companies include GitHub, Disqus, Justin.tv, and Gilt Groupe, among others.

MongoDB's relevance to analytics derives from its speed and from two key features: targeted atomic updates and capped collections. Atomic updates let clients efficiently increment counters and push values onto arrays. Capped collections are useful for logging because they store only the most recent documents. Storing logging data in a database, as compared with the filesystem, provides easier organization and greater query power. Now, instead of using `grep` or a custom log search utility, users can employ the MongoDB query language to examine log output.

CACHING

Many web-applications use a layer of caching to help deliver content faster. A data model that allows for a more holistic representation of objects (it's easy to shove a document into MongoDB without worrying much about the structure), combined with faster average query speeds, frequently allows MongoDB to be run as a cache with richer query capabilities, or to do away with the caching layer all together. The Business Insider, for example, was able to dispense with Memcached, serving page requests directly from MongoDB.

VARIABLE SCHEMAS

You can get some sample JSON data from <https://dev.twitter.com/rest/tools/console>, provided that you know how to use it. After getting the data and saving it as `sample.json`, you can import it to MongoDB as follows:

```
$ cat sample.json | mongoimport -c tweets
2015-08-28T11:48:27.584+0300      connected to: localhost
2015-08-28T11:48:27.660+0300      imported 1 document
```

Here you're pulling down a small sample of a Twitter stream and piping that directly into a MongoDB collection. Because the stream produces JSON documents, there's no need to alter the data before sending it to the database. The `mongoimport` tool directly translates the data to BSON. This means that each tweet is stored with its structure intact, as a separate document in the collection. This makes it easy to index and query its content with no need to declare the structure of the data in advance.

If your application needs to consume a JSON API, then having a system that so easily translates JSON is invaluable. It's difficult to know the structure of your data before you store it, and MongoDB's lack of schema constraints may simplify your data model.

1.5 *Tips and limitations*

For all these good features, it's worth keeping in mind a system's trade-offs and limitations. We'd like to note some limitations before you start building a real-world application on MongoDB and running it in production. Many of these are consequences of how MongoDB manages data and moves it between disk and memory in memory-mapped files.

First, MongoDB should usually be run on 64-bit machines. The processes in a 32-bit system are only capable of addressing 4 GB of memory. This means that as soon as your data set, including metadata and storage overhead, hits 4 GB, MongoDB will no longer be able to store additional data. Most production systems will require more than this, so a 64-bit system will be necessary.¹⁰

A second consequence of using virtual memory mapping is that memory for the data will be allocated automatically, as needed. This makes it trickier to run the database in a shared environment. As with database servers in general, MongoDB is best run on a dedicated server.

Perhaps the most important thing to know about MongoDB's use of memory-mapped files is how it affects data sets that exceed the size of available RAM. When you query such a data set, it often requires a disk access for data that has been swapped out of memory. The consequence is that many users report excellent MongoDB performance until the working set of their data exceeds memory and queries slow significantly. This problem isn't exclusive to MongoDB, but it's a common pitfall and something to watch.

A related problem is that the data structures MongoDB uses to store its collections and documents aren't terribly efficient from a data-size perspective. For example, MongoDB stores the document keys in each document. This means that every document with a field named 'username' will use 8 bytes to store the name of the field.

An oft-cited pain-point with MongoDB from SQL developers is that its query language isn't as familiar or easy as writing SQL queries, and this is certainly true in some cases. MongoDB has been more explicitly targeted at developers—not analysts—than most databases. Its philosophy is that a query is something you write once and embed in your application. As you'll see, MongoDB queries are generally composed of JSON objects rather than text strings as in SQL. This makes them simpler to create and parse programmatically, which can be an important consideration, but may be more difficult to change for ad-hoc queries. If you're an analyst who writes queries all day, you'll probably prefer working with SQL.

Finally, it's worth mentioning that although MongoDB is one of the simplest databases to run locally as a single node, there's a maintenance cost to running a large cluster. This is true of most distributed databases, but it's acute with MongoDB because it requires a cluster of three configuration nodes and handles replication separately

¹⁰ 64-bit architectures can theoretically address up to 16 exabytes of memory, which is for all intents and purposes unlimited.

with sharding. In some databases, such as HBase, data is grouped into shards that can be replicated on any machine of the cluster. MongoDB instead allows shards of replica sets, meaning that a piece of data is replicated only within its replica set. Keeping sharding and replication as separate concepts has certain advantages, but also means that each must be configured and managed when you set up a MongoDB cluster.

Let's have a quick look at the other changes that have happened in MongoDB.

1.6 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 edition, 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 differs 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 databases to enable journaling. Journaling is an important function that prevents data corruption.
- *\$and queries*—This version added the \$and query operator to complement the \$or operator.
- *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 relevant 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.

¹¹ MongoDB actually had a version jump from 2.6 straight to 3.0, skipping 2.8. See <http://www.mongodb.com/blog/post/announcing-mongodb-30> for more details about v3.0.

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 difficult 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 operations 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 version. 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.

- *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 duplicate data that was left behind after a chunk migration can be cleaned up automatically.
- *Security improvements*—Collection-level access control is supported in this version, 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.

1.7 Additional resources

This text is intended to be both a tutorial and a reference, so much of the language is intended to introduce readers to new subjects and then describe these subjects in more detail. If you're looking for a pure reference, the best resource is the MongoDB user's manual, available at <http://docs.mongodb.org/manual>. This is an in-depth guide to the database, which will be useful if you need to review a subject, and we highly recommend it.

If you have a specific problem or question about MongoDB, it's likely that someone else has as well. A simple web-search will usually return results about it from resources like blog posts or from Stack Overflow (<http://stackoverflow.com>), a tech-oriented

question and answer site. These are invaluable when you get stuck, but double-check that the answer applies to your version of MongoDB.

You can also get help in places like the MongoDB IRC chat or user forums. MongoDB, Inc. also offers consulting services intended to help make MongoDB easy to use in an enterprise environment. Many cities have their own MongoDB user groups, organized through sites like <http://meetup.com>. These are often a good way to meet folks knowledgeable about MongoDB and learn about how others are using the database. Finally, you can contact us (the authors) directly at the Manning forums, which have a space specifically for MongoDB in Action <http://manning-sandbox.com/forum.jspa?forumID=677>. This is a space to ask in-depth questions that might not be covered in the text and point out omissions or errata. Please don't hesitate to post a question!

1.8 Summary

We've covered a lot. To summarize, MongoDB is an open source, document-based database management system. Designed for the data and scalability requirements of modern internet applications, MongoDB features dynamic queries and secondary indexes, fast atomic updates and complex aggregations, and support for replication with automatic failover and sharding for scaling horizontally.

That's a mouthful, but if you've read this far, you should have a good feel for all these capabilities. You're probably itching to code. It's one thing to talk about a database's features, but another to use the database in practice. Fortunately, that's what you'll do in the next two chapters. First, you'll get acquainted with the MongoDB JavaScript shell, which is incredibly useful for interacting with the database. Then, in chapter 3 you'll start experimenting with the driver and build a simple MongoDB-based application in Ruby.

