THE EXPERT'S VOICE® IN OPEN SOURCE

The Definitive Guide to MongoDB

The NoSQL Database for Cloud and Desktop Computing

Simplify the storage of complex data by creating fast and scalable databases

Eelco Plugge, Peter Membrey and Tim Hawkins

Apress®

CHAPTER 1

Introduction to MongoDB

Imagine a world where using a database is so simple that you soon forget you're even using it. Imagine a world where speed and scalability *just work*, and there's no need for complicated configuration or setup. Imagine being able to focus only on the task at hand, get things done, and then—just for a change—leave work on time. That might sound a bit fanciful, but MongoDB promises to help you accomplish all these things (and many more).

MongoDB (derived from the word *humongous*) is a relatively new breed of database that has no concept of tables, schemas, SQL, or rows. It doesn't have transactions, ACID compliance, joins, foreign keys, or many of the other features that tend to cause headaches in the early hours of the morning. In short, MongoDB is probably a very different database than what you're used to, especially if you've used a relational database management system (RDBMS) in the past. In fact, you might even be shaking your head in wonder at the lack of so-called "standard" features.

Fear not! In a few moments, you will learn about MongoDB's background, guiding principles, and why the MongoDB team made the design decisions that it did. We'll also take a whistle-stop tour of MongoDB's feature list, providing just enough detail to ensure you'll be completely hooked on this topic for the rest of the book.

We'll start things off by looking at the philosophy and ideas behind the creation of MongoDB, as well as some of the interesting and somewhat controversial design decisions. We'll explore the concept of document-orientated databases, how they fit together, and what their strengths and weaknesses are. We'll also explore JSON and examine how it applies to MongoDB. To wrap things up, we'll step through some of the notable features of MongoDB.

Reviewing the MongoDB Philosophy

Like all projects, MongoDB has a set of design philosophies that help guide its development. In this section, we'll review some of the database's founding principles.

Using the Right Tool for the Right Job

The most important of the philosophies that underpin MongoDB is the notion that *one size does not fit all*. For many years, traditional SQL databases (MongoDB is a document-orientated database) have been used for storing content of all types. It didn't matter whether the data was a good fit for the relational model (which is used in all RDBMS databases, such as MySQL, PostgresSQL, SQLite, Oracle, MS SQL Server, and so on); the data was stuffed in there, anyway. Part of the reason for this is that, generally speaking, it's much easier (and more secure) to read and write to a database than it is to write to a file system. If you pick up any book that teaches PHP (such as *PHP for Absolute Beginners* (Apress, 2009)) by Jason Lengstorf, you'll probably find that almost right away the database is used to store information, not the file system. It's just so much easier to do things that way. And while using a database as a storage bin works, developers always have to work against the flow. It's usually obvious when we're not using the database the way it was intended; anyone who has ever tried to store information with even slightly complex data, had to set up five tables, and then tried to pull it all together knows what I'm talking about!

The MongoDB team decided that it wasn't going to create another database that tries to do everything for everyone. Instead, the team wanted to create a database that worked with documents rather than rows, was blindingly fast, massively scalable, and easy to use. To do this, the team had to leave some features behind, which means that MongoDB is not an ideal candidate for certain situations. For example, its lack of transaction support means that you wouldn't want to use MongoDB to write an accounting application. That said, MongoDB might be perfect for part of the aforementioned application (such as storing complex data). That's not a problem though because there is no reason why you can't use a traditional RDBMS for the accounting components and MongoDB for the document storage. Such hybrid solutions are quite common, and you can see them in production apps such as Sourceforge.

Once you're comfortable with the idea that MongoDB may not solve all your problems (the coffee-making plug-in is still in development), you will discover that there are certain problems that MongoDB is a perfect fit for resolving, such as analytics (think a realtime Google Analytics for your website) and complex data structures (e.g., as blog posts and comments). If you're still not convinced that MongoDB is a serious database tool, feel free to skip ahead to the "Reviewing the Feature List" section, where you will find an impressive list of features for MongoDB.

■ **Note** The lack of transactions and other traditional database features doesn't mean that MongoDB is unstable or that it cannot be used for managing important data.

Another key concept behind MongoDB's design: There should always be more than one copy of the database. If a single database should fail, then it can simply be restored from the other servers. Because MongoDB aims to be as fast as possible, it takes some shortcuts that make it more difficult to recover from a crash. The developers believe that most serious crashes are likely to remove an entire computer from service anyway; this means that, even if the database were perfectly restored, it would still not be usable. Remember: MongoDB does not try to be everything to everyone. But for many things (such as building a web application), MongoDB can be an awesome tool for implementing your solution.

So now you know where MongoDB is coming from. It's not trying to be the best at everything, and it readily acknowledges that it's not for everyone. However, for those who do choose to use it, MongoDB provides a rich document-orientated database that's optimized for speed and scalability. It can also run nearly anywhere you might want to run it. MongoDB's website includes downloads for Linux, the Mac, Windows, and Solaris; it also includes various unofficial versions of the program that enable you to install it on Fedora or CentOS, among other platforms.

