

A database for the modern web

In this chapter

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

If you've built web applications in recent years, you've probably used a relational database as the primary data store, and it probably performed acceptably. Most developers are familiar with SQL, and most of us can appreciate the beauty of a well-normalized data model, the necessity of transactions, and the assurances provided by a durable storage engine. And even if we don't like working with relational databases directly, a host of tools, from administrative consoles to object-relational mappers, helps alleviate any unwieldy complexity. Simply put, the relational database is mature and well known. So when a small but vocal cadre of developers starts advocating alternative data stores, questions about the viability and utility of these new technologies arise. Are these new data stores replacements for relational database systems? Who's using them in production, and why? What are the trade-offs involved in moving 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 for 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 then would you use MongoDB?

It turns out that MongoDB is immediately attractive, not because of its scaling strategy, but rather because of its intuitive data model. Given that a document-based data model can represent rich, hierarchical data structures, it's often possible to do without the complicated multi-table joins imposed by 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, we'll need to write a complicated SQL query full of joins. As a consequence, most developers will need to rely on a secondary piece of software to assemble the data into something meaningful.

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.¹ 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, the object defined in the programming language can be persisted "as is," removing some of the complexity of object mappers.

If the distinction between a tabular and object representation of data is new to you, then you probably have a lot of questions. Rest assured that by the end of this chapter I'll have provided a thorough overview of MongoDB's features and design goals, making it increasingly clear why developers from companies like Geek.net (SourceForge.net) and The New York Times have adopted MongoDB for their projects. We'll see the history of MongoDB and lead into a tour of the database's main features. Next, we'll explore some alternative database solutions and the so-called NoSQL movement,² explaining how MongoDB fits in. Finally, I'll describe in general where MongoDB works best and where an alternative data store might be preferable.

¹ JSON is an acronym for *JavaScript Object Notation*. As we'll see shortly, JSON structures are comprised of keys and values, and they can nest arbitrarily deep. They're analogous to the dictionaries and hash maps of other programming languages.

² The umbrella term *NoSQL* was coined in 2009 to lump together the many nonrelational databases gaining in popularity at the time.

1.1 Born in the cloud

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 called 10gen began work on a software platform-as-a-service, composed of an application server and a database, that would host web applications and scale them as needed. Like Google's AppEngine, 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* want 10gen's new database technology. This led 10gen to concentrate its efforts solely on the database that became MongoDB.

With MongoDB's increasing adoption and production deployments large and small, 10gen 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, all of MongoDB's core developers are either founders or employees of 10gen, 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, 10gen's business model is not unlike that of other well-known open source companies: support the development of an open source product and provide subscription services to end users.

This history contains a couple of important ideas. First is that MongoDB was originally developed for a platform that, by definition, required its database to scale gracefully across multiple machines. The second is that MongoDB was designed as a data store for web applications. As we'll see, MongoDB's design as a horizontally scalable primary data store sets it apart from other modern database systems.

1.2 MongoDB's key features

A database is defined in large part by its data model. In this section, we'll look at the document data model, and then we'll see the features of MongoDB that allow us to operate effectively on that model. We'll also look at operations, focusing on MongoDB's flavor of replication and on its strategy for scaling horizontally.

1.2.1 The 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 example.

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',
```

← **id field
is primary key**

```
author: 'msmith',
vote_count: 20,

tags: ['databases', 'mongodb', 'indexing'],

image: {
  url: 'http://example.com/db.jpg',
  caption: '',
  type: 'jpg',
  size: 75381,
  data: "Binary"
},