MongoDB through the JavaScript shell



This chapter covers

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

The previous chapter hinted at the experience of running MongoDB. If you're ready for a more hands-on introduction, this is it. Using the MongoDB shell, this chapter teaches the database's basic concepts through a series of exercises. You'll learn how to create, read, update, and delete (CRUD) documents and, in the process, get to know MongoDB's query language. In addition, we'll take a preliminary look at database indexes and how they're used to optimize queries. Then we'll explore some basic administrative commands and suggest a few ways of getting help as you continue working with MongoDB's shell. Think of this chapter as both an elaboration of the concepts already introduced and as a practical tour of the most common tasks performed from the MongoDB shell.

The MongoDB shell is the go-to tool for experimenting with the database, running ad-hoc queries, and administering running MongoDB instances. When you're writing an application that uses MongoDB, you'll use a language driver (like

MongoDB's Ruby gem) rather than the shell, but the shell is likely where you'll test and refine these queries. Any and all MongoDB queries can be run from the shell.

If you're completely new to MongoDB's shell, know that it provides all the features that you'd expect of such a tool; it allows you to examine and manipulate data and administer the database server itself. MongoDB's shell differs from others, however, in its query language. Instead of employing a standardized query language such as SQL, you interact with the server using the JavaScript programming language and a simple API. This means that you can write JavaScript scripts in the shell that interact with a MongoDB database. If you're not familiar with JavaScript, rest assured that only a superficial knowledge of the language is necessary to take advantage of the shell, and all examples in this chapter will be explained thoroughly. The MongoDB API in the shell is similar to most of the language drivers, so it's easy to take queries you write in the shell and run them from your application.

You'll benefit most from this chapter if you follow along with the examples, but to do that, you'll need to have MongoDB installed on your system. You'll find installation instructions in appendix A.

2.1 *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.

You'll begin by getting the shell up and running. Then you'll see how JavaScript represents documents, and you'll learn how to insert these documents into a MongoDB collection. To verify these inserts, you'll practice querying the collection. Then it's on to updates. Finally, we'll finish out the CRUD operations by learning to remove data and drop collections.

2.1.1 *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
>
```

Figure 2.1 MongoDB JavaScript shell on startup

If you know some JavaScript, you can start entering code and exploring the shell right away. In either case, read on to see how to run your first operations against MongoDB.

2.1.2 Databases, collections, and documents

As you probably know by now, MongoDB stores its information in documents, which can be printed out in JSON (JavaScript Object Notation) format. You'd probably like to store different types of documents, like users and orders, in separate places. This means that MongoDB needs a way to group documents, similar to a table in an RDBMS. In MongoDB, this is called a *collection*.

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 database (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
```

You'll see a message verifying that you've switched databases.

Why does MongoDB have both databases and collections? The answer lies in how MongoDB writes its data out to disk. All collections in a database are grouped in the same files, so it makes sense, from a memory perspective, to keep related collections in the same database. You might also want to have different applications access the same collections (multitenancy) and, it's also useful to keep your data organized so you're prepared for future requirements.

On creating databases and collections

You may be wondering how you can switch to the tutorial database without explicitly creating it. In fact, creating the database isn't required. Databases and collections are created only when documents are first inserted. This behavior is consistent with MongoDB's dynamic approach to data; just as the structure of documents needn't be defined in advance, individual collections and databases can be created at run-time. This can lead to a simplified and accelerated development process. That said, if you're concerned about databases or collections being created accidentally, most of the drivers let you enable a *strict mode* to prevent such careless errors.

It's time to create your first document. Because you're using a JavaScript shell, your documents will be specified in JSON. For instance, a simple document describing a user might look like this:

```
{username: "smith"}
```

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

2.1.3 *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 `users` collection. 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 tutorial 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" }
```

_ID FIELDS IN MONGODB

Note that an `_id` field has been added to the document. You can think of the `_id` value as the document's primary key. Every MongoDB document requires an `_id`, and if one isn't present when the document is created, a special MongoDB `ObjectId` will be generated and added to the document at that time. The `ObjectId` that appears in your console won't be the same as the one in the code listing, but it will be unique among all `_id` values in the collection, which is the only requirement for the field. You can set your own `_id` by setting it in the document you insert, the `ObjectId` is just MongoDB's default.

We'll have more to say about `ObjectIDs` in the next chapter. Let's continue for now by adding a second user to the collection:

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

There should now be two documents in the collection. Go ahead and verify this by running the `count` command:

```
> db.users.count()
2
```

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 documents 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" }
```

The query predicate `{username: "jones"}` returns all documents where the username is `jones`—it literally matches against the existing documents.

Note that calling the `find` method without any argument is equivalent to passing in an empty predicate; `db.users.find()` is the same as `db.users.find({})`.

You can also specify multiple fields in the query predicate, which creates an implicit AND among the fields. For example, you query with the following selector:

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

The three dots after the first line of the query are added by the MongoDB shell to indicate that the command takes more than one line.

The query predicate is identical to the returned document. The predicate ANDs the fields, so this query searches for a document that matches on both the `_id` and `username` fields.

You can also use MongoDB's `$and` operator explicitly. The previous query is identical to

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

Selecting documents with an OR is similar: just use the `$or` operator. Consider the following query:

```
> db.users.find({ $or: [
... { username: "smith" },
```

```
... { username: "jones" }
... ]})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

The query returns both the smith and jones documents, because we asked for either a username of smith or a username of jones.

This example is different than previous ones, because it doesn't just insert or search for a specific document. Rather, the query itself is a document. The idea of representing commands as documents is used often in MongoDB and may come as a surprise if you're used to relational databases. One advantage of this interface is that it's easier to build queries programmatically in your application because they're documents rather than a long SQL string.

We've presented the basics of creating and reading data. Now it's time to look at how to update that data.

2.1.4 **Updating documents**

All updates require at least two arguments. The first specifies which documents to update, and the second defines how the selected documents should be modified. The first few examples demonstrate modifying a single document, but the same operations can be applied to many documents, even an entire collection, as we show at the end of this section. But keep in mind that by default the `update()` method updates a single document.

There are two general types of updates, with different properties and use cases. One type of update involves applying modification operations to a document or documents, and the other type involves replacing the old document with a new one.

For the following examples, we'll look at this sample document:

```
> 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 specified 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 })
```

This update tells MongoDB to find a document where the username is smith, and then to set the value of the country property to Canada. You see the change reflected

in the message that gets sent back by the server. If you now issue a query, you'll see that the document has been updated accordingly:

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

REPLACEMENT UPDATE

Another way to update a document is to replace it rather than just set a field. This is sometimes mistakenly used when an operator update with a `$set` was intended. Consider a slightly different update command:

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

In this case, the document is replaced with one that only contains the `country` field, and the `username` field is removed because the first document is used only for matching and the second document is used for replacing the document that was previously matched. You should be careful when you use this kind of update. A query for the document yields the following:

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

The `_id` is the same, yet data has been *replaced* in the update. Be sure to use the `$set` operator if you intend to add or set fields rather than to replace the entire document. Add the `username` back to the record:

```
> db.users.update({country: "Canada"}, {$set: {username: "smith"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.users.find({country: "Canada"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "country" : "Canada",
  "username" : "smith" }
```

If you later decide that the `country` stored in the profile is no longer needed, the value can be removed as easily using the `$unset` operator:

```
> db.users.update({username: "smith"}, {$unset: {country: 1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

UPDATING COMPLEX DATA

Let's enrich this example. You're representing your data with documents, which, as you saw in chapter 1, can contain complex data structures. Let's suppose that, in addition to storing profile information, your users can store lists of their favorite things. A good document representation might look something like this:

```
{
  username: "smith",
  favorites: {
```

```

    cities: ["Chicago", "Cheyenne"],
    movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
  }
}

```

The favorites key points to an object containing two other keys, which point to lists of favorite cities and movies. Given what you know already, can you think of a way to modify the original smith document to look like this? The \$set operator should come to mind:

```

> db.users.update( {username: "smith"},
...   {
...     $set: {
...       favorites: {
...         cities: ["Chicago", "Cheyenne"],
...         movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
...       }
...     }
...   })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

```

Please note that the use of spacing for indenting isn't mandatory, but it helps avoid errors as the document is more readable this way.

