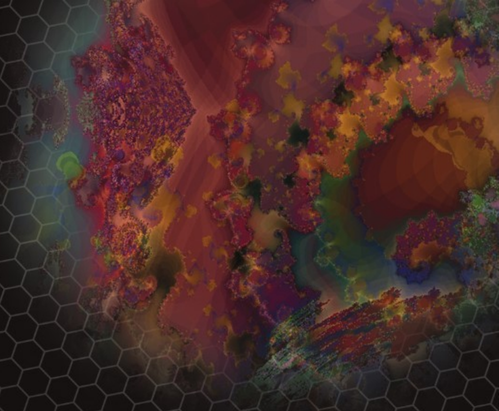
THE EXPERT'S VOICE® IN OPEN SOURCE



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®

The Definitive Guide to MongoDB

The NoSQL Database for Cloud and Desktop Computing

Eelco Plugge,
Peter Membrey
and Tim Hawkins

Apress®

The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing

ISBN-13 (pbk): 978-1-4302-3051-9

ISBN-13 (electronic): 978-1-4302-3052-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

President and Publisher: Paul Manning
Lead Editors: Frank Pohlmann, Michelle Lowman, James Markham
Technical Reviewer: Jonathon Drewett
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editor: Mary Tobin
Copy Editor: Patrick Meader
Compositor: MacPS, LLC
Indexer: Potomac Indexing, LLC
Artist: April Milne
Cover Designer: Anna Ishchenko

The source code for this book is available to readers at `www.apress.com`. You will need to answer questions pertaining to this book in order to successfully download the code.

For the love of my life, Marjolein, and my son Jesse—I wouldn't have been able to write this without your everlasting patience and love.

—Eelco Plugge

For my mother-in-law, Wan Ha Loi. First for actually letting me marry her wonderful daughter and second for coming out of retirement to look after our son Kaydyn. Her selfless generosity made this book possible, as, without her continuous support, there simply wouldn't be enough hours in the day.

—Peter Membrey

For Ester, for putting up with the long hours I stole from her to produce this book.

—Tim Hawkins

Contents at a Glance

Contents

About the Authors

■ **Eelco Plugge** was born in 1986 in the Netherlands and quickly developed an interest in computers and everything evolving around it. He enjoyed his study at the ICT Academie in Amersfoort, after which he became a data encryption specialist working at McAfee at the age of 21. He's a young BCS Professional Member and shows a great interest in everything IT security-related as well as in all aspects of the Japanese language and culture. He is currently working upon expanding his field of expertise through study, at the same time as maintaining a young family.

■ **Peter Membrey** lives in Hong Kong and is actively promoting Open Source in all its various forms and guises, especially in education. He has had the honor of working for Red Hat and received his first RHCE at the tender age of 17. He is now a Chartered IT Professional and one of the world's first professionally registered ICT Technicians. He has recently completed his master's degree and will soon start a PhD program at the Hong Kong Polytechnic University. He lives with his wife Sarah and his son Kaydyn, and is desperately trying (and sadly failing) to come to grips with Mandarin and Cantonese.

■ **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

■ **Jonathon Drewett** is an ICT specialist experienced in applying technology within the education sector. He operates his own consultancy and has worked on developing large international e-learning data repositories, as well as managing networks and information systems for educational establishments. Before moving into IT, he worked as an electronic engineer and was contracted to the RAF.

Jonathon graduated with an honors degree in Computer Science and is a member of both the British Computer Society and the Institute of Engineering and Technology. He is an ardent advocate of life-long learning and using technology to improve the world.

In his downtime, he restores classic cars, operates a large on-line social community network and, occasionally, sleeps.

Acknowledgments

I would like to sincerely thank Peter for giving me the opportunity to work on this book. His constant motivation kept me going and made it possible to enjoy writing every single page I worked on. I would also like to express my gratitude towards all the people at Apress for all the work they have done to get this book out. It goes without saying that this book wouldn't be here without all of you. Finally, I would like to thank Tim and Jon for jumping in at a crucial moment and helping out; the publication of this book would not have been possible without your help.

Eelco Plugge

First, I'd like to give special thanks to Eelco Plugge for consistently and constantly going above and beyond the call of duty. He has put an astonishing amount of time and energy into this book and it simply would not have been this good without him. I'd also like to thank Tim Hawkins who brought a tonne of hard-won real-world experience and expertise to the book. He joined the team part way through the project and worked incredibly hard (and fast) not only to write his chapters but also to overhaul them when new features and updates for MongoDB were made available. Both Eelco and Tim were the driving forces for the book and I remain especially grateful for all of their hard work.

Next, I'd like to thank Jon Drewett who provided the vast majority of technical review for the book. Not only did he provide great insights (requiring a not insubstantial amount of work on behalf of us authors), he also contributed greatly to ensuring that the book was both technically accurate and as useful and reader friendly as possible.