MongoDB succeeds at all these goals, and this is why using MongoDB (at least for me) is somewhat dream-like. You don't have to worry about squeezing your data into a table—just put the data together, and then pass it to MongoDB for handling.

Consider this real-world example. A recent application I worked on needed to store a set of eBay search results. There could be any number of results (up to 100 of them), and I needed an easy way to associate the results with the users in my database.

Had I been using MySQL, I would have had to design a table to store the data, write the code to store my results, and then write more code to piece it all back together again. This is a fairly common scenario and one most developers face on a regular basis. Normally, we just get on with it; however, for this project, I was using MongoDB and so things went a bit differently.

Specifically, I added this line of code:

```
request['ebay_results'] = ebay_results_array
collection.save(reqest)
```

In the preceding example, `request` is my document, `ebay_results` is the key, and `ebay_result_array` contains the results from eBay. The second line saves my changes. When I access this document in future, I will have the eBay results in exactly the same format as before. I don't need any SQL; I don't need to perform any conversions; nor do I need to create any new tables or write any special code—MongoDB just worked. It got out of the way, I finished my work early, and I got to go home on time.

Lacking Innate Support for Transactions

Another important design decision by MongoDB developers: The database does not include transactional semantics (the bit that offers guarantees about data consistency and storage). This is a solid tradeoff based on MongoDB's goal of being simple, fast, and scalable. Once you leave those heavyweight features at the door, it becomes much easier to scale horizontally.

Normally with a traditional RDBMS, you improve performance by buying a bigger, more powerful machine. This is scaling vertically but you can only take this so far. Horizontal scaling is where, rather than having one big machine, you have lots of less powerful small machines. Historically, clusters of servers like this were excellent for load balancing websites, but databases had always been a problem due to internal design limitations.

You might think this missing support constitutes a deal breaker; however, many people forget that one of the most popular table types in MySQL (`MYISAM`) doesn't support transactions, either. This fact hasn't stopped MySQL from becoming the dominant open-source database for well over a decade. As with most things when developing solutions, using MongoDB is going to be a matter of personal choice and whether the tradeoffs fit your project.

■ **Note** MongoDB offers durability when used in tandem with at least two servers, which is the recommended minimum for production deployments. It is possible to make the master server wait for the replica to confirm receipt of the data before the master server itself confirms the data has been accepted.

Although single server durability is not guaranteed, this may change in the future and is currently an area of active interest.

Drilling Down on JSON and How It Relates to MongoDB

JSON is more than a great way to exchange data; it's also a nice way to store data. An RDBMS is highly structured, with multiple files (tables) that store the individual pieces. MongoDB, on the other hand, stores everything together in a single document. MongoDB is like JSON in this way, and this model provides a rich and expressive way of storing data. Moreover, JSON effectively describes all the content in a given document, so there is no need to specify the structure of the document in advance. JSON is effectively schemaless because documents can be updated individually or changed independently of any other documents. As an added bonus, JSON also provides excellent performance by keeping all of the related data in one place.

MongoDB doesn't actually use JSON to store the data; rather, it uses an open data format developed by the MongoDB team called *BSON* (pronounced Bee-Son), which is short for Binary-JSON. For the most part, using BSON instead of JSON doesn't change how you will work with your data. BSON makes

MongoDB even faster by making it much easier for a computer to process and search documents. BSON also adds a couple of features that aren't available in standard JSON, including the ability to add types for handling binary data. We'll look at BSON in more depth later in the chapter when we cover the feature list.

The original specification for JSON can be found in RFC 4627, and it was written by Douglas Crockford. JSON allows complex data structures to be represented in a simple, human-readable text format that is generally considered to be much easier to read and understand than XML. Like XML, JSON was envisaged as a way to exchange data between a web client (such as a browser) and web applications. When combined with the rich way that it can describe objects, its simplicity has made it the exchange format of choice for the majority of developers.

You might wonder what is meant here by *complex data structures*. Historically, data was exchanged using the comma-separated values (CSV) format (indeed, this approach remains very common today). CSV is a simple text format that separates rows with a new line and fields with a comma. For example, a CSV file might look like this:

```
Membrey, Peter, +852 1234 5678
Thielen, Wouter, +81 1234 5678
```

A human can look at this information and see quite quickly what information is being communicated. Or maybe not—is that number in the third column a phone number or a fax number? It might even be the number for a pager. To combat this, CSV files often have a header field, where the first row defines what comes in the file. The following snippet takes the previous example one step further:

```
Surname, Forename, Phone Number
Membrey, Peter, +852 1234 5678
Thielen, Wouter, +81 1234 5678
```

Okay, that's a bit better. But now assume you have more than one phone number. You could add another field for an office phone number, but you face a new set of issues if you want several office phone numbers. And you face yet another set of issues if you also want to incorporate multiple e-mail addresses. Most people have more than one, and these addresses can't usually be neatly defined as either home or work. Suddenly, CSV starts to show its limitations. CSV files are only good for storing data that is flat and doesn't have repeating values. Similarly, it's not uncommon for several CSV files to be provided, each with the separate bits of information. These files are then combined (usually in an RDBMS) to create the whole picture. As an example, a large retail company may receive CSV files from each of its stores at the end of each day. These files must be combined before the company can see how it performed on a given day. This process is not exactly straightforward, and it certainly increases chances of a mistake as the number of required files grows.