Let's modify jones similarly, but in this case you'll only add a couple of favorite movies:

```

> db.users.update( {username: "jones"},
...   {
...     $set: {
...       favorites: {
...         movies: ["Casablanca", "Rocky"]
...       }
...     }
...   })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

```

If you make a typo, you can use the up arrow key to recall the last shell statement.

Now query the users collection to make sure that both updates succeeded:

```

> > db.users.find().pretty()
{
  "_id" : ObjectId("552e458158cd52bcb257c324"),
  "username" : "smith",
  "favorites" : {
    "cities" : [
      "Chicago",
      "Cheyenne"
    ],
    "movies" : [
      "Casablanca",
      "For a Few Dollars More",
      "The Sting"
    ]
  }
}

```

```
{
  "_id" : ObjectId("552e542a58cd52bcb257c325"),
  "username" : "jones",
  "favorites" : {
    "movies" : [
      "Casablanca",
      "Rocky"
    ]
  }
}
```

Strictly speaking, the `find()` command returns a *cursor* to the returning documents. Therefore, to access the documents you'll need to iterate the cursor. The `find()` command automatically returns 20 documents—if they're available—after iterating the cursor 20 times.

With a couple of example documents at your fingertips, you can now begin to see the power of MongoDB's query language. In particular, the query engine's ability to reach into nested inner objects and match against array elements proves useful in this situation. Notice how we appended the `pretty` operation to the `find` operation to get nicely formatted results returned by the server. Strictly speaking, `pretty()` is actually `cursor.pretty()`, which is a way of configuring a cursor to display results in an easy-to-read format.

You can see an example of both of these concepts demonstrated in this query to find all users who like the movie *Casablanca*:

```
> db.users.find({"favorites.movies": "Casablanca"})
```

The dot between `favorites` and `movies` instructs the query engine to look for a key named `favorites` that points to an object with an inner key named `movies` and then to match the value of the inner key. Thus, this query will return both user documents because queries on arrays will match if any element in the array matches the original query.

To see a more involved example, suppose you know that any user who likes *Casablanca* also likes *The Maltese Falcon* and that you want to update your database to reflect this fact. How would you represent this as a MongoDB update?

MORE ADVANCED UPDATES

You could conceivably use the `$set` operator again, but doing so would require you to rewrite and send the entire array of movies. Because all you want to do is to add an element to the list, you're better off using either `$push` or `$addToSet`. Both operators add an item to an array, but the second does so uniquely, preventing a duplicate addition. This is the update you're looking for:

```
> db.users.update( {"favorites.movies": "Casablanca"},
...   {$addToSet: {"favorites.movies": "The Maltese Falcon"} },
...   false,
...   true )
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })
```

Most of this should be decipherable by now. The first argument is a query predicate that matches against users who have *Casablanca* in their movies list. The second argument adds *The Maltese Falcon* to that list using the `$addToSet` operator.

The third argument, `false`, controls whether an upsert is allowed. This tells the update operation whether it should insert a document if it doesn't already exist, which has different behavior depending on whether the update is an operator update or a replacement update.

The fourth argument, `true`, indicates that this is a multi-update. By default, a MongoDB update operation will apply only to the first document matched by the query selector. If you want the operation to apply to all documents matched, you must be explicit about that. You want your update to apply to both `smith` and `jones`, so the multi-update is necessary.

We'll cover updates in more detail later, but try these examples before moving on.

2.1.5 Deleting data

Now you know the basics of creating, reading, and updating data through the MongoDB shell. We've saved the simplest operation, removing data, for last.

If given no parameters, a remove operation will clear a collection of all its documents. To get rid of, say, a `foo` collection'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()
```

Creating, reading, updating, and deleting are the basic operations of any database; if you've followed along, you should be in a position to continue practicing basic CRUD operations in MongoDB. In the next section, you'll learn how to enhance your queries, updates, and deletes by taking a brief look at secondary indexes.

2.1.6 Other shell features

You may have noticed this already, but the shell does a lot of things to make working with MongoDB easier. You can revisit earlier commands by using the `up` and `down`

arrows, and use autocomplete for certain inputs, like collection names. The autocomplete feature uses the tab key to autocomplete or to list the completion possibilities.¹ You can also discover more information in the shell by typing this:

```
> help
```

A lot of functions print pretty help messages that explain them as well. Try it out:

```
> db.help()
DB methods:
  db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs
    command [ just calls db.runCommand(...) ]
  db.auth(username, password)
  db.cloneDatabase(fromhost)
  db.commandHelp(name) returns the help for the command
  db.copyDatabase(fromdb, todb, fromhost)
...
```

Help on queries is provided through a different function called `explain`, which we'll investigate in later sections. There are also a number of options you can use when starting the MongoDB shell. To display a list of these, add the help flag when you start the MongoDB shell:

```
$ mongo --help
```

You don't need to worry about all these features, and we're not done working with the shell yet, but it's worth knowing where you can find more information when you need it.

2.2 Creating and querying with indexes

It's common to create indexes to enhance query performance. Fortunately, MongoDB's indexes can be created easily from the shell. If you're new to database indexes, this section should make the need for them clear; if you already have indexing experience, you'll see how easy it is to create indexes and then profile queries against them using the `explain()` method.

2.2.1 Creating a large collection

An indexing example makes sense only if you have a collection with many documents. So you'll add 20,000 simple documents to a `numbers` collection. Because the MongoDB shell is also a JavaScript interpreter, the code to accomplish this is simple:

```
> for(i = 0; i < 20000; i++) {
    db.numbers.save({num: i});
  }
WriteResult({ "nInserted" : 1 })
```

¹ For the full list of keyboard shortcuts, please visit <http://docs.mongodb.org/v3.0/reference/program/mongo/#mongo-keyboard-shortcuts>.

That's a lot of documents, so don't be surprised if the insert takes a few seconds to complete. Once it returns, you can run a couple of queries to verify that all the documents are present:

```
> db.numbers.count()
20000
> db.numbers.find()
{ "_id": ObjectId("4bfbf132db1aa7c30ac830a"), "num": 0 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac830b"), "num": 1 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac830c"), "num": 2 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac830d"), "num": 3 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac830e"), "num": 4 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac830f"), "num": 5 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8310"), "num": 6 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8311"), "num": 7 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8312"), "num": 8 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8313"), "num": 9 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8314"), "num": 10 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8315"), "num": 11 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8316"), "num": 12 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8317"), "num": 13 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8318"), "num": 14 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8319"), "num": 15 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac831a"), "num": 16 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac831b"), "num": 17 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac831c"), "num": 18 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac831d"), "num": 19 }
Type "it" for more
```

The `count()` command shows that you've inserted 20,000 documents. The subsequent query displays the first 20 results (this number may be different in your shell). You can display additional results with the `it` command:

```
> it
{ "_id": ObjectId("4bfbf132db1aa7c30ac831e"), "num": 20 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac831f"), "num": 21 }
{ "_id": ObjectId("4bfbf132db1aa7c30ac8320"), "num": 22 }
...
```

The `it` command instructs the shell to return the next result set.²

With a sizable set of documents available, let's try a couple queries. Given what you know about MongoDB's query engine, a simple query matching a document on its `num` attribute makes sense:

```
> db.numbers.find({num: 500})
{ "_id" : ObjectId("4bfbf132db1aa7c30ac84fe"), "num" : 500 }
```

² You may be wondering what's happening behind the scenes here. All queries create a cursor, which allows for iteration over a result set. This is somewhat hidden when using the shell, so it isn't necessary to discuss in detail at the moment. If you can't wait to learn more about cursors and their idiosyncrasies, see chapters 3 and 4.

RANGE QUERIES

More interestingly, you can also issue range queries using the special `$gt` and `$lt` operators. They stand for greater than and less than, respectively. Here's how you query for all documents with a `num` value greater than 199,995:

```
> db.numbers.find( {num: {"$gt": 19995 }} )
{ "_id" : ObjectId("552e660b58cd52bcb2581142"), "num" : 19996 }
{ "_id" : ObjectId("552e660b58cd52bcb2581143"), "num" : 19997 }
{ "_id" : ObjectId("552e660b58cd52bcb2581144"), "num" : 19998 }
{ "_id" : ObjectId("552e660b58cd52bcb2581145"), "num" : 19999 }
```

You can also combine the two operators to specify upper and lower boundaries:

```
> db.numbers.find( {num: {"$gt": 20, "$lt": 25 }} )
{ "_id" : ObjectId("552e660558cd52bcb257c33b"), "num" : 21 }
{ "_id" : ObjectId("552e660558cd52bcb257c33c"), "num" : 22 }
{ "_id" : ObjectId("552e660558cd52bcb257c33d"), "num" : 23 }
{ "_id" : ObjectId("552e660558cd52bcb257c33e"), "num" : 24 }
```

You can see that by using a simple JSON document, you're able to specify a range query in much the same way you might in SQL. `$gt` and `$lt` are only two of a host of operators that comprise the MongoDB query language. Others include `$gte` for greater than or equal to, `$lte` for (you guessed it) less than or equal to, and `$ne` for not equal to. You'll see other operators and many more example queries in later chapters.

Of course, queries like this are of little value unless they're also efficient. In the next section, we'll start thinking about query efficiency by exploring MongoDB's indexing features.

2.2.2 Indexing and explain()

If you've spent time working with relational databases, you're probably familiar with SQL's `EXPLAIN`, an invaluable tool for debugging or optimizing a query. When any database receives a query, it must plan out how to execute it; this is called a query plan. `EXPLAIN` describes query paths and allows developers to diagnose slow operations by determining which indexes a query has used. Often a query can be executed in multiple ways, and sometimes this results in behavior you might not expect. `EXPLAIN` explains. MongoDB has its own version of `EXPLAIN` that provides the same service. To get an idea of how it works, let's apply it to one of the queries you just issued. Try running the following on your system:

```
> db.numbers.find({num: {"$gt": 19995}}).explain("executionStats")
```

The result should look something like what you see in the next listing. The `"executionStats"` keyword is new to MongoDB 3.0 and requests a different mode that gives more detailed output.

Listing 2.1 Typical `explain("executionStats")` output for an unindexed query

```

{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "tutorial.numbers",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "num" : {
        "$gt" : 19995
      }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "num" : {
          "$gt" : 19995
        }
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 4,
    "executionTimeMillis" : 8,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 20000,
    "executionStages" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "num" : {
          "$gt" : 19995
        }
      },
      "nReturned" : 4,
      "executionTimeMillisEstimate" : 0,
      "works" : 20002,
      "advanced" : 4,
      "needTime" : 19997,
      "needFetch" : 0,
      "saveState" : 156,
      "restoreState" : 156,
      "isEOF" : 1,
      "invalidates" : 0,
      "direction" : "forward",
      "docsExamined" : 20000
    }
  },
  "serverInfo" : {
    "host" : "rMacBook.local",
    "port" : 27017,
    "version" : "3.0.6",

```



```

    "gitVersion" : "nogitversion"
  },
  "ok" : 1
}

```

Upon examining the `explain()` output,³ you may be surprised to see that the query engine has to scan the entire collection, all 20,000 documents (`docsExamined`), to return only four results (`nReturned`). The value of the `totalKeysExamined` field shows the number of index entries scanned, which is zero. Such a large difference between the number of documents scanned and the number returned marks this as an inefficient query. In a real-world situation, where the collection and the documents themselves would likely be larger, the time needed to process the query would be substantially greater than the eight milliseconds (`millis`) noted here (this may be different on your machine).

What this collection needs is an index. You can create an index for the `num` key within the documents using the `createIndex()` method. Try entering the following index creation code:

```

> db.numbers.createIndex({num: 1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}

```

The `createIndex()` method replaces the `ensureIndex()` method in MongoDB 3. If you're using an older MongoDB version, you should use `ensureIndex()` instead of `createIndex()`. In MongoDB 3, `ensureIndex()` is still valid as it's an alias for `createIndex()`, but you should stop using it.

As with other MongoDB operations, such as queries and updates, you pass a document to the `createIndex()` method to define the index's keys. In this case, the `{num: 1}` document indicates that an ascending index should be built on the `num` key for all documents in the `numbers` collection.

You can verify that the index has been created by calling the `getIndexes()` method:

```

> db.numbers.getIndexes()
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },

```

³ In these examples we're inserting "hostname" as the machine's hostname. On your platform this may appear as `localhost`, your machine's name, or its name plus `.local`. Don't worry if your output looks a little different than ours'; it can vary based on your platform and your exact version of MongoDB.

