THE EXPERT'S VOICE® IN OPEN SOURCE

The Definitive Guide to MongoDB

A complete guide to dealing with Big Data using MongoDB

—

Third Edition

—

David Hows
Peter Membrey
Eelco Plugge
Tim Hawkins

Apress®

The Definitive Guide to MongoDB

A complete guide to dealing with Big Data using MongoDB

Third Edition

David Hows
Peter Membrey
Eelco Plugge
Tim Hawkins

Apress®

The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB

ISBN-13 (pbk): 978-1-4842-1183-0

ISBN-13 (electronic): 978-1-4842-1182-3

Managing Director: Welmoed Spahr
Lead Editor: Michelle Lowman
Technical Reviewer: Stephen Steneker
Editorial Board: Steve Anglin, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Mary Bearden
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com/9781484211830. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

For Dr. Rocky Chan, for going the extra mile and always being there when I need him. I hope one day I can properly thank him for his support.

—Peter Membrey

To my uncle, Luut, who introduced me to the vast and ever-challenging world of IT. Thank you.

—Eelco Plugge

Contents at a Glance

Contents

About the Authors

David Hows is an Honors graduate from the University of Woolongong in NSW, Australia. He got his start in computing trying to drive more performance out of his family PC without spending a fortune. This led to a career in IT, where David has worked as a Systems Administrator, Performance Engineer, Software Developer, Solutions Architect, and Database Engineer. David has tried in vain for many years to play soccer well, and his coffee mug reads "Grumble Bum."

Peter Membrey is a Chartered IT Fellow with over 15 years of experience using Linux and Open Source solutions to solve problems in the real world. An RHCE since the age of 17, he has also had the honor of working for Red Hat and writing several books covering Open Source solutions. He holds a master's degree in IT (Information Security) from the University of Liverpool and is currently an EngD candidate at the Hong Kong Polytechnic University, where his research interests include time synchronization, cloud computing, big data, and security. He lives in Hong Kong with his wonderful wife Sarah and son Kaydyn.

Eelco Plugge is a techie who works and lives in the Netherlands. Currently working as an engineer in the mobile device management-industry where he spends most of his time analyzing logs, configs and errors, he previously worked as a data encryption specialist at McAfee and held a handful of IT/system engineering jobs. Eelco is the author of various books on MongoDB and Load Balancing, a skilled troubleshooter and holds a casual interest in IT security-related subjects complementing his MSc in IT Security.

Eelco is a father of two, and any leisure time left is spent behind the screen or sporadically reading a book. Interested in science and nature's oddities, currency trading (FX), programming, security and sushi.

Tim Hawkins produced one of the world's first online classifieds portals in 1993, loot.com, before moving on to run engineering for many of Yahoo EU's non-media-based properties, such as search, local search, mail, messenger, and its social networking products. He is currently managing a large offshore team for a major US eTailer, developing and deploying next-gen eCommerce applications. Loves hats, hates complexity.

About the Technical Reviewer

Stephen Steneker (aka Stennie) is an experienced full stack software developer, consultant, and instructor. Stephen has a long history working for Australian technology startups including founding technical roles at Yahoo! Australia & NZ, HomeScreen Entertainment, and Grox. He holds a BSc (Computer Science) from the University of British Columbia.

In his current role as a Technical Services Engineer for MongoDB, Inc., Stephen provides support, consulting, and training for MongoDB. He frequently speaks at user groups and conferences, and is the founder and wrangler for the Sydney MongoDB User Group (`http://www.meetup.com/SydneyMUG/`).

You can find him on Twitter, StackOverflow, or Github as `@stennie`.

About the Contributor

A. Jesse Jiryu Davis is a Staff Engineer at MongoDB in New York City, specializing in C, Python, and asynchronous I/O. He is the lead developer of the MongoDB C Driver, author of Motor, and a contributor to Python, PyMongo, and Tornado. He is the co-author with Guido van Rossum of the chapter "A Web Crawler With asyncio Coroutines" in 500 Lines or Less, the fourth book in the Architecture of Open Source Applications series.

Acknowledgments

My thanks to all members of the MongoDB team, past and present. Without them we would not be here, and the way people think about the storage of data would be radically different. I would like to pay extra special thanks to my colleagues at the MongoDB team in Sydney, as without them I would not be here today.