comments: [
  { user: 'bjones',
```

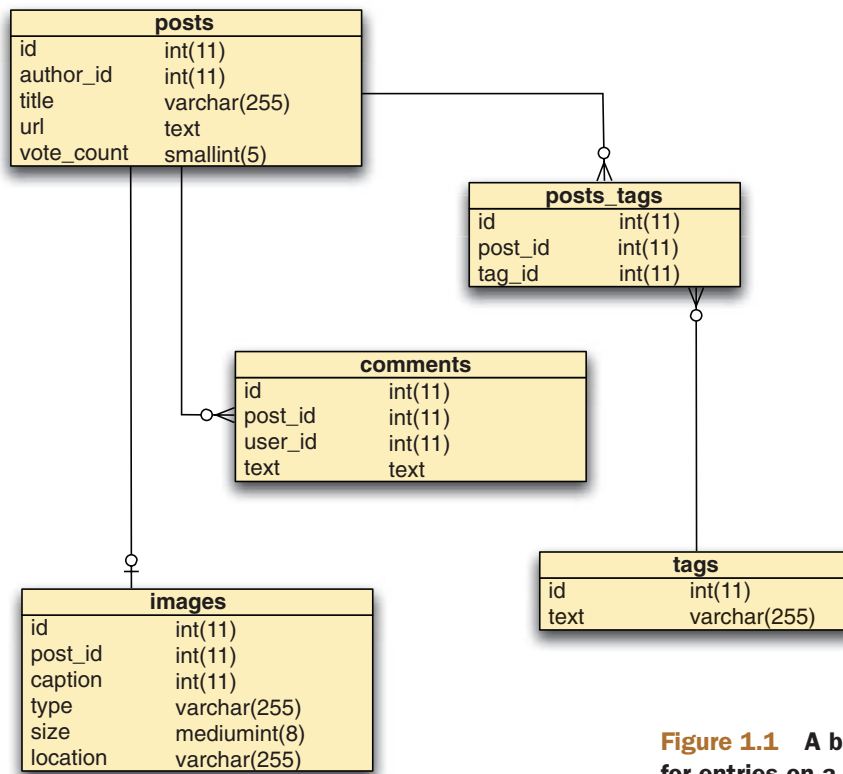


Figure 1.1 A basic relational data model for entries on a social news site

You've probably noticed that in addition to providing a richness of structure, documents need not 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 documents will be relatively uniform. For instance, every document in the `posts` collection will have fields for the title, tags, comments, and so forth.

But 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 schemaless model allows you to represent data with truly variable properties. For example, imagine you're building an e-commerce 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. What you're seeing is one section of the data model for Magento, an open source 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

³ http://en.wikipedia.org/wiki/Entity-attribute-value_model

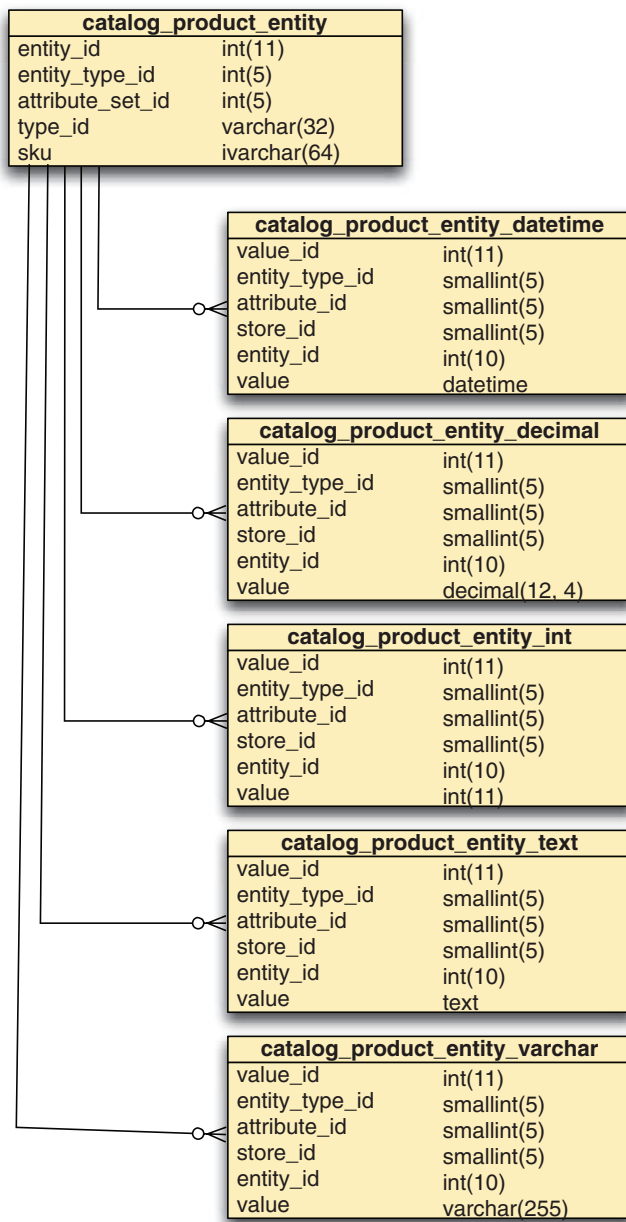


Figure 1.2 A portion of the schema for the PHP e-commerce project Magento. These tables facilitate dynamic attribute creation for products.

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.

1.2.2 Ad hoc queries

To say that a system supports ad hoc queries is to say that it's not necessary to define in advance what sorts of queries the system will accept. Relational databases have this property; they will 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 greater 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 queryability.

As mentioned earlier, some databases don't support ad hoc queries because the data model is too simple. For example, you can query a key-value store by primary key only. The values pointed to by those keys are opaque as far as the queries are concerned. The only way to query by a secondary attribute, such as this example's vote count, is to write custom code to manually build entries where the primary key indicates a given vote count and the value stores a list of the primary keys of the documents containing said vote count. If you took this approach with a key-value store, you'd be guilty of implementing a hack, and although it might work for smaller data sets, stuffing multiple indexes into what's physically a single index isn't a good idea. What's more, the hash-based index in a key-value store won't support range queries, which would probably be necessary for querying on an item like a vote count.

If you're coming from a relational database system where ad hoc queries are the norm, then it's sufficient to note that MongoDB features a similar level of queryability. If you've been evaluating a variety of database technologies, you'll want to keep in mind that not all of these technologies support ad hoc queries and that if you do need them, MongoDB could be a good choice. But ad hoc queries alone aren't enough. Once your data set grows to a certain size, indexes become necessary for query efficiency. Proper indexes will increase query and sort speeds by orders of magnitude; consequently, any system that supports ad hoc queries should also support secondary indexes.

1.2.3 Secondary indexes

The best way to understand database indexes is by analogy: many books have indexes mapping 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.

Secondary indexes in MongoDB are implemented as *B-trees*. B-tree indexes, also the default for most relational databases, are optimized for a variety of queries, including range scans and queries with sort clauses. 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, and even geospatial indexes 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 compatible. We'll begin looking at indexes in the next chapter, and because an understanding of indexing is so crucial to efficiently operating a database, I devote all of chapter 7 to the topic.