XML largely solves this problem, but using XML for most things is a bit like using a sledgehammer to crack a nut: it works, but it feels like overkill. The reason for this: XML is highly extensible. Rather than define a particular data format, XML defines how you define a data format. This can be useful when you need to exchange complex and highly structured data; however, for simple data exchange, this often results in too much work. Indeed, this scenario is the source of the phrase "XML hell."

JSON provides a happy medium. Unlike CSV, it can store structured content; but unlike XML, JSON makes it easy to understand and simple to use. Let's revisit the previous example; however, this time you will use JSON rather than CSV:

```
{
    "forename": "Peter",
    "surname": "Membrey",
    "phone_numbers": [
        "+852 1234 5678",
        "+44 1234 565 555"
    ]
}
```

In the preceding example, each JSON object (or document) contains all the information needed to understand it. If you look at phone_numbers, you can see that you have a list of different numbers. This list can be as large as you want. You could also be more specific about the type of number being recorded, as in this example:

```
{
    "forename": "Peter",
    "surname": "Membrey",
    "numbers": [
        {
            "phone": "+852 1234 5678"
        },
        {
            "fax": "+44 1234 565 555"
        }
    ]
}
```

The preceding example improves on things a bit more. Now you can clearly see what each number is for. JSON is extremely expressive, and, although it's quite easy to write JSON by hand, it is usually generated automatically in software. For example, Python includes a module called `simplejson` that takes existing Python objects and automatically converts them to JSON. Because JSON is supported and used on so many platforms, it is an ideal choice for exchanging data.

When you add items such as the list of phone numbers, you are actually creating what is known as an *embedded document*. This happens whenever you add complex content such as a list (or *array*, to use the term favored in JSON). Generally speaking, there is also a logical distinction too. For example, a `Person` document might have several `Address` documents embedded inside it. Similarly, an `Invoice` document might have numerous `LineItem` documents embedded inside it. Of course, the embedded `Address` document could also have its own embedded document inside it that contains phone numbers, for example.

Whether you choose to embed a particular document is determined when you decide how to store your information. This is usually referred to as *schema design*. It might seem odd to refer to schema design when MongoDB is considered a schemaless database. However, while MongoDB doesn't force you to create a schema or enforce one that you create, you do still need to think about how your data fits together. We'll look at this in more depth in Chapter 3.

Adopting a Non-Relational Approach

Improving performance with a relational database is usually straightforward: you buy a bigger, faster server. And this works great until you reach the point where there isn't a bigger server available to buy. At that point, the only option is to spread out to two servers. This might sound easy, but it is a stumbling block for most databases. For example, neither MySQL nor PostgresSQL can run a single database on two servers, where both servers can both read and write data (this is often referred to as an *active/active cluster*). And although Oracle can do this with its impressive Real Application Clusters (RAC) architecture, you can expect to take out a mortgage if you want to use that solution—implementing a RAC-based solution requires multiple servers, shared storage, and several software licenses.

You might wonder why having an active/active cluster on two databases is so difficult. When you query your database, the database has to find all the relevant data and link it all together. RDBMS solutions feature many ingenious ways to improve performance, but they all rely on having a complete picture of the data available. And this is where you hit a wall: this approach simply doesn't work when half the data is on another server.

Of course, you might have a small database that simply gets lots of requests, so you just need to share the workload. Unfortunately, here you hit another wall. You need to ensure that data written to the first server is available to the second server. And you face additional issues if updates are made on two separate masters simultaneously. For example, you need to determine which update is the correct one. Another problem you can encounter: someone might query for information on the second server that has just been written to the first server, but that information hasn't been updated yet on the second server. When you consider all these issues, it becomes easy to see why the Oracle solution is so expensive—these problems are extremely hard to address.

MongoDB solves this problem in a very clever way—it avoids it completely. Recall that MongoDB stores data in BSON documents, so the data is self-contained. That is, although similar documents are stored together, individual documents aren't made up of relationships. This means everything you need is all in one place. Because queries in MongoDB look for specific keys and values in a document, this information can be easily spread across as many servers as you have available. Each server checks the content it has and returns the result. This effectively allows almost linear scalability and performance. As an added bonus, it doesn't even require that you take out a new mortgage to pay for this functionality.

Admittedly, MongoDB does not offer *master/master replication*, where two separate servers can both accept write requests. However, it does have sharding, which allows data to split across multiple machines, with each machine responsible for updating different parts of the dataset. The benefit of this design is that, while some solutions allow two master databases, MongoDB can potentially scale to hundreds of machines as easily as it can run on two.

■ **Note** We just mentioned that MongoDB doesn't support master-master replication; however, that's not entirely true. It turns out it is possible to use MongoDB in a master-master configuration; however, this approach is not recommended, so we won't discuss it further in this book. If you're curious, you can find additional details on this subject on the MongoDB website at `www.mongodb.org/display/DOCS/Master+Master+Replication`.