—David Hows

Writing a book is always a team effort. Even when there is just a single author, there are many people working behind the scenes to pull everything together. With that in mind I want to thank everyone in the MongoDB community and everyone at Apress for all their hard work, patience, and support. Thanks go to Dave and Eelco for really driving the Third Edition home.

I'd also like to thank Dou Yi, a PhD student also at the Hong Kong Polytechnic University (who is focusing on security and cryptographic based research), for helping to keep me sane and (patiently) explaining mathematical concepts that I really should have grasped a long time ago. She has saved me hours of banging my head against a very robust brick wall.

Special thanks go to Dr. Rocky Chang for agreeing to supervise my EngD studies and for introducing me to the world of Internet Measurement (which includes time synchronization). His continued support, patience and understanding are greatly appreciated.

—Peter Membrey

To the 9gag community, without whom this book would have been finished months ago.

—Eelco Plugge

I would like to acknowledge the members of the mongodb-user and mongodb-dev mail lists for putting up with my endless questions.

—Tim Hawkins

Introduction

I am a relative latecomer to the world of databases, starting with MySQL in 2006. This followed the logical course for any computer science undergraduate, leading me to develop on a full LAMP stack backed by rudimentary tables. At the time I thought little about the complexities of what went into SQL table management. However, as time has gone on, I have seen the need to store more and more heterogeneous data and how a simple schema can grow and morph over time as life takes its toll on systems.

My first introduction to MongoDB was in 2011, when Peter Membrey suggested that instead of a 0 context table of 30 key and 30 value rows, I simply use a MongoDB instance to store data. And like all developers faced with a new technology I scoffed and did what I had originally planned. It wasn't until I was halfway through writing the code to use my horrible monstrosity that Peter insisted I try MongoDB, and I haven't looked back since. Like all newcomers from SQL-land, I was awed by the ability of this system to simply accept whatever data I threw at it and then return it based on whatever criteria I asked. I am still hooked.

Our Approach

And now, in this book, Peter, Eelco Plugge, Tim Hawkins, and I have the goal of presenting you with the same experiences we had in learning the product: teaching you how you can put MongoDB to use for yourself, while keeping things simple and clear. Each chapter presents an individual sample database, so you can read the book in a modular or linear fashion; it's entirely your choice. This means you can skip a certain chapter if you like, without breaking your example databases.

Throughout the book, you will find example commands followed by their output. Both appear in a fixed-width "code" font, with the commands also in boldface to distinguish them from the resulting output. In most chapters, you will also come across tips, warnings, and notes that contain useful, and sometimes vital, information.

—David Hows

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 set up. 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 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 a very different database than you're probably 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 the following pages, you will learn about MongoDB's background and guiding principles and why the MongoDB team made the design decisions it did. We'll also take a whistle-stop tour of MongoDB's feature list, providing just enough detail to ensure that you'll be completely hooked on this topic for the rest of the book.

We'll start 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-oriented databases, how they fit together, and what their strengths and weaknesses are. We'll also explore JavaScript Object Notation 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 relational (SQL) databases (MongoDB is a document-oriented database) have been used for storing content of all types. It didn't matter whether the data were 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 were 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 2nd edition,* by Jason Lengstorf and Thomas Blom Hansen (Apress, 2014), you'll probably discover almost right away that 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 and had to set up several tables and then try to pull them all together knows what we're 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 and that 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 the one used for the *New York Times* website

Once you're comfortable with the idea that MongoDB may not solve all your problems, you will discover that there are certain problems that MongoDB is a perfect fit for resolving, such as analytics (think a real-time Google Analytics for your website) and complex data structures (for example, 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 is that 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 purposes (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 choose to use it, MongoDB provides a rich document-oriented 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, Mac OS, Windows, and Solaris.

MongoDB succeeds at all these goals, and this is why using MongoDB (at least for us) 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 them to MongoDB for handling.

Consider this real-world example. A recent application that co-author Peter Membrey worked on needed to store a set of eBay search results. There could be any number of results (up to 100 of them), and he needed an easy way to associate the results with the users in his database. Had Peter been using MySQL, he would have had to design a table to store the data, write the code to store his 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, he was using MongoDB, so things went a bit differently.