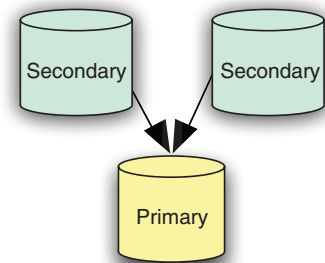
1.2.4 Replication

MongoDB provides database replication via a topology known as a *replica set*. Replica sets distribute data across 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.

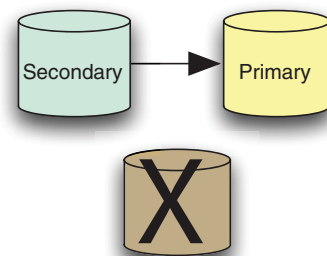
Replica sets consist of exactly one primary node and one or more secondary nodes. Like the master-slave replication that you may be familiar with from other databases, a replica set's primary node can accept both reads and writes, but the secondary nodes are read-only. What makes replica sets unique 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.

I discuss replication in chapter 8.

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

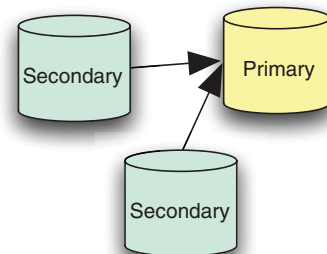


Figure 1.3 Automated failover with a replica set

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 is, maybe, and it depends on both your database system and the hardware hosting it. 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.

In MongoDB's case, users control the speed and durability trade-off by choosing write semantics and deciding whether to enable journaling. All writes, by default, are *fire-and-forget*, which means that these writes are sent across a TCP socket without requiring a database response. If users want a response, they can issue a write using a special *safe mode* provided by all drivers. This forces a response, ensuring that the write has been received by the server with no errors. Safe mode is configurable; it can also be used to block until a write has been replicated to some number of servers. 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 preferable.

In MongoDB v2.0, journaling is enabled by default. With journaling, every write is committed to an append-only log. 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.

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. Since 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 since some sort of write to disk is necessary to guarantee durability, it's important that the write be sequential; 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.

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 must run with replication, preferably to a second data center, to increase the likelihood that a pristine copy of the data will still exist even if there's a failure.

The topics of replication and durability are vast; you'll see a detailed exploration of them in chapter 8.

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 disk IOPS, memory, and 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. If you're running on virtualized hardware (such as Amazon's EC2), then you may find that a sufficiently large instance isn't available. If you're running on physical hardware, there may come a point where the cost of a more powerful server becomes prohibitive.

It then makes sense to consider scaling *horizontally*, or *scaling out*. Instead of beefing up a single node, scaling horizontally means distributing the database across multiple machines. Because a horizontally scaled architecture can use commodity hardware, the costs for hosting the total data set can be significantly reduced. What's more, the 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 upon 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 since a single machine represents a much smaller percentage of the system as a whole.

MongoDB has been designed to make horizontal scaling manageable. It does so via a range-based partitioning mechanism, known as *auto-sharding*, which automatically manages the distribution of data across nodes. 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,⁴ 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.

