MongoDB IN ACTION

Kyle Banker

MEAP

MANNING

MEAP Edition
Manning Early Access Program
MongoDB in Action version 9

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

Table of Contents

1 A Document Database for the Modern Web

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

MongoDB is a database management system designed from the start for web applications and Internet infrastructure. The data model and persistence strategies are built for high read- and write-throughput and the ability to scale easily with automatic failover. Whether an application requires just one database node or dozens of them, MongoDB can provide surprisingly good performance. If you've experienced difficulties scaling relational databases, this may be great news. But not everyone needs to operate at scale. Maybe all you've ever needed for your applications is a single database server. Why then would you use MongoDB in this simpler case?

It turns out that MongoDB is often immediately attractive, not because of its scaling strategy, but rather because its intuitive data model. Given that a

document-based data model can represent rich, hierarchical data structures, it's commonly possible to do without the complicated multi-table joins imposed upon us by relational databases. For example, suppose we're modeling products for an e-commerce site. With a fully-normalized relational data model, the information for any one product will be divided among perhaps dozens of tables. If we want to get a product representation from the database shell, we'll have to write a complicated SQL query full of joins. As a consequence, most developers will have to rely on a secondary piece of software to assemble the data into something meaningful.

With a document model, by contrast, most of a product's information can be represented within a single document. When we open the MongoDB JavaScript shell, we can easily get a comprehensible representation of our product, with all it's information hierarchically organized in a JSON-like structure. We can also query for it and manipulate it. In fact, MongoDB's query capabilities are designed specifically for manipulating structured documents, so users switching from relational databases experience a similar level of query power.

If the distinction just made is brand-new to you, then you probably have a lot of lingering questions. Rest assured that by the end of this chapter we'll have provided a thorough overview of MongoDB's features and design goals, making it increasingly clear why developers from companies like SourgeForge to The New York Times have adopted MongoDB for their projects. We'll start with a history of MongoDB and lead into a tour of the database's main features. Next, we'll explore some of the other alternative database solutions and the so-called NoSQL movement[1], explaining how MongoDB fits in. Finally, we'll describe in general where MongoDB works best and where an alternative data store might be preferable.

Footnote 1 The umbrella term NoSQL, as its currently used, was coined in 2009 as a way of lumping together the many non-relational databases gaining in popularity at the time.

1.1 Born in the Cloud

The history of MongoDB is brief but worth recounting, for it was born out of a much more ambitious project. In mid-2007, a startup called 10gen began work on a software platform-as-a-service, comprising an application server and a database, that would host web applications and scale them as needed. Like Google's AppEngine, 10gen's platform was designed to handle the scaling and management of hardware and software infrastructure automatically, freeing developers to focus solely on their application code. 10gen ultimately discovered that most developers didn't feel comfortable giving up so much control over their technology stacks; but happily, users *were* interested in the database server itself. This led 10gen to concentrate its efforts solely on the database that became MongoDB.

With MongoDB's increasing adoption and production deployments large and small, 10gen continues to sponsor the database's development as an open-source project. Despite its being open-source, all of MongoDB's core developers are either founders or employees of 10gen. The project's road map continues to be determined by the needs of its user community and the overarching goal of creating a database that combines the best features of relational databases and distributed key-value stores. 10gen's business model, then, is not unlike that of other well-known open-source companies: support the development of an open-source product and provide support services to end-users.

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

1.2 MongoDB's Key Features

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

1.2.1 The Document Data Model

MongoDB is a document-oriented database. If you're not familiar with documents in the context of databases, the concept can be most easily demonstrated by example.

Listing 1.1 A document representing an entry on a social news site.

```
{ _id: ObjectID('4bd9e8e17cefd644108961bb'),
  title: 'Adventures in Databases',
  url: 'http://example.com/databases.txt',
  author: 'msmith',
  vote_count: 20,

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

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

  comments: [
    {user: 'bjones',
      text: 'Interesting article!'
    },

    {user: 'blogger',
      text 'Another related article is at http://example.com/db/db.txt'
    }
 ]
}
```

❶ **The _id field is the primary key.**

❷

❸ **This attribute points to another document.**

❹ **Comments are stored as an array of comment objects.**

Listing 1.1 shows a sample document representing an article on a social news site (e.g., Digg). Documents are generally represented using dictionary-like objects. Here, we employ JSON, but we just as easily could have used a Python dictionary, a Ruby hash, or another similar language structure. Notice that a document is essentially a set of property names and their values. The values can be simple data types, like strings, numbers, and dates. But these values can also be arrays ❷ and even other documents ❸, and it is these constructs that let documents represent a variety of rich data structures. You'll see that our sample document has a property, tags ❷, that stores the article's tags in an array. But even more interesting is the comment property ❹, which references an array of comment documents.

