MongoDB
IN ACTION

SECOND EDITION

Kyle Banker
Peter Bakkum
Shaun Verch
Doug Garrett
Tim Hawkins

MEAP

MANNING

Covers MongoDB version 3.0

MEAP Edition
Manning Early Access Program
MongoDB in Action
Second Edition
Version 16

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *MongoDB in Action, 2nd edition*. This new edition reflects our commitment to keeping our books up to date and interesting. We have taken a good look at MongoDB technology as it has expanded and improved; this new edition will give a MongoDB beginner a flying start in MongoDB operations and development.

Although this is a new edition, the book has retained its basic structure. In this MEAP release, we have started out with making chapters 1 to 4 conform to the newest MongoDBversion releases. Chapter 1 introduces the features of MongoDB that are familiar from version 1.0, as well as those that have been introduced in versions 1.8 and 2.0. The aggregation framework and text searching are introduced for the first time as well.

In Chapter 2 we are asking the reader to interact with the Javascript shell—to get to know MongoDB's document data model in detail for the first time. By chapter 3, you will be making a start with writing MongoDB applications. This concludes the introduction to MongoDB.

At the beginning of part II, chapter 4 dives deeply into the MongoDB document datamodel. Part II will also contain a chapter on the aggregation framework, while part III will handle text searching, replication and sharding.

—Kyle Banker, Peter Bakkum, Tim Hawkins, Doug Garrett, and Tim Hawkins

brief contents

Part 1

Getting started

Part 1 provides a broad, practical introduction to MongoDB. It also introduces the JavaScript shell and the Ruby driver, both of which are used in examples throughout the book.

We've written this book with developers in mind, but it should be useful even if you're a casual user of MongoDB. Some programming experience will prove helpful in understanding the examples, though we focus most on MongoDB itself. If you've worked with relational databases in the past, great! We compare these to MongoDB often.

MongoDB version 3.0.x is the most recent MongoDB version at the time this book was written, but most of the discussion applies to previous versions of MongoDB (and presumably later versions). We usually mention it when a particular feature wasn't available in previous versions.

You'll use JavaScript for most examples, because MongoDB's JavaScript shell makes it easy for you to experiment with these queries. Ruby is a popular language among MongoDB users, and our examples show how the use of Ruby in real-world applications can take advantage of MongoDB. Rest assured, even if you're not a Ruby developer you can access MongoDB much the same way in other languages.

In chapter 1, you'll look at MongoDB's history, design goals, and application use cases. You'll also see what makes MongoDB unique as you compare it with other databases emerging in the "NoSQL" space.

In chapter 2, you'll become conversant in the language of MongoDB's shell. You'll learn the basics of MongoDB's query language, and you'll practice by creating, querying, updating, and deleting documents. The chapter also features some advanced shell tricks and MongoDB commands.

Chapter 3 introduces the MongoDB drivers and MongoDB's data format, BSON. Here you'll learn how to talk to the database through the Ruby programming language, and you'll build a simple application in Ruby demonstrating MongoDB's flexibility and query power.

To get the most out of this book, follow along and try out the examples. If you don't have MongoDB installed yet, appendix A can help you get it running on your machine.

1

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-normalized[1] 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. So when developers start advocating alternative datastores, questions about the viability and utility of these new technologies arose. Are these new datastores 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 the need 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

[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 your data, such as users and orders, into their own tables to reduce redundant storage of usernames.

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?

Perhaps the biggest reason developers use MongoDB isn't because of its scaling strategy, but rather 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 just storing a few fields of information about a user (he sounds cool). So 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 user 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 with 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 NoSQL[2] 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, just like any other database, and you should know about its limitations and its 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 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

[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.