We've covered a lot of MongoDB's most compelling features; in chapter 2, we'll begin to see how some of these work in practice. But at this point, we're going to take

⁴ Technically, each replica set will have at least three nodes, but only two of these need carry a copy of the data.

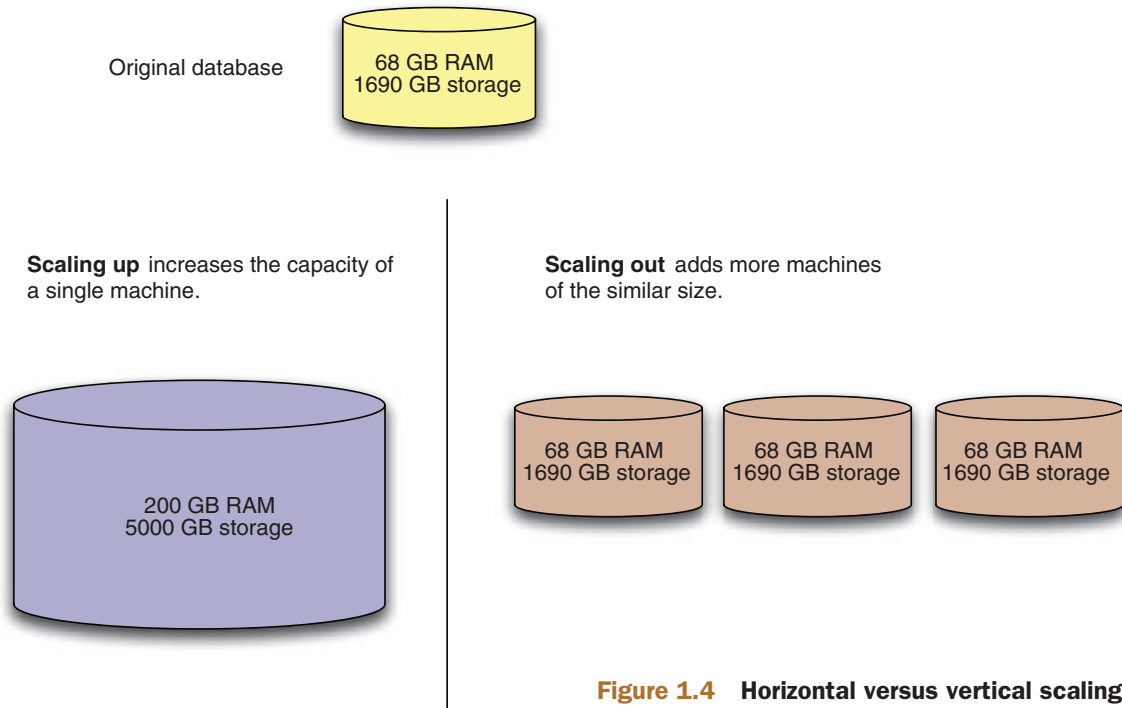


Figure 1.4 Horizontal versus vertical scaling

a more pragmatic look at the database. In the next section, we'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 10gen. The project compiles on all major operating systems, including Mac OS X, Windows, and most flavors of Linux. Precompiled binaries are available for each of these platforms at mongodb.org. MongoDB is open source and licensed under the GNU-AGPL. The source code is freely available on GitHub, and contributions from the community are frequently accepted. But the project is guided by the 10gen core server team, and the overwhelming majority of commits come from this group.

ON THE GNU-AGPL The GNU-AGPL is subject to some controversy. What this licensing means in practice is that the source code is freely available and that contributions from the community are encouraged. The primary limitation of the GNU-AGPL is that any modifications made to the source code must be published publicly for the benefit of the community. For companies wanting to safeguard their core server enhancements, 10gen 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 v2.0.⁵

⁵ You should always use the latest stable point release; for example, v2.0.1.

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.

1.3.1 The core server

The core database server runs via an executable called `mongod` (`mongodb.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`.⁶

`mongod` can be run in several modes, the most common of which is as a member of a replica set. Since replication is recommended, you generally see replica set configurations consisting of two replicas, plus an arbiter process residing on a third server.⁷ For MongoDB's auto-sharding architecture, the components consist of `mongod` processes configured as per-shard replica sets, with special metadata servers, known as *config servers*, on the side. A separate routing server called `mongos` is also used to send requests to the appropriate shard.

Configuring a `mongod` process is relatively simple compared with other database systems such as MySQL. Though it's possible to specify standard ports and data directories, there are few options for tuning the database. Database tuning, which in most RDBMSs means tinkering with a wide array of parameters controlling memory allocation and the like, has become something of a black art. MongoDB's design philosophy dictates that memory management is better handled by the operating system than by a DBA or application developer. Thus, data files are mapped to a system's virtual memory using the `mmap()` system call. This effectively offloads memory management responsibilities to the OS kernel. I'll have more to say about `mmap()` later in the book; for now it suffices to note that the lack of configuration parameters is a design feature, not a bug.

1.3.2 The JavaScript shell

The MongoDB command shell is a JavaScript-based tool for administering the database and manipulating data. The `mongo` executable loads the shell and connects to a specified `mongod` process. The shell has many of the same powers as the MySQL shell, the primary difference being that SQL isn't used. Instead, most commands are issued using JavaScript expressions. For instance, you can pick your database and then insert a simple document into the `users` collection like so:

```
> use mongodb-in-action
> 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:

⁶ `c:\data\db` on Windows.

⁷ These arbiter processes are lightweight and can easily be run on an app server, for instance.

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

The `find` method returns the inserted document, with the 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, then 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. 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 represent documents using a special document builder class that implements `LinkedHashMap`.

Because the drivers provide a rich, language-centric interface to the database, little abstraction beyond the driver itself is required to build an application. This contrasts notably with the application design for an RDBMS, where a library is almost certainly necessary to mediate between the relational data model of the database and the object-oriented model of most modern programming languages. Still, even if the heft of an object-relational mapper isn't required, many developers like using a thin wrapper over the drivers to handle associations, validations, and type checking.⁸

At the time of this writing, 10gen officially supports drivers for C, C++, C#, Erlang, Haskell, Java, Perl, PHP, Python, Scala, and Ruby—and the list is always growing. If you need support for another language, there's probably a community-supported driver for it. If no community-supported driver exists for your language, specifications for building a new driver are documented at mongodb.org. Since 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, I describe how the drivers work and how to use them to write programs.

⁸ A few popular wrappers at the time of this writing include Morphia for Java, Doctrine for PHP, and MongoMapper for Ruby.

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`—These utilities export and import JSON, CSV, and TSV 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 you should note in passing that before importing, it’s often desirable to adjust the data model to take best advantage of MongoDB. In those 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. Essentially translates the BSON going over the wire to human-readable shell statements.
- `mongostat`—Similar to `iostat`; 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.

The remaining utilities, `bsondump` and `monfiles`, are discussed later in the book.

1.4 **Why MongoDB?**

I’ve already provided a few reasons why MongoDB might be a good choice for your projects. Here, I’ll make this more explicit, first by considering the overall design objectives of the MongoDB project. According to its creators, MongoDB has been designed to combine the best features of key-value stores and relational databases. Key-value stores, because of their simplicity, are extremely fast and relatively easy to scale. Relational databases are more difficult to scale, at least horizontally, but admit a rich data model and a powerful query language. If MongoDB represents a mean between these two designs, then the reality is a database that scales easily, stores rich data structures, and provides sophisticated query mechanisms.

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

The preceding claims are bold. In order 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, I’ll discuss some specific MongoDB use cases and provide examples of them in production. Finally, I’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 the sections that follow, I describe simple and sophisticated key-value stores, relational databases, and document databases, and show how these compare and contrast 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 data store.	Caching. Web ops.
Sophisticated key-value stores	Cassandra, Project Voldemort, Riak	Variable. Cassandra uses a key-value structure known as a <i>column</i> . Voldemort stores binary blobs.	Eventually consistent, multinode distribution for high availability and easy failover.	High throughput verticals (activity feeds, message queues). Caching. Web ops.
Relational databases	Oracle database, 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 put 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 (pronounced *mem-cash-dee*). Memcached stores its data in memory only, so it trades persistence for speed. It's also distributed; memcached nodes running across multiple servers can act as a single data store, 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 data stores. 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, we 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 entitled *Dynamo: Amazon's Highly Available Key-value Store*. The aim of Dynamo is to be a database robust enough to continue functioning in the face of network failures, data center outages, and other similar disruptions. This requires that the system can 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, Project Voldemort, and Riak.

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 data store built by Facebook for its inbox search feature. The system scaled horizontally to index more than 50 TB of inbox data, allowing for searches on inbox keywords and recipients. Data was indexed by user ID, where each record consisted of an array of search terms for keyword searches and an array of recipient IDs for recipient searches.⁹

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 single master (per shard), a richer data model, and secondary indexes. The last two of these attributes go hand in hand; if a system allows for modeling multiple domains, as, for example, would be required to build a complete web application, then it's going to be necessary to query across the entire data model, thus requiring secondary indexes.

⁹ <http://mng.bz/5321>

Because of the richer data model, MongoDB can be considered a more general solution to the problem of large, scalable web applications. MongoDB's scaling architecture is sometimes criticized because it's not inspired by Dynamo. But there are different scaling solutions for different domains. MongoDB's auto-sharding is inspired by Yahoo!'s PNUTS data store and Google's BigTable. Anyone who reads the white papers presenting these data stores will see that MongoDB's approach to scaling has already been implemented, and successfully so.

RELATIONAL DATABASES

Much has already been said of relational databases in this introduction, so in the interest of brevity, I need only discuss what RDBMSs have in common with MongoDB and where they diverge. MongoDB and MySQL¹⁰ are both capable of representing a rich data model, although where MySQL uses fixed-schema tables, MongoDB has schema-free documents. MySQL and MongoDB both support B-tree indexes, and those accustomed to working with indexes in MySQL can expect similar behavior in MongoDB. MySQL supports both joins and transactions, so if you must use SQL or if you require transactions, then you'll need to use MySQL or another RDBMS. That said, MongoDB's document model is often rich enough to represent objects without requiring joins. And its updates can be applied atomically to individual documents, providing a subset of what's possible with traditional transactions. Both MongoDB and MySQL support replication. As for scalability, MongoDB has been designed to scale horizontally, with sharding and failover handled automatically. Any sharding on MySQL has to be managed manually, and given the complexity involved, it's more common to see a vertically scaled MySQL system.