```

    "name" : "_id_",
    "ns" : "tutorial.numbers"
  },
  {
    "v" : 1,
    "key" : {
      "num" : 1
    },
    "name" : "num_1",
    "ns" : "tutorial.numbers"
  }
]

```

The collection now has two indexes. The first is the standard `_id` index that's automatically built for every collection; the second is the index you created on `num`. The indexes for those fields are called `_id_` and `num_1`, respectively. If you don't provide a name, MongoDB sets hopefully meaningful names automatically.

If you run your query with the `explain()` method, you'll now see the dramatic difference in query response time, as shown in the following listing.

Listing 2.2 `explain()` output for an indexed query

```

> db.numbers.find({num: {"$gt": 19995 }}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "tutorial.numbers",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "num" : {
        "$gt" : 19995
      }
    },
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "num" : 1
        },
        "indexName" : "num_1",
        "isMultiKey" : false,
        "direction" : "forward",
        "indexBounds" : {
          "num" : [
            "(19995.0, inf.0)"
          ]
        }
      },
      "rejectedPlans" : [ ]
    }
  },
  "rejectedPlans" : [ ]
},

```

Using the
num_1 index

```

"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 4,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 4,
  "totalDocsExamined" : 4,
  "executionStages" : {
    "stage" : "FETCH",
    "nReturned" : 4,
    "executionTimeMillisEstimate" : 0,
    "works" : 5,
    "advanced" : 4,
    "needTime" : 0,
    "needFetch" : 0,
    "saveState" : 0,
    "restoreState" : 0,
    "isEOF" : 1,
    "invalidates" : 0,
    "docsExamined" : 4,
    "alreadyHasObj" : 0,
    "inputStage" : {
      "stage" : "IXSCAN",
      "nReturned" : 4,
      "executionTimeMillisEstimate" : 0,
      "works" : 4,
      "advanced" : 4,
      "needTime" : 0,
      "needFetch" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 1,
      "invalidates" : 0,
      "keyPattern" : {
        "num" : 1
      },
      "indexName" : "num_1",
      "isMultiKey" : false,
      "direction" : "forward",
      "indexBounds" : {
        "num" : [
          "(19995.0, inf.0]"
        ]
      }
    },
    "keysExamined" : 4,
    "dupsTested" : 0,
    "dupsDropped" : 0,
    "seenInvalidated" : 0,
    "matchTested" : 0
  }
},
"serverInfo" : {
  "host" : "rMacBook.local",
  "port" : 27017,
  "version" : "3.0.6",

```

Four documents returned

Only four documents scanned

Much faster!

Using the num_1 index

```

    "gitVersion" : "nogitversion"
  },
  "ok" : 1
}

```

Now that the query utilizes the index `num_1` on `num`, it scans only the four documents pertaining to the query. This reduces the total time to serve the query from 8 ms to 0 ms!

Indexes don't come free; they take up some space and can make your inserts slightly more expensive, but they're an essential tool for query optimization. If this example intrigues you, be sure to check out chapter 8, which is devoted to indexing and query optimization. Next you'll look at the basic administrative commands required to get information about your MongoDB instance. You'll also learn techniques for getting help from the shell, which will aid in mastering the various shell commands.

2.3 *Basic administration*

This chapter promised to be an introduction to MongoDB via the JavaScript shell. You've already learned the basics of data manipulation and indexing. Here, we'll present techniques for getting information about your `mongod` process. For instance, you'll probably want to know how much space your various collections are taking up, or how many indexes you've defined on a given collection. The commands detailed here can take you a long way in helping to diagnose performance issues and keep tabs on your data.

We'll also look at MongoDB's command interface. Most of the special, non-CRUD operations that can be performed on a MongoDB instance, from server status checks to data file integrity verification, are implemented using database commands. We'll explain what commands are in the MongoDB context and show how easy they are to use. Finally, it's always good to know where to look for help. To that end, we'll point out places in the shell where you can turn for help to further your exploration of MongoDB.

2.3.1 *Getting database information*

You'll often want to know which collections and databases exist on a given installation. Fortunately, the MongoDB shell provides a number of commands, along with some syntactic sugar, for getting information about the system.

`show dbs` prints a list of all the databases on the system:

```

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

```

`show collections` displays a list of all the collections defined on the current database.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()` JavaScript 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
}
```

You can also run the `stats()` command on an individual collection:

```
> db.numbers.stats()
{
  "ns" : "tutorial.numbers",
  "count" : 20000,
```

⁴ You can also enter the more succinct `show tables`.

```

    "size" : 960064,
    "avgObjSize" : 48,
    "storageSize" : 2793472,
    "numExtents" : 5,
    "nindexes" : 2,
    "lastExtentSize" : 2097152,
    "paddingFactor" : 1,
    "paddingFactorNote" : "paddingFactor is unused and unmaintained in 3.0.
    It remains hard coded to 1.0 for compatibility only.",
    "systemFlags" : 1,
    "userFlags" : 1,
    "totalIndexSize" : 1169168,
    "indexSizes" : {
      "_id_" : 654080,
      "num_1" : 515088
    },
    "ok" : 1
  }
}

```

Some of the values provided in these result documents are useful only in complicated debugging or tuning situations. But at the very least, you'll be able to find out how much space a given collection and its indexes are occupying.

2.3.2 *How commands work*

A certain set of MongoDB operations—distinct from the insert, update, remove, and query operations described so far in this chapter—are known as database commands. Database commands are generally administrative, as with the `stats()` methods just presented, but they may also control core MongoDB features, such as updating data.

Regardless of the functionality they provide, what all database commands have in common is their implementation as queries on a special virtual collection called `$cmd`. To show what this means, let's take a quick example. Recall how you invoked the `stats()` database command:

```
> db.stats()
```

The `stats()` method is a helper that wraps the shell's command invocation method. Try entering the following equivalent operation:

```
> db.runCommand( {dbstats: 1} )
```

The results are identical to what's provided by the `stats()` method. Note that the command is defined by the document `{dbstats: 1}`. In general, you can run any available command by passing its document definition to the `runCommand()` method. Here's how you'd run the collection stats command:

```
> db.runCommand( {collstats: "numbers"} )
```

The output should look familiar.

But to get to the heart of database commands, you need to see how the `runCommand()` method works. That's not hard to find out because the MongoDB shell will print the implementation of any method whose executing parentheses are omitted. Instead of running the command like this

```
> db.runCommand()
```

you can execute the parentheses-less version and see the internals:

```
> db.runCommand
function ( obj, extra ){
  if ( typeof( obj ) == "string" ){
    var n = {};
    n[obj] = 1;
    obj = n;
    if ( extra && typeof( extra ) == "object" ) {
      for ( var x in extra ) {
        n[x] = extra[x];
      }
    }
  }
  return this.getCollection( "$cmd" ).findOne( obj );
}
```

The last line in the function is nothing more than a query on the `$cmd` collection. To define it properly, then, a database command is a query on a special collection, `$cmd`, where the query selector defines the command itself. That's all there is to it. Can you think of a way to run the collection stats command manually? It's this simple:

```
> db.$cmd.findOne( {collstats: "numbers"} );
```

Using the `runCommand` helper is easier but it's always good to know what's going on beneath the surface.

2.4 Getting help

By now, the value of the MongoDB shell as a testing ground for experimenting with data and administering the database should be evident. But because you'll likely spend a lot of time in the shell, it's worth knowing how to get help.

The built-in help commands are the first place to look. `db.help()` prints a list of commonly used methods for operating on databases. You'll find a similar list of methods for operating on collections by running `db.numbers.help()`.