Let's take a moment to contrast this with a standard relational database representation of the same data. Figure 1.1 shows a likely relational analogue. Tables are essentially flat; this means that representing the various one-to-many relationships in our post is going to require multiple tables. We start with a posts table containing the core information for each post. Then we create three other

tables, each of which includes a field, `post_id`, referencing the original post. The technique of separating an object's data into multiple tables likes this is known as normalization. Normalization is often important because it can assure us that each unit of data is represented in one place only. One problem with duplicated data, or a data model, is that updates become more expensive: we have to alter the data in more than one place. This is inefficient, introduces application complexity, and, in the worst case, may result in an inconsistent data set.

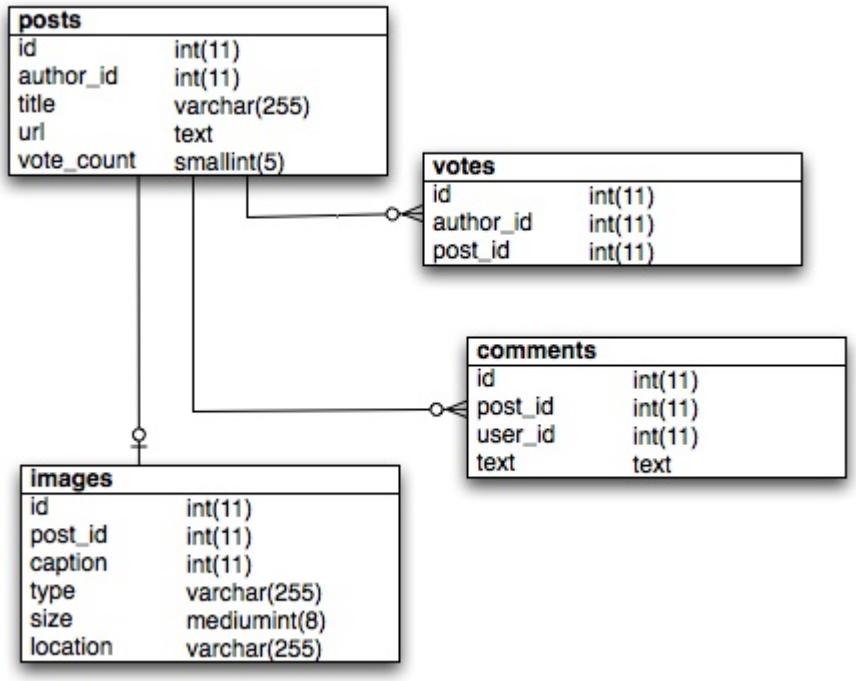

Figure 1.1 A basic relational data model for entries on a social news site.

That said, normalization is not without its costs, the first of which is that some assembly is required. To display the post referenced above, we'll need to perform a join between the post and tags tables. We also have to query separately for the comments or possibly include them in a join as well. Ultimately, the question of whether to normalize depends on the kind of data we're modeling, and we'll have much more to say about the topic in Chapter 4. What's important to notice here is that document-oriented data model handles the denormalized representation naturally, allowing us to work with an object holistically: all the data representing a post, from comments to tags, can be fit into a single database object.

You've probably noticed that in addition to providing a richness of structure, documents need not conform to a prespecified schema. With a relational database,

we store rows in a table. Each table has a strictly-defined schema specifying which columns and types are permitted. If any row in a table needs an extra field, we have to alter the table explicitly. MongoDB, however, groups document into collections, which don't impose any sort of schema. In theory, each document in a collection can have a completely different structure; in practice, a collection's documents will be relatively uniform. For instance, all the documents in the posts collection will have fields for the title, tags, comments, etc.

But this lack of imposed schema confers some advantages. First, our application code, and not the database, enforces the data's structure. This can speed up initial application development when the is changing frequently. Second, and even more significantly, a schemaless model allows us to represent data with truly variable properties. For example, imagine you're building an e-commerce product catalog. There's no way of knowing in advance what attributes a product will have, so the application will have to account for that variability. The traditional way of handling this in a fixed-schema database is with the entity-attribute-value pattern[2], shown in figure 1.2. What you're seeing is one section of the data model for Magento, an open-source e-commerce framework. Notice the set of tables that are all essentially the same, save a single attribute, `value`, that varies only by data type. This structure allows an administrator to define additional product types and their attributes, but the result is significant complexity. Think about firing up the MySQL shell to examine or update a product modeled in this way; the SQL joins required to assemble the product would be enormously complex. Modeled as a document, by contrast, no join is required, and new attributes can be added to a single document dynamically.