Opting for Performance vs. Features

Performance is important, but MongoDB also provides a large feature set. We've already discussed some of the features MongoDB doesn't implement, and you might be somewhat skeptical of the claim that MongoDB achieves its impressive performance partly by judiciously excising certain features common to other databases. However, there are analogous database systems available that are extremely fast, but also extremely limited, such as those that implement a key / value store.

A perfect example is *memcached*. This application was written to provide high-speed data caching, and it is mind-numbingly fast. When used to cache website content, it can speed up an application many times over. This application is used by extremely large websites, such as Facebook and LiveJournal.

The catch is that this application has two significant shortcomings. First, it is a memory-only database. If the power goes out, then all the data is lost. Second, you can't actually search for data using memcached; you can only request specific keys.

These might sound like serious limitations; however, you must remember the problems that memcached is designed to solve. First and foremost, memcached is a data-cache. That is, it's not supposed to be a permanent data store, but only to provide a caching layer for your existing database. When you build a dynamic web page, you generally request very specific data (such as the current top ten articles). This means you can specifically ask memcached for that data—there is no need to perform a search. If the cache is out-of-date or empty, you would query your database as normal, build up the data, and then store it in memcached for future use.

Once you accept these limitations, you can see how memcached offers superb performance by implementing a very limited feature set. This performance, by the way, is unmatched by that of a traditional database. That said, memcached certainly can't replace an RDBMS. The important thing to keep in mind is that it's not supposed to.

Compared to memcached, MongoDB is itself feature rich. To be useful, MongoDB must offer a strong feature set, such as being able to search for specific documents. It must also be able to store those documents on disk, so that they can survive a reboot. Fortunately, MongoDB provides enough features for it to be a strong contender for most web applications and many other types of applications, as well.

Like memcached, MongoDB is not a one-size-fits-all database. As is usually the case in computing, tradeoffs must be made to achieve the intended goals of the application.

Running the Database Anywhere

MongoDB is written in C++, which makes it relatively easy to port and/or run the application practically anywhere. Currently, binaries can be downloaded from the MongoDB website for Linux, the Mac, Windows, and Solaris. There are also various unofficial versions available for Fedora and CentOS, among other platforms. You can even download the source code and build your own MongoDB, although it is recommended that you use the provided binaries wherever possible. All the binaries are available in both 32-bit and 64-bit versions.

■ **Caution** The 32-bit version of MongoDB is limited to databases of 2GB or less. This is because, internally, MongoDB uses memory-mapped files to achieve high performance. Anything larger than 2GB on a 32 bit system would require some fancy footwork that wouldn't be all that fast and would also complicate the application's code. The official stance on this limitation is that 64-bit environments are easily available; therefore, increasing code complexity is not a good tradeoff. The 64-bit version for all intents and purposes has no such restriction.

MongoDB's modest requirements allow it to run on high-powered servers, virtual machines, or even to power cloud-based applications. By keeping things simple and focusing on speed and efficiency, MongoDB provides solid performance wherever you choose to deploy it.

Fitting Everything Together

Before we look at MongoDB's feature list, we need to review a few basic terms. MongoDB doesn't require much in the way of specialized knowledge to get started, and many of the terms specific to MongoDB can be loosely translated to RDBMS equivalents that you are probably already familiar with. Don't worry, though: we'll explain each term fully. Even if you're not familiar with standard database terminology, you will still be able to follow along easily.

Generating or Creating a Key

A document represents the unit of storage in MongoDB. In an RDBMS, this would be called a row. However, documents are much more than rows because they can store complex information such as lists, dictionaries, and even lists of dictionaries. In contrast to a traditional database where a row is fixed, a document in MongoDB can be made up of any number of keys and values (you'll learn more about this in the next section). Ultimately, a *key* is nothing more than a label; it is roughly equivalent to the name you might give to a column in an RDBMS. You use a key to reference pieces of data inside your document.

In a relational database, there should always be some way to uniquely identify a given record; otherwise it becomes impossible to refer to a specific row. To that end, you are supposed to include a field that holds a unique value (called a *primary key*) or a collection of fields that can uniquely identify the given row (called a *compound primary key*).

MongoDB requires that each document have a unique identifier for much the same reason; in MongoDB, this identifier is called `_id`. Unless you specify a value for this field, MongoDB will generate a unique value for you. Even in the well-established world of RDBMS databases, opinion is divided as to whether you should use a unique key provided by the database or generate a unique key yourself. Recently, it has become more popular to allow the database to create the key for you.

The reason for this: human-created unique numbers such as car registration numbers have a nasty habit of changing. For example, in 2001, the United Kingdom implemented a new number plate scheme that was completely different from the previous system. It happens that MongoDB can cope with this type of change perfectly well; however, chances are that you would need to do some careful thinking if you used the registration plate as your primary key. A similar scenario may have occurred when ISBN numbers were upgraded from 10 digits to 13.