There's also built-in tab completion. Start typing the first characters of any method and then press the Tab key twice. You'll see a list of all matching methods. Here's the tab completion for collection methods beginning with `get`:

```
> db.numbers.get
db.numbers.getCollection(          db.numbers.getIndexes(
db.numbers.getShardDistribution(
```

<code>db.numbers.getDB(</code>	<code>db.numbers.getIndices(</code>
<code>db.numbers.getShardVersion(</code>	
<code>db.numbers.getDiskStorageStats(</code>	<code>db.numbers.getMongo(</code>
<code>db.numbers.getSlaveOk(</code>	
<code>db.numbers.getFullName(</code>	<code>db.numbers.getName(</code>
<code>db.numbers.getSplitKeysForChunks(</code>	
<code>db.numbers.getIndexKeys(</code>	<code>db.numbers.getPagesInRAM(</code>
<code>db.numbers.getWriteConcern(</code>	
<code>db.numbers.getIndexSpecs(</code>	<code>db.numbers.getPlanCache(</code>
<code>db.numbers.getIndexStats(</code>	<code>db.numbers.getQueryOptions(</code>

The official MongoDB manual is an invaluable resource and can be found at <http://docs.mongodb.org>. It has both tutorials and reference material, and it's kept up-to-date with new releases of MongoDB. The manual also includes documentation for each language-specific MongoDB driver implementation, such as the Ruby driver, which is necessary when accessing MongoDB from an application.

If you're more ambitious, and are comfortable with JavaScript, the shell makes it easy to examine the implementation of any given method. For instance, suppose you'd like to know exactly how the `save()` method works. Sure, you could go trolling through the MongoDB source code, but there's an easier way: enter the method name without the executing parentheses. Here's how you'd normally execute `save()`:

```
> db.numbers.save({num: 123123123});
```

And this is how you can check the implementation:

```
> db.numbers.save
function ( obj , opts ){
  if ( obj == null )
    throw "can't save a null";

  if ( typeof( obj ) == "number" || typeof( obj ) == "string" )
    throw "can't save a number or string"

  if ( typeof( obj._id ) == "undefined" ){
    obj._id = new ObjectId();
    return this.insert( obj , opts );
  }
  else {
    return this.update( { _id : obj._id } , obj , Object.merge({
      upsert:true }, opts));
  }
}
```

Read the function definition closely, and you'll see that `save()` is merely a wrapper for `insert()` and `update()`. After checking the type of the `obj` argument, if the object you're trying to save doesn't have an `_id` field, then the field is added, and `insert()` is invoked. Otherwise an update is performed.

This trick for examining the shell's methods comes in handy. Keep this technique in mind as you continue exploring the MongoDB shell.

2.5 Summary

You've now seen the document data model in practice, and we've demonstrated a variety of common MongoDB operations on that data model. You've learned how to create indexes and have seen an example of index-based performance improvements through the use of `explain()`. In addition, you should be able to extract information about the collections and databases on your system, you now know all about the clever `$cmd` collection, and if you ever need help, you've picked up a few tricks for finding your way around.

You can learn a lot by working in the MongoDB shell, but there's no substitute for the experience of building a real application. That's why we're going from a carefree data playground to a real-world data workshop in the next chapter. You'll see how the drivers work, and then, using the Ruby driver, you'll build a simple application, hitting MongoDB with some real, live data.



Writing programs using MongoDB

This chapter 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

It's time to get practical. Though there's much to learn from experimenting with the MongoDB shell, you can see the real value of this database only after you've built something with it. That means jumping into programming and taking a first look at the MongoDB drivers. As mentioned before, MongoDB, Inc. provides officially supported, Apache-licensed MongoDB drivers for all of the most popular programming languages. The driver examples in the book use Ruby, but the principles we'll illustrate are universal and easily transferable to other drivers. Throughout the book we'll illustrate most commands with the JavaScript shell, but examples of using MongoDB from within an application will be in Ruby.

We're going to explore programming in MongoDB in three stages. First, you'll install the MongoDB Ruby driver and we'll introduce the basic CRUD (create, read, update, delete) operations. This process should go quickly and feel familiar because

the driver API is similar to that of the shell. Next, we're going to delve deeper into the driver, explaining how it interfaces with MongoDB. Without getting too low-level, this section will show you what's going on behind the scenes with the drivers in general. Finally, you'll develop a simple Ruby application for monitoring Twitter. Working with a real-world data set, you'll begin to see how MongoDB works in the wild. This final section will also lay the groundwork for the more in-depth examples presented in part 2 of the book.

New to Ruby?

Ruby is a popular and readable scripting language. The code examples have been designed to be as explicit as possible so that even programmers unfamiliar with Ruby can benefit. Any Ruby idioms that may be hard to understand will be explained in the book. If you'd like to spend a few minutes getting up to speed with Ruby, start with the official 20-minute tutorial at <http://mng.bz/THR3>.

3.1 MongoDB through the Ruby lens

Normally when you think of drivers, what comes to mind are low-level bit manipulations and obtuse interfaces. Thankfully, the MongoDB language drivers are nothing like that; instead, they've been designed with intuitive, language-sensitive APIs so that many applications can sanely use a MongoDB driver as the sole interface to the database. The driver APIs are also fairly consistent across languages, which means that developers can easily move between languages as needed; anything you can do in the JavaScript API, you can do in the Ruby API. If you're an application developer, you can expect to find yourself comfortable and productive with any of the MongoDB drivers without having to concern yourself with low-level implementation details.

In this first section, you'll install the MongoDB Ruby driver, connect to the database, and learn how to perform basic CRUD operations. This will lay the groundwork for the application you'll build at the end of the chapter.

3.1.1 Installing and connecting

You can install the MongoDB Ruby driver using RubyGems, Ruby's package management system.

Many newer operating systems come with Ruby already installed. You can check if you already have Ruby installed by running `ruby --version` from your shell. If you don't have Ruby installed on your system, you can find detailed installation instructions at www.ruby-lang.org/en/downloads.

You'll also need Ruby's package manager, RubyGems. You may already have this as well; check by running `gem --version`. Instructions for installing RubyGems can be found at <http://docs.rubygems.org/read/chapter/3>. Once you have RubyGems installed, run:

```
gem install mongo
```

This should install both the `mongo` and `bson`¹ gems. You should see output like the following (the version numbers will likely be newer than what's shown here):

```
Fetching: bson-3.2.1.gem (100%)
Building native extensions. This could take a while...
Successfully installed bson-3.2.1
Fetching: mongo-2.0.6.gem (100%)
Successfully installed mongo-2.0.6
2 gems installed
```

We also recommend you install the `bson_ext` gem, though this is optional. `bson_ext` is an official gem that contains a C implementation of BSON, enabling more efficient handling of BSON in the MongoDB driver. This gem isn't installed by default because installation requires a compiler. Rest assured, if you're unable to install `bson_ext`, your programs will still work as intended.