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 idea 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 on 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 in listing 1.1. A JSON document needs double quotes everywhere except from 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'),                    #A
  title: 'Adventures in Databases',
  url: 'http://example.com/databases.txt',
  author: 'msmith',
  vote_count: 20,
  tags: ['databases', 'mongodb', 'indexing'],                   #1
  image: {                                                      #2
    url: 'http://example.com/db.jpg',
    caption: 'A database.',
    type: 'jpg',
    size: 75381,
    data: 'Binary'
  },
  comments: [                                                   #3
    {
      user: 'bjones',
      text: 'Interesting article.'
    },
    {
      user: 'sverch',
      text: 'Color me skeptical!'
    }
  ]
}
```

#A _id field, primary key

#1 Tags stored as array of strings
#2 Attribute pointing to another document
#3 Comments stored as array of comment objects

Listing 1.1 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 (#2). These latter constructs permit documents to represent a variety of rich data structures. You'll see that the sample document has a property, `tags` (#1), which stores the article's tags in an array. But even more interesting is the `comments` property (#3), which is an array of comment documents.

Internally, MongoDB stores documents in a format called Binary JSON, or BSON. BSON has a very 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 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 is going to require 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.

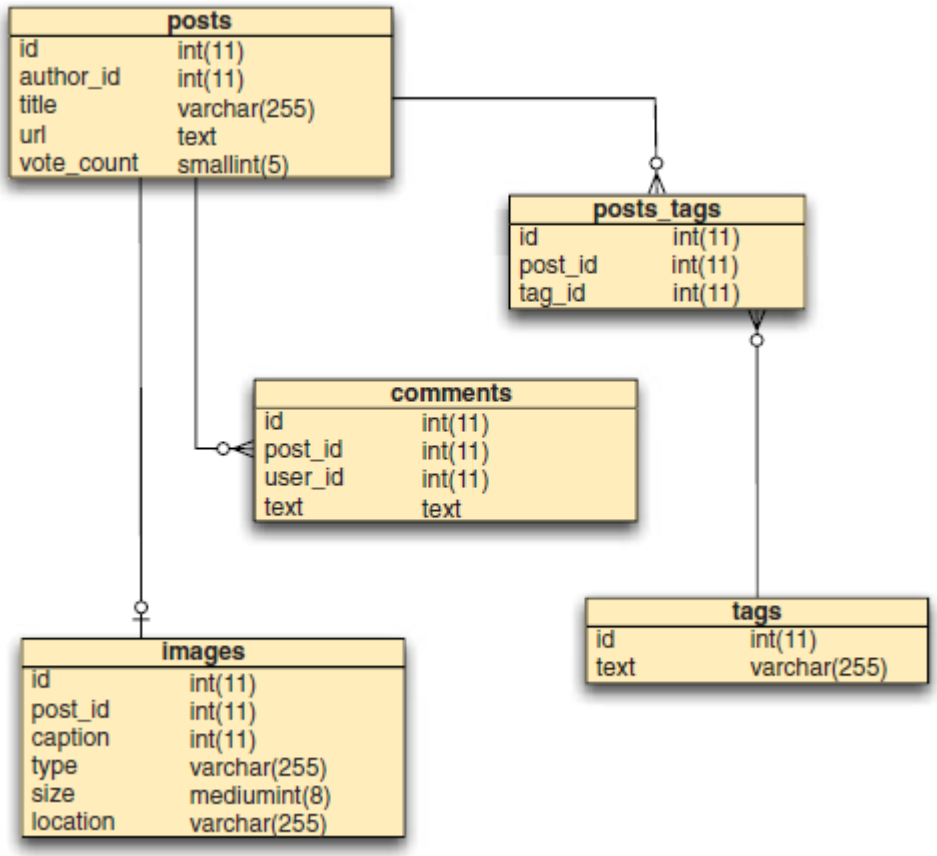

Figure 1.1 A basic relational data model for entries on a social news site. The line terminator that looks like a cross represents a one-to-one relationship, so there is only one row from the images table associated with a row from the posts table. The line terminator that branches apart represents a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

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.

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

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 schema-less 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 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.

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

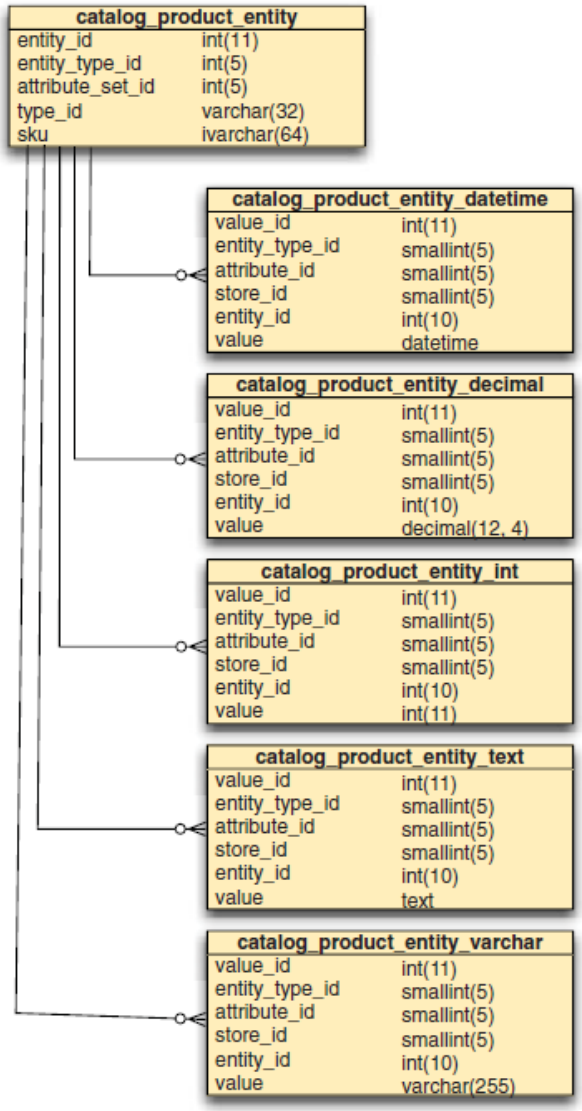

Figure 1.2 A portion of the schema for an e-commerce application. These tables facilitate dynamic attribute creation for products.

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'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 RDMBS; ascending, descending, unique, compound-key, hashed, text, and even geospatial indexes[4] 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.

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.

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

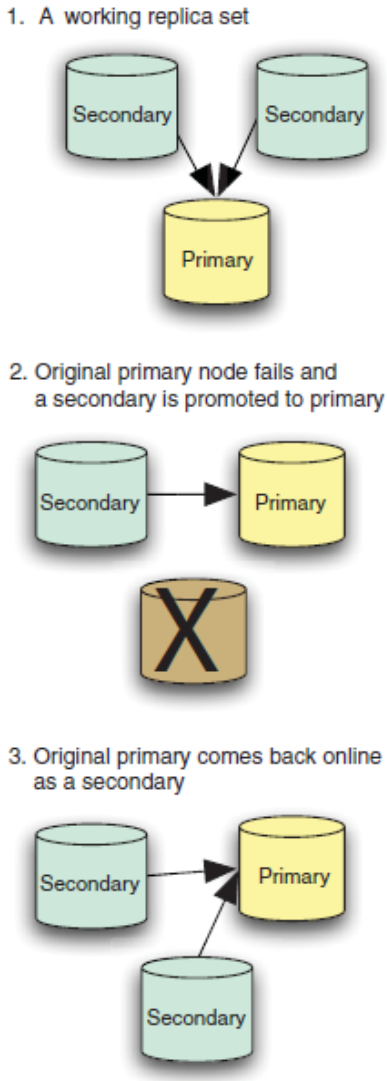

Figure 1.3 Automated failover with a replica set

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 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.

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 nowadays as MongoDB is 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.

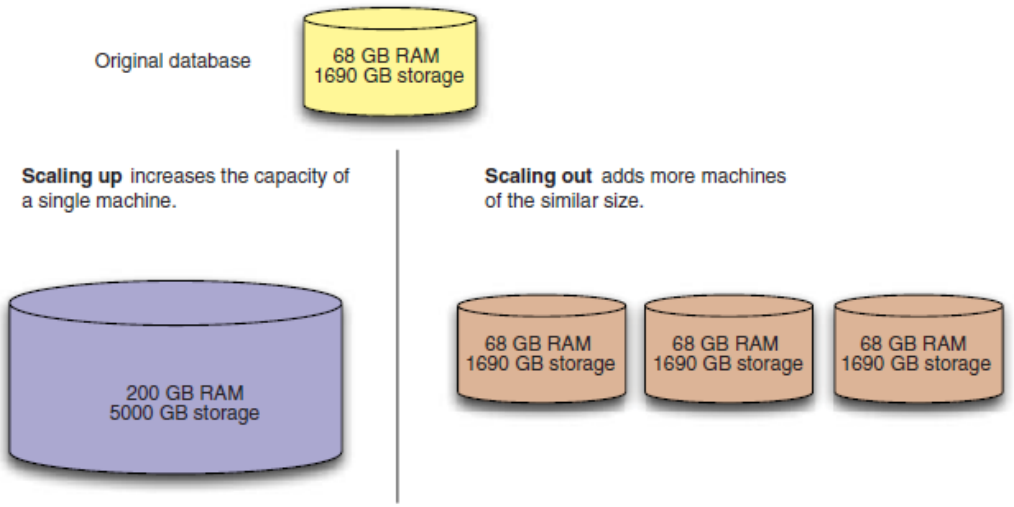

Figure 1.4 Horizontal versus vertical scaling

MongoDB was designed to make horizontal scaling manageable. It does so via a range-based partitioning mechanism, known as *sharding*, which automatically manages 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 object of some controversy. What this licensing means in practice is 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.