Of course, without the support of my dear wife Sarah (who grows wiser and more beautiful every day) and my son Kaydyn (who miraculously knew just how to disrupt the writing process for maximum effect), I would not have been able to start work on the book, much less see it completed.

I'd also like to thank all the guys (and gals) at Apress who as usual showed the patience of saints. Special thanks to Mary Tobin who was tasked with managing us—which is somewhat akin to trying to herd cats.

John Hornbeck and Wouter Thielen both deserve a special mention for helping create the table of contents and the structure for the book. Although unfortunately they weren't able to take part in the actual writing, their effort shaped the way for the rest of us.

Last but certainly not least, special thanks to 10gen for sponsoring the Beijing MongoDB workshop—a great time was had by all.

Peter Membrey

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

Tim Hawkins

A Special “Thanks” to *MongoDB Beijing*

On May the 28th 2010, the first ever official MongoDB event was held in Beijing, China. At Thoughtworks, a group of like-minded people got together to discuss MongoDB and how it could solve the problems that the group were facing. Mars Cheng, who organized the event, arranged for the venue, while 10gen paid for travel and accommodation for Peter Membrey. Apress gave away free copies of the e-book to attendees and this made up a large proportion of the lab work for the session. Special thanks then to Mars, 10gen and Apress who not only put together the first ever MongoDB experience in China but also the first ever collaboratively technical reviewed books!

A presentation was given by Peter to talk about some of the high points of MongoDB and how it had made a difference to him personally. A big part of the presentation looked at how he used MongoDB to save hours of work when developing a project for his master’s degree at the University of Liverpool. The presentation also explored the key benefits that MongoDB could offer and the areas where it really shined in comparison to traditional RDBMS such as MySQL.

After the presentation, everyone was invited to go to the Apress website where they could obtain an Alpha version of the e-book. The Alpha version is a collection of chapters written by the authors that haven’t yet been through the full editorial process. In other words they can be pretty raw, with typing mistakes and other minor errors. By giving away free Alpha books, Apress was in effect offering a group of people who were very interested in MongoDB the chance to look at what we had so far and to offer suggestions for improvement.

The labs went extremely well with everyone getting involved and offering ideas and insights, many of which were incorporated into the book itself. As a special thank you to the team, we would like to acknowledge those who took part. In no particular order (as provided by Mars):

Mars Cheng	Runchao Li
Blade Wang	Guozhu Wen
Sarah Membrey	Qiu Huang
Yao Wang	Shixin He
Zhen Chen	Chaoqun Fu
Jian Han	Lin Huang
Fan Pan	

All in all, everyone had a great day and the presentation and labs were considered to be a big success. It is very likely that this will be the first of many MongoDB activities in China and that there will be a growing demand for related skills in the job market. More details of the event can be found on the MongoDB website at `http://www.mongodb.org/display/community/MongoDB+Beijing+Meetup+2010`.

Introduction

The seed for *The Definitive Guide to MongoDB* was actually planted some years ago when I walked into a local bookstore, and first spotted a book on databases. I started reading the back-cover copy and a few pages of the front matter, but quickly found the book closed in my hands, as I quietly mumbled to myself: "Humph. Who needs databases, other than a very large enterprise?" I put the book back, and headed home without thinking any more about it.

Nearly two years later, I was toying with the idea of setting up a simple website in plain HTML code, and, while searching for some "funky" ideas that I could use with my limited space and options, I came across the term "databases" again and again. As I was no longer able to ignore the existence of databases, I began to pay more attention to them. But I still wasn't convinced they were my thing, partly because of all the puzzling expressions that were being used, such as "entity-relation models" and "cardinality," and even the more common words, such as "keys," baffled me. That would soon change.

While enrolled at the ICT Academie in the Netherlands for my first proper education in the IT world, I was confronted with databases yet again. This time, I was required to take an actual exam on them, and, knowing just the basic concepts of databases (how they worked, and how to create, manage and delete them), I did what many beginners would do: I panicked.

This was the moment, however, where I finally decided to pull my head out of the sand and learn all I could about databases. Surprisingly, I quickly grew fond of them, and started to use one "just for the fun of it" with my now more sophisticated PHP/MySQL-driven website. I wasn't quite there yet, though. Then came MongoDB…

In early 2010, I was introduced to MongoDB by my close friend and co-author Peter Membrey. I was immediately hooked and intrigued by its concepts, simplicity, and strengths. I found myself reading each section of the MongoDB website over and over again, readily absorbing its capabilities and advantages over the traditional RDBMS applications. I finally felt comfortable with databases.

Our Approach

And now, in this book, our goal is to present 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 that example commands are written in **bold** styled code 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.

We trust you will find this book easy to grasp and pleasant to read, and, with that said, we hope you enjoy *The Definitive Guide to MongoDB*.

Eelco Plugge

PART I

Basics

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.