You'll start by connecting to MongoDB. First, make sure that `mongod` is running by running the `mongo` shell 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 reference to the `users` collection in the `$users` variable, and print the string `connected!`. We place a `$` in front of each variable to make it global so that it'll be accessible outside 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!
```

If no exceptions are raised, you've successfully connected to MongoDB from Ruby and you should see `connected!` printed to your shell. That may not seem glamorous, but connecting is the first step in using MongoDB from any language. Next, you'll use that connection to insert some documents.

¹ BSON, explained in the next section, is the JSON-inspired binary format that MongoDB uses to represent documents. The `bson` Ruby gem serializes Ruby objects to and from BSON.

3.1.2 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 natural 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

If you're following along, you can continue adding code to the `connect.rb` file. Alternatively, a nice approach is to use Ruby's interactive shell, `irb`. `irb` is a REPL (Read, Evaluate, Print Loop) console, in which you can type Ruby code to have it dynamically executed, making it ideal for experimentation. Anything you write in `irb` can be put in a script, so we recommend using it to learn new things, then copying your commands when you'd like them executed in a program. You can launch `irb` and require `connect.rb` so that you'll immediately have access to the connection, database, and collection objects initialized therein. You can then run Ruby code and receive immediate feedback. 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>
```

`irb` gives you a command line shell with a prompt followed by `>` (this may look a little different on your machine). The prompt allows you to type in commands, and in the previous code we've highlighted the user input in bold. When you run a command in `irb` it will print out the value returned by the command, if there is one; that's what is shown after `=>` above.

Let's build some documents for your users' collection. You'll create two documents representing two users, Smith and Jones. Each document, expressed as a Ruby hash, is assigned to a variable:

```
smith = {"last_name" => "smith", "age" => 30}
jones = {"last_name" => "jones", "age" => 40}
```

² In Ruby 1.9, you may optionally use a colon as the key-value separator, like `hash = {foo: 'bar'}`, but we'll stick with the hash rocket in the interest of backward compatibility.

To save the documents, you'll pass them to the collection's `insert` method. Each call to `insert` returns a unique ID, which you'll store in a variable to simplify later retrieval:

```
smith_id = $users.insert_one(smith)
jones_id = $users.insert_one(jones)
```

You can verify that the documents have been saved with some simple queries, so you can query with the user collection's `find()` method like this:

```
irb(main):013:0> $users.find("age" => {"$gt" => 20}).each.to_a do |row|
irb(main):014:1* puts row
irb(main):015:1> end
=> [{"_id"=>BSON::ObjectId('55e3f7dd5ae119516a000002'), "last_name"=>"smith",
    "age"=>30}, {"_id"=>BSON::ObjectId('55e3f7e25ae119516a000003'),
    "last_name"=>"jones", "age"=>40}]
```

The return values for these queries will appear at the prompt if run in `irb`. If the code is being run from a Ruby file, prepend Ruby's `p` method to print the output to the screen:

```
p $users.find(:age => {"$gt" => 20}).to_a
```

You've successfully inserted two documents from Ruby. Let's now take a closer look at queries.

3.1.3 *Queries and cursors*

Now that you've created documents, it's on to the read operations (the R in CRUD) provided by MongoDB. The Ruby driver defines a rich interface for accessing data and handles most of the details for you. The queries we show in this section are fairly simple selections, but keep in mind that MongoDB allows more complex queries, such as text searches and aggregations, which are described in later chapters.

You'll see how this is so by looking at the standard `find` method. Here are two possible `find` operations on your data set:

```
$users.find({"last_name" => "smith"}).to_a
$users.find({"age" => {"$gt" => 30}}).to_a
```

The first query searches for all user documents where the `last_name` is `smith` and that the second query matches all documents where `age` is greater than 30. Try entering the second query in `irb`:

```
2.1.4 :020 > $users.find({"age" => {"$gt" => 30}})
=> #<Mongo::Collection::View:0x70210212601420 namespace='tutorial.users'
@selector={"age"=>{"$gt"=>30}} @options={}>
```

The results are returned in a `Mongo::Collection::View` object, which extends `Iterable` and makes it easy to iterate through the results. We'll discuss cursors in

more detail in Section 3.2.3. In the meantime, you can fetch the results of the `$gt` query:

```
cursor = $users.find({"age" => {"$gt" => 30}})
cursor.each do |doc|
  puts doc["last_name"]
end
```

Here you use Ruby's `each` iterator, which passes each result to a code block. The `last_name` attribute is then printed to the console. The `$gt` used in the query is a MongoDB operator; the `$` character has no relation to the `$` placed before global Ruby variables like `$users`. Also, if there are any documents in the collection without `last_name`, you might notice that `nil` (Ruby's null value) is printed out; this indicates the lack of a value and it's normal to see this.

The fact that you even have to think about cursors here may come as a surprise given the shell examples from the previous chapter. But the shell uses cursors the same way every driver does; the difference is that the shell automatically iterates over the first 20 cursor results when you call `find()`. To get the remaining results, you can continue iterating manually by entering the `it` command.

3.1.4 Updates and deletes

Recall from chapter 2 that *updates* require at least two arguments: a query selector and an update document. Here's a simple example using the Ruby driver:

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

The view allows you to decide whether you only want to update one document or all documents matching the query. In the preceding example, even if you had several users with the last name of `smith`, only one document would be updated. To apply the update to a particular `smith`, you'd need to add more conditions to your query selector. But if you actually want to apply the update to all `smith` documents, you must replace the `update_one` with the `update_many` method:

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

Deleting data is much simpler. We've discussed how it works in the MongoDB shell and the Ruby driver is no different. To review: you simply use the `remove` method. This method takes an optional query selector that will remove only those documents matching the selector. If no selector is provided, all documents in the collection will

be removed. Here, you're removing all user documents where the age attribute is greater than or equal to 40:

```
$users.find({"age" => {"$gte" => 40}}).delete_one
```

This will only delete the first one matching the matching criteria. If you want to delete all documents matching the criteria, you'd have to run this:

```
$users.find({"age" => {"$gte" => 40}}).delete_many
```

With no arguments, the drop method deletes all remaining documents:

```
$users.drop
```

3.1.5 Database commands

In the previous chapter you saw the centrality of database commands. There, we looked at the two stats commands. Here, we'll look at how you can run commands from the driver using the `listDatabases` command as an example. This is one of a number of commands that must be run on the admin database, which is treated specially when authentication is enabled. For details on the authentication and the admin database, see chapter 10.

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"=>83886080.0,
  "empty"=>false
},
{
  "name"=>"tutorial",
  "sizeOnDisk"=>83886080.0,
  "empty"=>false
},
{
  "name"=>"admin",
  "sizeOnDisk"=>1.0, "empty"=>true
}], "totalSize"=>167772160.0, "ok"=>1.0}]>
=> nil
```


This may look a little different with your version of `irb` and the MongoDB driver, but it should still be easy to access. Once you get used to representing documents as Ruby hashes, the transition from the shell API is almost seamless.

Most drivers provide you convenient functionality that wraps database commands. You may recall from the previous chapter that `remove` doesn't actually drop the collection. To drop a collection and all its indexes, use the `drop_collection` method:

```
db = $client.use('tutorial')
db['users'].drop
```

It's okay if you're still feeling shaky about using MongoDB with Ruby; you'll get more practice in section 3.3. But for now, we're going to take a brief intermission to see how the MongoDB drivers work. This will shed more light on some of MongoDB's design and prepare you to use the drivers effectively.

3.2 How the drivers work

At this point it's natural to wonder what's going on behind the scenes when you issue commands through a driver or via the MongoDB shell. In this section, you'll see how the drivers serialize data and communicate it to the database.

All MongoDB drivers perform three major functions. First, they generate MongoDB 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.

3.2.1 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 `_id` field 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 guaranteed to be unique within a certain context. How can this be guaranteed? Let's examine 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`. You may not have realized that this text is the hex

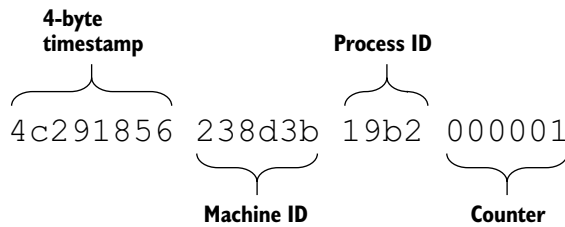


Figure 3.1 MongoDB object ID format

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) timestamp³. 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 documents, 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 if you 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

³ Many Unix machines (we’re including Linux when we say Unix machine) store time values in a format called Unix Time or POSIX time; they just count up the number of seconds since 00:00 on January 1st, 1970, called the epoch. This means that a timestamp can be stored as an integer. For example, 2010-06-28 21:47:02 is represented as 1277761622 (or 0x4c291856 in hexadecimal), the number of seconds since the epoch.

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.

3.3 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 documents. 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.

3.3.1 Setting up

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

⁴ This example will actually not work; it's meant as a thoughtful exercise. By now you should have enough knowledge to create meaningful data for the query to return something. Why not take the time and try it out?

to chapter3 and make sure an `ls` command shows the `Gemfile`. You can then install them from your system command line like this:

```
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. The `sinatra` gem is a framework for running a simple web server in Ruby, and we discuss it in more detail in section 3.3.3.

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 application. 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 registered an application, you should see a page with its authentication information, perhaps 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.

3.3.2 *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
  end
end
```

The `initialize` method instantiates a connection, a database object, and the collection 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 configuration; the `config` variable is passed to a Ruby block, in which you set its values.

MongoDB allows you to insert data regardless of its structure. With a relational database, each table needs a well-defined schema, which requires planning out which values you would like to store. 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_doc)
  end
end
```

Before saving each Tweet document, make two small modifications. To simplify later queries, add the search term to a `tags` attribute. 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 `save` method.

To use this code in a class, you need some additional code. First, you must configure the MongoDB driver so that it connects to the correct `mongod` and 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

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

  private
```

**Create
a new
instance
of Tweet-
Archive.**

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

**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
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) containing 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 collection directly:

```

> use twitter-archive
switched to db twitter-archive
> db.tweets.count()
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.

3.3.3 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
  "response"
end

```

⁵ It's possible to accomplish this in far fewer lines of code. Doing so is left as an exercise to the reader.

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'
```

1 Required libraries

```

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

2 Instantiate collection for tweets

```

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

3 Dynamically build query selector...

```

  else
    selector = {}
  end
```

4 ...or use blank selector

```

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

5 Issue query

6 Render view

```
end
```

The first lines require the necessary libraries along with your config file **1**. Next there's a configuration block that creates a connection to MongoDB and stores a reference to your tweets collection in the constant `TWEETS` **2**.

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 tags URL parameter has been provided, you create a query selector that restricts the result set to the given tags ❸. Otherwise, you create a blank selector, which returns all documents in the collection ❹. You then issue the query ❺. By now, you should know that what gets assigned to the @tweets variable isn't a result set but a cursor. You'll iterate over that cursor in your view.

The last line ❻ 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>
  
<% 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.each` code, 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.

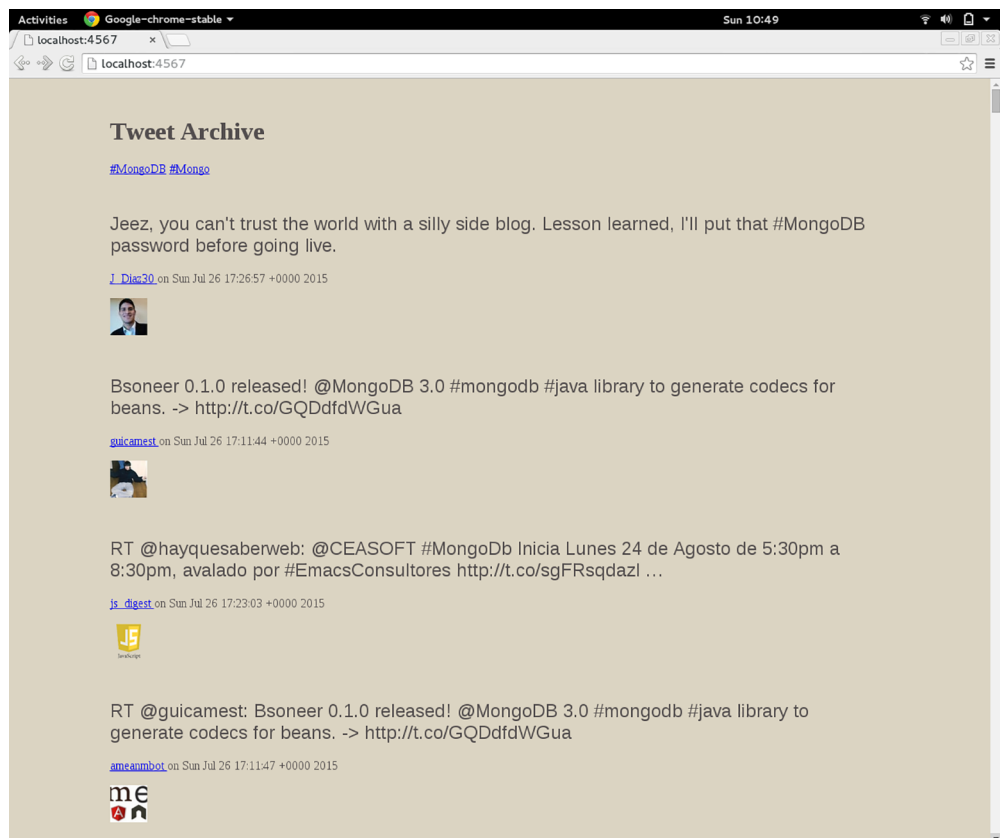


Figure 3.2 Tweet Archiver output rendered in a web browser

That's the extent of the application. It's admittedly simple, but it demonstrates some of the ease of using MongoDB. You didn't have to define your schema in advance, you took advantage of secondary indexes to make your queries fast and prevent duplicate inserts, and you had a relatively simple integration with your programming language.

3.4 Summary

You've just learned the basics of talking to MongoDB through the Ruby programming language. You saw how easy it is to represent documents in Ruby, and how similar Ruby's CRUD API is to that of the MongoDB shell. We dove into some internals, exploring how the drivers in general are built and looking in detail at object IDs, BSON, and the MongoDB network protocol. Finally, you built a simple application to show the use of MongoDB with real data. Though using MongoDB in the real world often requires more complexity, the prospect of writing applications with the database should be in reach.

Beginning with chapter 4, we're going to take everything you've learned so far and drill down. Specifically, you'll investigate how you might build an e-commerce application in MongoDB. That would be an enormous project, so we'll focus solely on a few sections on the back end. We'll present some data models for that domain, and you'll see how to insert and query that kind of data.