Specifically, he added this line of code:

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

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

Lacking Innate Support for Transactions

Here's another important design decision by MongoDB developers: The database does not include transactional semantics (the element 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 it so far. With horizontal scaling, 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 because of 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`—which also happens to be the default) doesn't support transactions either. This fact hasn't stopped MySQL from becoming and remaining the dominant open source database for well over a decade. As with most choices when developing solutions, using MongoDB is going to be a matter of personal preference and whether the tradeoffs fit your project.

■ **Note** MongoDB offers durability when used in tandem with at least two data-bearing servers as part of a three-node cluster. This is the recommended minimum for production deployments. MongoDB also supports the concept of "write concerns." This is where a given number of nodes can be made to confirm the write was successful, giving a stronger guarantee that the data are safely stored.

Single server durability is ensured since version 1.8 of MongoDB with a transaction log. This log is append only and is flushed to disk every 100 milliseconds.

JSON and MongoDB

JSON (JavaScript Object Notation) 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* (that is, it doesn't require a schema), 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 won't change how you 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 a number of extended types for numeric data (such as int32 and int64) and support for handling binary data. We'll look at BSON in more depth in "Using Document-Oriented Storage (BSON)," later in this chapter.

The original specification for JSON can be found in RFC 7159, 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 were exchanged using the comma-separated values x(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
```

Someone 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 avoid this ambiguity, CSV files often have a header field, in which the first row defines what comes in the file. The following snippet takes the previous example one step further:

```
Lastname, Firstname, Phone Number
Membrey, Peter, +852 1234 5678
Thielen, Wouter, +81 1234 5678
```

Okay, that's a bit better. But now assume some people in the CSV file 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 are flat and don'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 sales data in the form of 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 the 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 is that XML is not only designed for machines to read (whereas JSON is designed for humans), but it is also 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, it 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 the content easy to understand and simple to use. Let's revisit the previous example; however, this time we used JSON rather than CSV:

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

In this version of the example, each JSON object (or document) contains all the information needed to understand it. If you look at `phone_numbers`, you can see that it contains 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:

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

This version of the 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 from scratch, it is usually generated automatically in software. For example, Python includes a module called (somewhat predictably) `json` 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. 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 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 fit together. We'll look at this in more depth in Chapter 3.

Adopting a Nonrelational 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, PostgreSQL can't run a single database on two servers, where both servers can both read and write data (often referred to as an *active/active cluster*), and MySQL can only do it with a special add-on package. 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 them 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 are 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 are 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 is if someone queries the second server for information 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 the active/active cluster problems in a very clever way—it avoids them completely. Recall that MongoDB stores data in BSON documents, so the data are self-contained. That is, although similar documents are stored together, individual documents aren't made up of relationships. This means that 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.

Admittedly, MongoDB does not offer *master/master replication*, in which two separate servers can both accept write requests. However, it does have *sharding*, which allows data to be partitioned across multiple machines, with each machine responsible for updating different parts of the dataset. The benefit of a sharded cluster is that additional shards can be added to increase resource capacity in your deployment without any changes to your application code. Nonsharded database deployments are limited to vertical scaling: you can add more RAM/CPU/disk, but this can quickly get expensive. Sharded deployments can also be scaled vertically, but more importantly, they can be scaled horizontally based on capacity requirements: a sharded cluster can be comprised of many more affordable commodity servers rather than a few very expensive ones. Horizontal scaling is a great fit for elastic provisioning with cloud-hosted instances and containers.

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 are 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 a means 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 outdated 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 set of features, such as the ability to search for specific documents. It must also be able to store those documents on disk, so they can survive a reboot. Fortunately, MongoDB provides enough features 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 or run the application practically anywhere. Currently, binaries can be downloaded from the MongoDB website for Linux, Mac OS, Windows, and Solaris. Officially supported Linux packages include Amazon Linux, RHEL, Ubuntu Server LTS, and SUSE. You can even download the source code and build your own MongoDB, although it is recommended that you use the provided binaries wherever possible.

■ **Caution** The 32-bit version of MongoDB is limited to databases of 2GB or less. This is because MongoDB uses memory-mapped files internally to achieve high performance. Anything larger than 2GB on a 32-bit system would require some fancy footwork that wouldn't be 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 or virtual machines, and 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.