That said, most developers who use MongoDB seem to prefer creating their own unique keys, taking it upon themselves to ensure that the number will remain unique. However, as is the case when working with RDBMS databases, which approach you take mostly comes down to personal preference. I personally prefer to use a database-provided value because it means I can be sure my key is unique and independent of anything else. Others, as noted, prefer to provide their own keys.

Ultimately, you must decide what works best for you. If you are confident that your key is unique (and likely to remain unchanged), then you should probably feel free to use it. If you're unsure about your key's uniqueness or you don't want to worry about it, then you can simply use the default key provided by MongoDB.

Using Keys and Values

Documents are made up of keys and values. Let's take another look at the example discussed previously in this chapter:

```
{
    "forename": "Peter",
    "surname": "Membrey",
    "phone_numbers": [
        "+852 1234 5678",
        "+44 1234 565 555"
    ]
}
```

Keys and values always come in pairs. Unlike an RDBMS, where all fields must have a value, even if it's `NULL` (somewhat paradoxically, this means *unknown*), MongoDB doesn't require that a document have a particular value. For example, if you don't know the phone number for a particular document, you simply leave it out. A popular analogy for this sort of thing is a business card. If you have a fax number, you usually put it on your business card; however, if you don't have one, you don't write: "Fax number: none." Instead, you simply leave the information out. If the key value pair isn't included in a MongoDB document, it is assumed that it doesn't exist.

Implementing Collections

Collections are somewhat analogous to tables, but they are far less rigid. A collection is a lot like a box with a label on it. You might have a box at home labeled “DVDs” into which you put, well, your DVDs. This makes sense, but there is nothing stopping you from putting CDs or even tapes into this box if you wanted to. In an RDBMS, tables are strictly defined, and you can only put designated items into the table. In MongoDB, a collection is simply that: a collection of similar items. The items don’t have to be similar (MongoDB is inherently flexible); however, once we start looking at indexing and more advanced queries, you’ll soon see the benefits of placing similar items in a collection.

While you could mix various items together in a collection, there’s little need to do so. Had the collection been called `media`, then all of the DVDs, CDs, and tapes would be at home there. After all, these items all have things in common, such as an artist name, a release date, and content. In other words, it really does depend on your application whether certain documents should be stored in the same collection. Performance-wise, having multiple collections is no slower than having only one collection. Remember: MongoDB is about making your life easier, so you should do whatever feels right to you.

Last but not least, collections are effectively created on demand. Specifically, a collection is created when you first attempt to save a document that references it. This means that you could create collections on demand (not that you necessarily should). Because MongoDB also lets you create indexes and perform other database-level commands dynamically, you can use leverage this behavior to build some very dynamic applications.

Understanding Databases

Perhaps the easiest way to think of a database is as a collection of collections. Like collections, databases are created on demand. This means that it’s easy to create a database for each customer—your application code can even do it for you. You can do this with databases other than MongoDB, as well; however, creating databases in this manner with MongoDB is a very natural process. That said, just because you can create a database in this manner doesn’t mean you have to or even that you should. All the same, you have that power if you want to exercise it.

Reviewing the Feature List

Now that you understand what MongoDB is and what it offers, it’s time to run through its feature list. You can find a complete list of MongoDB’s features on the database’s website at `www.mongodb.org/`; be sure to visit this site for an up-to-date list of them. The feature list in this chapter covers a fair bit of material that goes on behind the scenes, but you don’t need to be familiar with every feature listed to use MongoDB itself. In other words, if you feel your eyes beginning to close as you review this list, feel free to jump to the end of the section!

Using Document-Orientated Storage (BSON)

We’ve already discussed MongoDB’s document-orientated design. We’ve also briefly touched on BSON. As you learned, JSON makes it much easier to store and retrieve documents in their real form, effectively removing the need for any sort of mapper or special conversion code. The fact that this feature also makes it much easier for MongoDB to scale up is icing on the cake.

BSON is an open standard; you can find its specification at `http://bsonspec.org/`. When people hear that BSON is a binary form of JSON, they expect it to take up much less room than text-based JSON.

However, this isn't necessarily the case; indeed, there are many cases where the BSON version takes up more space than its JSON equivalent.

You might wonder why you should use BSON at all. After all, CouchDB (another powerful document-orientated database) uses pure JSON, and it's reasonable to wonder whether it's worth the trouble of converting documents back and forth between BSON and JSON.

First, we must remember that MongoDB is designed to be fast, rather than space efficient. This doesn't mean that MongoDB wastes space (it doesn't); however, a small bit of overhead in storing a document is perfectly acceptable if that makes it faster to process the data (which it does). In short, BSON is much easier to *traverse* (i.e., to look through) and index very quickly. Although BSON requires slightly more disk space than JSON, this extra space is unlikely to be a problem because disks are cheap, and MongoDB can scale across machines. The tradeoff in this case is quite reasonable: you exchange a bit of extra disk space for better query and indexing performance.