DOCUMENT DATABASES

Few databases identify themselves as document databases. As of this writing, the only well-known document database apart from 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, which is more involved than 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. Here I provide a few broadly defined use cases for MongoDB and give some examples of its use in production.¹¹

¹⁰ I'm using MySQL here generically, since the features I'm describing apply to most relational databases.

¹¹ For an up-to-date list of MongoDB production deployments, see <http://mng.bz/z2CH>.

WEB APPLICATIONS

MongoDB is well suited as a primary data store 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 does it benefit 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.

In production, MongoDB has proven capable of managing all aspects of an app, from the primary data domain to add-ons such as logging and real-time analytics. This is the case with *The Business Insider (TBE)*, which has used MongoDB as its primary data store since January 2008. TBE 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 also processes and stores real-time analytics data. These analytics are used by TBE to generate dynamic heat maps indicating click-through rates for the various news stories. The site doesn't host enough data to warrant sharding yet, but it does use replica sets to ensure automatic failover in the event of an outage.

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 Shutterstock 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.

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, an ORM. Thus, MongoDB often complements projects with shorter development cycles and agile, mid-sized teams.

ANALYTICS AND LOGGING

I alluded earlier to the idea that MongoDB works well for analytics and logging, and the number of application using MongoDB for these is growing fast. 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, often useful for logging, feature fixed allocation, which lets them age out automatically. Storing

logging data in a database, as compared with the file system, provides easier organization and much greater query power. Now, instead of using `grep` or a custom log search utility, users can employ the MongoDB query language they know and love to examine log output.

CACHING

A data model that allows for a more holistic representation of objects, combined with faster average query speeds, frequently allows MongoDB to be run in place of the more traditional MySQL/memcached duo. For example, TBE, mentioned earlier, has been able to dispense with memcached, serving page requests directly from MongoDB.

VARIABLE SCHEMAS

Look at this code example:¹²

```
curl https://stream.twitter.com/1/statuses/sample.json -umongodb:secret  
| mongoimport -c tweets
```

Here you're pulling down a small sample of the Twitter stream and piping that directly into a MongoDB collection. Since the stream produces JSON documents, there's no need to munge the data before sending it to 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. So you can immediately operate on the data, whether you want to query, index, or perform a map-reduce aggregation on it. And there's 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. If you can't possibly know the structure of your data before you store it, then MongoDB's lack of schema constraints may greatly 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. Some limitations should be noted before building a real-world application on MongoDB and running in production. Most of these are consequences of MongoDB's use of memory-mapped files.

First, MongoDB should usually be run on 64-bit machines. 32-bit systems are capable of addressing only 4 GB of memory. Acknowledging that typically half of this memory will be allocated by the operating system and program processes, this leaves just 2 GB of memory on which to map the data files. So if you're running 32-bit, and if you have even a modest number of indexes defined, you'll be strictly limited to as little as 1.5 GB of data. Most production systems will require more than this, and so a 64-bit system will be necessary.¹³

¹² This idea comes from <http://mng.bz/52XI>. If you want to run this code, you'll need to replace `<code>-umongodb:secret</code>` with your own Twitter username and password.

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

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.

Finally, it's important to run MongoDB with replication, especially if you're not running with journaling enabled. Because MongoDB uses memory-mapped files, any unclean shutdown of a `mongod` not running with journaling may result in corruption. Therefore, it's necessary in this case to have a replicated backup available for failover. This is good advice for any database—it'd be imprudent not to do likewise with any serious MySQL deployment—but it's especially important with nonjournaled MongoDB.

1.6 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. After all, 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 be doing 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 you'll build a simple MongoDB-based application in Ruby.

MongoDB through the JavaScript shell



In this chapter

- CRUD operations in the MongoDB shell
- Building indexes and using `explain()`
- 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 end the chapter by exploring some basic administrative commands, and I'll suggest a few ways of getting help as you continue on. 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.

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. Where it differs from similar tools is 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. 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.

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. Installation instructions can be found 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 for documents, collections, and the database's particular query language. Think of the following walkthrough as a practical introduction to MongoDB.

We'll begin by getting the shell up and running. Then we'll look at 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, you'll learn how to clear and drop collections.

2.1.1 *Starting the shell*

If you've followed the instructions in appendix A, you should have a working MongoDB installation on your computer. Make sure you have 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 the screenshot in figure 2.1. The shell heading displays the version of MongoDB you're running along with some additional information about the currently selected database.

If you know some JavaScript, you can start entering code and exploring the shell right away. Otherwise, read on to see how to insert your first datum.



Figure 2.1 The MongoDB JavaScript shell on startup

2.1.2 Inserts and queries

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.