1.3.1 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 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`.

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

1.3.2 JavaScript shell

The MongoDB command shell is a JavaScript[6]-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 so:

```
> 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" }
```

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

[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.

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's probably a community-supported driver 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 TSV[7] data; this is useful if you need your data in widely supported formats. `mongoimport` can also be good for initial imports of large data sets, although before importing, it's often desirable to adjust the data model to take best advantage of MongoDB. In such cases, it's easier to import the data through one of the drivers using a custom script.
- `mongosniff`—A wire-sniffing tool for viewing operations sent to the database. It essentially translates the BSON going over the wire to human-readable shell statements.
- `mongostat`—Similar to `iostat`, this utility constantly polls MongoDB and the system to

[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.

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. 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 have a rich data model and a powerful query language. Because 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.

In terms of use cases, MongoDB is well suited as a primary datastore for web applications, for analytics and logging applications, and for 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 *column*. 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.

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]

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.

---

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

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, etc; 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 targeted more at developers, who write a query once to embed it in their application. Both models have their strengths and weaknesses, and sometimes it simply 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 a 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 very 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 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.

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.

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.

---

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

ANALYTICS AND LOGGING

We alluded earlier to the idea that MongoDB works well for analytics and logging, and the number of application 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 (its 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, and 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 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.

---

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

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; now 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 the MongoDB company.
- *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.

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

[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.

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 on 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, 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 at 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 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.