The second key benefit to using BSON is that it is easy and quick to convert BSON to a programming language's native data format. If the data were stored in pure JSON, a relatively high-level conversion would need to take place. There are MongoDB drivers for a large number of programming languages (such as Python, Ruby, PHP, C, C++ and C#), and each works slightly differently. Using a simple binary format, native data structures can be quickly built for each language, without requiring that you first process JSON. This makes the code simpler and faster, both of which are in keeping with MongoDB's stated goals.

BSON also provides some extensions to JSON. For example, it enables you to store binary data and incorporates a specific date type. Thus, while BSON can store any JSON document, a valid BSON document may not be valid JSON. This doesn't matter because each language has its own driver that converts data to and from BSON without needing to use JSON as an intermediary language.

At the end of the day, BSON is not likely to be a big factor in how you use MongoDB. Like all great tools, MongoDB will quietly sit in the background and do what it needs to do. Apart from possibly using a graphical tool to look at your data, you will generally work in your native language and let the driver worry about persisting to MongoDB.

Supporting Dynamic Queries

MongoDB's support for dynamic queries means that you can run a query without planning for it in advance. This is similar to being able to run SQL queries against an RDBMS. You might wonder why this is listed as a feature; surely this is something that every database supports—right?

Actually, no. For example, CouchDB (which is generally considered as MongoDB's biggest "competitor") doesn't support dynamic queries. This is because CouchDB has come up with a completely new (and admittedly exciting) way of thinking about data. A traditional RDBMS has static data and dynamic queries. This means that the structure of the data is fixed in advance—tables must be defined, and each row has to fit into that structure. Because the database knows in advance how the data is structured, it can make certain assumptions and optimizations that enable fast dynamic queries.

CouchDB has turned this on its head. As a document-orientated database, CouchDB has no schema (i.e., it is *schemaless*), so the data is dynamic. However, the new idea here is that queries are static. That is, you define them in advance, before you can use them.

This isn't as bad as it might sound because many queries can be easily defined in advance. For example, a system that lets you search for a book will probably let you search by ISBN. In CouchDB, you would create an index that builds a list of all the ISBNs for all the documents. When you punch in an ISBN, the query is very fast because it doesn't actually need to search for any data. Whenever new data is added to the system, CouchDB will automatically update its index.

Technically, you can run a query against CouchDB without generating an index; in this case, however, CouchDB will have to create the index itself before it can process your query. This won't be a problem if you only have a hundred books; however, this will result in poor performance if you're filing hundreds of thousands of books because each query will generate the index again (and again). For this reason, the CouchDB team does not recommend dynamic queries—that is, queries that haven't been predefined—in production.

CouchDB also lets you write your queries as `map` and `reduce` functions. If that sounds like a lot of effort, then you're in good company; CouchDB has a somewhat severe learning curve. In fairness to CouchDB, an experienced programmer can probably pick it up quite quickly; for most people, however, the learning curve is probably severe enough that they won't bother with the tool.

Fortunately for us mere mortals, MongoDB is much easier to use. We'll cover how to use MongoDB in more detail throughout the book, but here's the short version: in MongoDB, you simply provide the parts of the document you want to match against, and MongoDB does the rest. MongoDB can do much more, however. For example, you won't find MongoDB lacking if you want to use `map` or `reduce` functions. At this same time, you can ease into using MongoDB; you don't have to know all the tool's advanced features up front.

Indexing Your Documents

MongoDB includes extensive support for indexing your documents. All documents are automatically indexed on the `_id` key. This is considered a special case because you cannot delete this index; it is what ensures that each value is unique. One of the benefits of this key is that you can be assured that each document is uniquely identifiable, something that isn't guaranteed by an RDBMS.

When you create your own indexes, you can decide whether you want them to enforce uniqueness. If you do decide to create a unique index, you can tell MongoDB to drop all the duplicates. This may (or may not) be what you want, so you should think carefully before using this option because you might accidentally delete half your data. By default, an error will be returned if you try to create a unique index on a key that has duplicate values.

There are many occasions where you will want to create an index that allows duplicates. For example, if your application searches by surname, it makes sense to build an index on the surname key. Of course, you cannot guarantee that each surname will be unique; and in any database of a reasonable size, duplicates are practically guaranteed.

MongoDB's indexing abilities don't end there, however. MongoDB can also create indexes on embedded documents. For example, if you store numerous addresses in the address key, you can create an index on the zip or post code. This means that you can easily pull back a document based on any post code—and do so very quickly.

MongoDB takes this a step further by allowing *composite indexes*. In a composite index, two or more keys are used to build a given index. For example, you might build an index that combines both the `surname` and `forename` tags. A search for a full name would be very quick because MongoDB can quickly isolate the surname and then, just as quickly, isolate the forename.

We will look at indexing in more depth in Part III of this book, but suffice it to say that MongoDB has you covered as far as indexing is concerned.

Leveraging Geospatial Indexes

One form of indexing worthy of special mention is *geospatial indexing*. This new, specialized indexing technique was introduced in MongoDB 1.4. You use this feature to index location-based data, enabling you to answer queries such as how many items are within a certain distance from a given set of coordinates.