ON CREATING DATABASES AND COLLECTIONS You may be wondering how we 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 need not be defined in advance, individual collections and databases can be created at runtime. This can lead to a simplified and accelerated development process, and essentially facilitates dynamic namespace allocation, which can frequently be useful. 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. Since you're using a JavaScript shell, your documents will be specified in JSON (JavaScript Object Notation). For instance, the simplest imaginable document describing a user might look like this:

```
{username: "jones"}
```

The document contains a single key and value for storing Jones's username. 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"})
```

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, then you've just saved your first document. You can issue a simple query to verify that the document has been saved:

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

The response will look something like this:

```
{ _id : ObjectId("4bf9bec50e32f82523389314"), username : "smith" }
```

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, then a special MongoDB object ID will be generated and added to the document at that time. The object ID 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 hard requirement for the field.

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

```
> db.users.save({username: "jones"})
```

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

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

Now that you have more than one document in the collection, we can look at some slightly more sophisticated queries. Like before, you can still query for all the documents in the collection:

```
> db.users.find()
{ _id : ObjectId("4bf9bec50e32f82523389314"), username : "smith" }
{ _id : ObjectId("4bf9bec90e32f82523389315"), username : "jones" }
```

But 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 so:

```
> db.users.find({username: "jones"})
{ _id : ObjectId("4bf9bec90e32f82523389315"), username : "jones" }
```

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

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

2.1.3 **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. There are two styles of modification; in this section we're going to focus on *targeted modifications*, which are most representative of MongoDB's distinct features.

To take an example, 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"}})
```

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`. If you now issue a query, you'll see that the document has been updated accordingly:

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("4bf9ec440e32f82523389316"),
  "country" : "Canada", username : "smith" }
```

If the user later decides that she no longer wants her country stored in her profile, she can remove the value just as easily using the `$unset` operator:

```
> db.users.update({username: "smith"}, {$unset: {country: 1}})
```

Let's enrich this example. You're representing your data with documents, which, as you saw in chapter 1, can contain complex data structures. So let's suppose that, in addition to storing profile information, your users can also 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 `$set` operator should come to mind. Note in this example that you're practically rewriting the document and that this is a perfectly acceptable use of `$set`:

```
> db.users.update( {username: "smith"},
{ $set: {favorites:
  {
    cities: ["Chicago", "Cheyenne"],
    movies: ["Casablanca", "The Sting"]
  }
}
})
```

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

```
db.users.update( {username: "jones"},
{"$set": {favorites:
  {
    movies: ["Casablanca", "Rocky"]
  }
}
})
```

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

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

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. You can accomplish this kind of query using a special dot notation. Suppose that you want to find all users who like the movie *Casablanca*. Such a query looks like this:

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

You could conceivably use the `$set` operator again, but that would require you to rewrite and send the entire array of movies. Since all you want to do is add an element to the list, you're better off using either `$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 )
```

Most of this should be decipherable. The first argument, a query selector, 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`, can be ignored for now. 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, then you must be explicit about that. Because you want your update to apply to both `smith` and `jones`, the multi-update is necessary.

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

2.1.4 **Deleting data**

You now 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, 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 pretty straightforward:

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

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` and `TRUNCATE TABLE` directives.

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.

2.2 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 clear the need for them; 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 only makes sense if you have a collection with many documents. So you'll add 200,000 simple documents to a `numbers` collection. Since the MongoDB shell is also a JavaScript interpreter, the code to accomplish this is simple:

```
for(i=0; i<200000; i++) {
  db.numbers.save({num: i});
}
```

This is 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()
200000

> db.numbers.find()
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830a"), "num" : 0 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830b"), "num" : 1 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830c"), "num" : 2 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830d"), "num" : 3 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830e"), "num" : 4 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830f"), "num" : 5 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8310"), "num" : 6 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8311"), "num" : 7 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8312"), "num" : 8 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8313"), "num" : 9 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8314"), "num" : 10 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8315"), "num" : 11 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8316"), "num" : 12 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8317"), "num" : 13 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8318"), "num" : 14 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8319"), "num" : 15 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831a"), "num" : 16 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831b"), "num" : 17 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831c"), "num" : 18 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831d"), "num" : 19 }
has more
```

The `count()` command shows that you've inserted 200,000 documents. The subsequent query displays the first 20 results. You can display additional results with the `it` command:

```
> it
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831e"), "num" : 20 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831f"), "num" : 21 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8320"), "num" : 22 }
...
```

The `it` command instructs the shell to return the next result set.¹

With a sizeable set of documents available, let's try a few 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("4bfbf132dba1aa7c30ac84fe"), "num" : 500 }
```

But more interestingly, you can also issue range queries using the special `$gt` and `$lt` operators. You first saw these operators in chapter 1. (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": 199995 }} )
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcade"), "num" : 199996 }
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcadf"), "num" : 199997 }
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcae0"), "num" : 199998 }
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcae1"), "num" : 199999 }
```

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

```
> db.numbers.find( {num: { "$gt": 20, "$lt": 25 }} )
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831f"), "num" : 21 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8320"), "num" : 22 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8321"), "num" : 23 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8322"), "num" : 24 }
```

You can see that by using a simple JSON document, you're able to specify a sophisticated range query much in the way you might do in SQL. Because `$gt` and `$lt` are just two of a host of special keywords that comprise the MongoDB query language, you'll be seeing many more example queries in later chapters.

Of course, queries like this are of little value if they're not also efficient. In the next section, we'll start thinking about query efficiency by exploring MongoDB's indexing features.

¹ 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's not 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.

2.2.2 Indexing and explain()

If you've spent time working with relational databases, you're probably familiar with SQL's EXPLAIN. EXPLAIN describes query paths and allows developers to diagnose slow operations by determining which indexes a query has used. 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": 199995 }} ).explain()
```

The result should look something like what you see in the following listing.