As an increasing number of web applications start making use of location-based data, this feature will play an increasingly prominent role in everyday development. For now, though, geospatial indexing remains a somewhat niche feature; nevertheless, you will be very glad it's there if you ever find that you need it.

Profiling Queries

MongoDB comes with a profiling tool that lets you see how MongoDB works out which documents to return. This is useful because, in many cases, a query can be easily improved simply by adding an index. If you have a complicated query, and you're not really sure why it's running so slowly, then the query profiler can provide you with extremely valuable information. Again, you'll learn more about the profiler later in Part III.

Updating Information In-Place

When a database updates a row (or in the case of MongoDB, a document), it has a couple of choices about how to do it. Many databases choose the multi-version concurrency control (MVCC) approach, which allows multiple users to see different versions of the data. This approach is useful because it ensures that the data won't be changed part way through by another program during a given transaction.

The downside to this approach is that the database needs to track multiple copies of the data. For example, CouchDB provides very strong versioning, but this comes at the cost of writing the data out in its entirety. While this ensures that the data is stored in a robust fashion, it also increases complexity and reduces performance.

MongoDB, on the other hand, updates information *in-place*. This means that (in contrast to CouchDB) MongoDB can update the data wherever it happens to be. This typically means that no extra space needs to be allocated, and the indexes can be left untouched.

Another benefit of this method is that MongoDB performs *lazy writes*. Writing to and from memory is very fast, but writing to disk is thousands of times slower. This means that you want to limit reading and writing from the disk as much as possible. This isn't possible in CouchDB because that program ensures each document is quickly written to disk. While this guarantees that the data is written safely to disk, this also impacts performance significantly.

MongoDB only writes to disk when it has to, which is usually once every second or so. This means that if a value is being updated many times a second—a not uncommon scenario if you're using a value as a page counter or for live statistics—then the value will only be written once, rather than the thousands of times that CouchDB would require.

This approach makes MongoDB much faster, but, again, it comes with a tradeoff. CouchDB may be slower, but it does guarantee that data is stored safely on the disk. MongoDB makes no such guarantee, and this is why a traditional RDBMS is probably a better solution for managing critical data such as billing or accounts receivable.

Storing Binary Data

GridFS is MongoDB's solution to storing binary data in the database. BSON supports saving up to 4MB of binary data in a document, and this could well be enough for your needs. For example, if you want to store a profile picture or a sound clip, then 4MB might be more space than you need. On the other hand, if you want to store movie clips, high-quality audio clips, or even files that are several hundred megabytes in size, then MongoDB has you covered here, too.

GridFS works by storing the information about the file (called *metadata*) in the `files` collection. The data itself is broken down into pieces called *chunks* that are stored in the `chunks` collection. This approach makes storing data both easy and scalable; it also makes range operations (such as retrieving specific parts of a file) much easier to use.

Generally speaking, you would use GridFS through your programming language's MongoDB driver, so it's unlikely you'd ever have to get your hands dirty at such a low level. As with everything else in MongoDB, GridFS is designed for both speed and scalability. This means you can be confident that MongoDB will be up to the task if you want to work with large data files.

Replicating Data

When we talked about the guiding principles behind MongoDB, we mentioned that RDBMS databases offer certain guarantees for data storage that are not available in MongoDB. These guarantees weren't implemented for a handful of reasons. First, these features would slow the database down. Second, they would greatly increase the complexity of the program. Third, it was felt that the most common failure on a server would be hardware, which would render the data unusable anyway, even if the data were safely saved to disk.

Of course, none of this means that data safety isn't important. MongoDB wouldn't be of much use if you couldn't count on being able to access the data when you need it. MongoDB provides a safety net using a feature called master-slave replication. This means that only one database is active for writing at any given time, an approach that is also fairly common in the RDBMS world.

The theory behind this approach goes something like this: by passing all writes to the first database (the *master database*) to a replica (the *slave database*) of the master database, you have nothing to worry about if the master database fails (for either hardware or software reasons) because the slave database can carry on in its place.

■ **Caution** It is possible that some of the data written by the master database will not yet have made it to the slave database at the point a failure occurs.

One powerful feature in MongoDB is the concept of replica pairs. This feature is similar to the master-slave setup, with one exception: the two servers automatically decide which server is the master and which is the slave. If a server fails, the two servers will automatically sort out how to proceed when the failed server comes back online.

Implementing Auto Sharding

For those involved with large-scale deployments, the auto sharding feature will probably prove one of MongoDB's most significant and oft-used features. Although many people will be perfectly happy with a single server or perhaps a replica pair, sharding enables you to implement much more scalable deployments.

There are a couple different types of sharding: auto and manual. *Manual sharding* is already possible to a certain extent. In that scenario, you set up two MongoDB master servers and store half your data on one and the rest of your data on the other. With manual sharding, you are responsible for keeping track of what data is on which server, as well as for running the queries that pull the data back together. This is doable, but it can get very complex, and you lose one of MongoDB's best features: its simplicity.

In the *auto sharding* scenario, MongoDB takes care of all the data splitting and recombination for you. It makes sure the data goes to the right server and that queries are run and combined in the most efficient manner possible. In fact, from a developer's point of view, there is no difference between talking to a MongoDB database with a hundred shards and talking to a single MongoDB server. This feature is not yet production-ready; when it is, however, it will push MongoDB's scalability through the roof.

In the meantime, if you're just starting out or you're building your first MongoDB-based website, then you'll probably find that a single instance of MongoDB is sufficient for your needs. If you end up building the next Facebook or Amazon, however, you will be glad that you built your site on a technology that can scale so limitlessly.

Using Map and Reduce Functions

For many people, hearing the phrase *map/reduce* sends shivers down their spines. At the other extreme, many RDBMS advocates scoff at the complexity of `map` and `reduce` functions. It's scary for some because these functions require a completely different way of thinking about finding and sorting your data, and many professional programmers have trouble getting their heads around the concepts that underpin `map` and `reduce` functions. That said, these functions provide an extremely powerful way to query data. In fact, CouchDB supports only this approach, which is one reason CouchDB has such a high learning curve.

MongoDB doesn't require that you use `map` and `reduce` functions. In fact, MongoDB relies on a simple querying syntax that is more akin to what you see in MySQL. However, MongoDB does make these functions available for those who want them. These functions are written in JavaScript and run on the server. The job of the `map` function is to find all the documents that meet a certain criteria. These results are then passed to the `reduce` function, which processes the data. The `reduce` function doesn't usually return a collection of documents; rather, it returns a new document that contains the information derived. As a general rule, if you would normally use GROUP BY in SQL, then the `map` and `reduce` functions are probably the right tools for the job in MongoDB.

We won't go into too much depth on the topic of map/reduce here. While these functions are very powerful, you don't need them to get up and running or to accomplish most day-to-day tasks with MongoDB.

■ **Note** You should not think of MongoDB's `map` and `reduce` functions as poor imitations of the approach adopted by CouchDB. If you so desired, you could use MongoDB's `map` and `reduce` functions for everything in lieu of MongoDB's innate query support.

Getting Help

MongoDB has a great community, and the core developers are very active, easily approachable, and typically go to great lengths to help other members of the community. MongoDB is easy to use and comes with great documentation; however, it's still nice to know that you're not alone, and help is available, should you need it.

Visiting the Website

The first place to look for updated information or help is on the MongoDB website (`http://mongodb.org`). This site is updated regularly and contains all the latest MongoDB goodness. On this site, you can find drivers, tutorials, examples, frequently asked questions, and much more.

Chatting with the MongoDB Developers

The MongoDB developers hang out on Internet Relay Chat (IRC) at `#MongoDB` on the Freenode network (`www.freenode.net`). MongoDB's developers are based in New York, but they are often found chatting in this channel well into the night. Of course, the developers do need to sleep at some point (coffee only works for so long!); fortunately, there are also many knowledgeable MongoDB users from around the world who are ready to help out. Many people who visit the `#MongoDB` channel aren't experts; however, the general atmosphere is so friendly that they stick around anyway. Please do feel free to join `#MongoDB` channel and chat to people there—you may find some great hints and tips. If you're really stuck, you'll probably be able to quickly get back on track.

Cutting and Pasting MongoDB Code

Pastie (`http://pastie.org`) is not strictly a MongoDB site; however, it is something you will come across if you float about in `#MongoDB` for any length of time. Pastie is a site that basically lets you cut and paste (hence the name) some output or program code, and then put it online for others to view. In IRC, pasting multiple lines of text can be messy or hard to read. If you need to post a fair bit of text (such as three lines or more), then you should visit `http://pastie.org`, paste in your content, and then paste the link to your new page into the channel.

Finding Solutions on Google Groups

MongoDB also has a Google group called `mongodb-user` (`http://groups.google.com/group/mongodb-user`). This group is a great place to ask questions or search for answers. You can also interact with the group via e-mail. Unlike IRC, which is very transient, the Google group is a great long-term resource. If you really want to get involved with the MongoDB community, joining the group is a great way to start.

Leveraging the JIRA Tracking System

MongoDB uses the JIRA issue tracking system. You can view this site at `http://jira.mongodb.org/`, and you are actively encouraged to report any bugs or problems that you come across to this site. Reporting such issues is viewed by the community as a genuinely good thing to do. Of course, you can also search through previous issues, and you can even view the roadmap and planned updates for the next release.

If you haven't posted to JIRA before, you might want to visit the IRC room first. You will quickly find out whether you've found something new, and, if so, you will be shown how to go about reporting it.

Summary

This chapter has provided a whistle-stop tour of the benefits MongoDB brings to the table. We've looked at the philosophies and guiding principles behind MongoDB's creation and development, as well as the tradeoffs MongoDB's developers made when implementing these ideals. We've also looked at some of the key terms used in conjunction with MongoDB, how they fit together, and their rough SQL equivalents.

Next, we looked at some of the features MongoDB offers, including how and where you might want to use them. Finally, we wrapped up the chapter with a quick overview of the community and where you can go to get help, should you need it.