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

```
{
  "cursor" : "BasicCursor",
  "nscanned" : 200000,
  "nscannedObjects" : 200000,
  "n" : 4,
  "millis" : 171,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : { }
}
```

Examining the `explain()` output, you may be surprised to see that the query engine has to scan the entire collection, all 200,000 documents (`nscanned`), to return just four results (`n`). The `BasicCursor` cursor type verifies that this query hasn't used an index to return the result set. 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 171 milliseconds noted here.

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

```
> db.numbers.ensureIndex({num: 1})
```

As with other MongoDB operations, such as queries and updates, you pass a document to the `ensureIndex()` 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()
[
  {
```



```

    "name" : "_id_",
    "ns" : "tutorial.numbers",
    "key" : {
      "_id" : 1
    }
  },
  {
    "_id" : ObjectId("4bfc646b2f95a56b5581efd3"),
    "ns" : "tutorial.numbers",
    "key" : {
      "num" : 1
    },
    "name" : "num_1"
  }
]

```

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 just created on `num`.

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

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

```

> db.numbers.find({num: {"$gt": 199995 }}).explain()
{
  "cursor" : "BtreeCursor num_1",
  "indexBounds" : [
    [
      {
        "num" : 199995
      },
      {
        "num" : 1.7976931348623157e+308
      }
    ]
  ],
  "nscanned" : 5,
  "nscannedObjects" : 4,
  "n" : 4,
  "millis" : 0
}

```

Now that the query utilizes the index on `num`, it scans only the five documents pertaining to the query. This reduces the total time to serve the query from 171 ms to less than 1 ms.

If this example intrigues you, be sure to check out chapter 7, which is devoted to indexing and query optimization. Now we'll proceed now to look at the basic administrative commands required to get information about our MongoDB instance. You'll also learn some 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 I’ll present some 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 you 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. I’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, I’ll point out some 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
local
test
tutorial
```

`show collections` displays a list of all the collections defined on the current database.² 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. You can query this collection directly, but its output is more easily viewed using the `getIndexes()` method, as we saw earlier.

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

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

² You can also enter the more succinct `show tables`.

```

"collections" : 4,
"objects" : 200012,
"dataSize" : 7200832,
"storageSize" : 21258496,
"numExtents" : 11,
"indexes" : 3,
"indexSize" : 27992064,
"ok" : 1
}

```

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

```

> db.numbers.stats()
{
  "ns" : "tutorial.numbers",
  "count" : 200000,
  "size" : 7200000,
  "storageSize" : 21250304,
  "numExtents" : 8,
  "nindexes" : 2,
  "lastExtentSize" : 10066176,
  "paddingFactor" : 1,
  "flags" : 1,
  "totalIndexSize" : 27983872,
  "indexSizes" : {
    "_id_" : 21307392,
    "num_1" : 6676480
  },
  "ok" : 1
}

```

Some of the values provided in these result documents are useful only in complicated debugging 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, delete, 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 map-reduce.

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 actually works. That's not hard to find out because the MongoDB shell will print the implementation of any method whose executing parentheses are omitted. So instead of running the command like this

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

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

```
> db.runCommand
function (obj) {
  if (typeof obj == "string") {
    var n = {};
    n[obj] = 1;
    obj = n;
  }
  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 just 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 since you'll likely be spending 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 database objects. You'll find a similar list of methods for operating on collections by running `db.foo.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.foo.get
db.foo.getCollection(    db.foo.getIndexSpecs(    db.foo.getName(
```

```

db.foo.getDB(           db.foo.getIndexes(           db.foo.getShardVersion(
db.foo.getFullName(     db.foo.getIndices(
db.foo.getIndexKeys(     db.foo.getMongo(

```

If you're more ambitious, and 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. Simply 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) {
  if (obj == null || typeof obj == "undefined") {
    throw "can't save a null";
  }
  if (typeof obj._id == "undefined") {
    obj._id = new ObjectId;
    return this.insert(obj);
  } else {
    return this.update({_id:obj._id}, obj, true);
  }
}

```

Read the function definition closely, and you'll see that `save()` is merely a wrapper for `insert()` and `update()`. 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 at hand 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 seen a real-life 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, in the next chapter, we're going from a carefree data playground to a real-world data workshop. 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.