Footnote 2 http://en.wikipedia.org/wiki/Entity-attribute-value_model

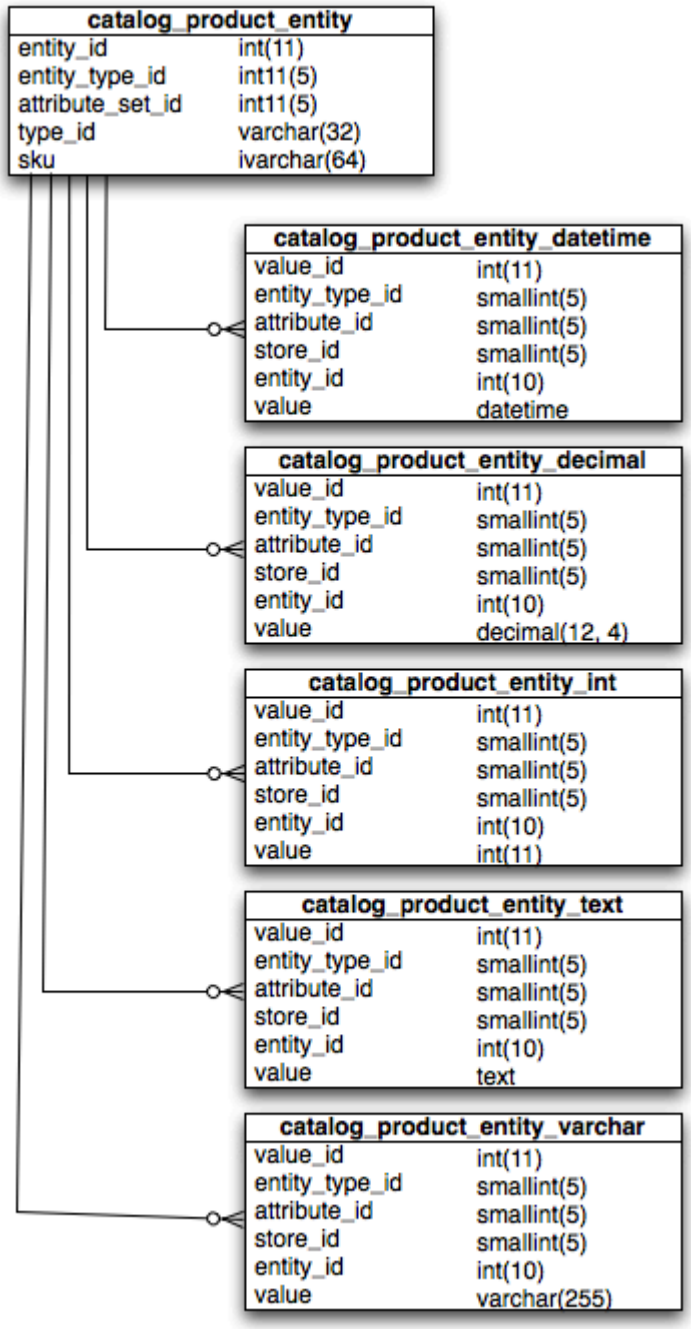

Figure 1.2 A portion of the schema for the PHP e-commerce project Magento. These tables facilitate dynamic attribute creation for products.

1.2.2 Dynamic Queries

To say that a system support dynamic queries is to say that it's not necessary to define in advance what sorts of queries the system will accept. Relational databases have this property; these databases will faithfully execute any well-formed SQL query with any number of conditions. Dynamic queries are easy to take for granted if the only databases you've ever used have been relational. But, in fact, not all databases support dynamic queries. For instance, key-value stores are queryable on one axis only: the value's key. Like many other systems, key-value stores sacrifice rich query power in exchange for a simple scalability model. One of MongoDB's design goals is to preserve most of the dynamic query power that's been so fundamental to the relational database world.

To see how MongoDB's query language works, let's take a simple example involving posts and comments. Suppose we want to find all posts tagged with the term 'politics' having greater than ten votes. A SQL query would look like this:

```
SELECT * FROM posts
  INNER JOIN posts_tags ON posts.id = posts_tags.post_id
  INNER JOIN tags ON posts_tags.tag_id == tags.id
  WHERE tags.text = 'politics' AND posts.vote_count > 10;
```

The equivalent query in MongoDB is specified as a document to match against. The special key indicates the 'greater than' condition.

```
db.posts.find({'tags': 'politics', 'vote_count': {'$gt': 10}});
```

Notice that the two queries assume a different data model. The SQL query relies on a normalized model, where posts and tags are stored in the distinct tables, whereas the MongoDB query assumes that tags are stored within each post document. What both queries demonstrate, however, is an ability to query on arbitrary combinations of attributes, which is the essence of dynamic queryability.

As hinted at earlier, some databases don't support dynamic queries because the data model is too simple. For example, key-value stores can only query their data by primary key. The values pointed to by those keys are entirely opaque as far as queries are concerned. Indeed, the only way to query by a secondary attribute, such as the vote count, would be to write some custom application-level code to manually build out an entry where the primary key indicated a vote count and the value, a list of primary keys. If we took this approach with a key-value store, we'd be guilty of implementing a kind of hack, and although it might work for smaller

data sets, stuffing multiple indexes into what is physically a single index isn't such a good idea. What's more, the hash-based index in a key-value store won't support range queries, which probably be necessary for querying on an item like a vote count.

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

1.2.3 Secondary Indexes

The best way to understand database indexes is by analogy: most books, for example, have indexes mapping keywords to page numbers. Suppose we have a cookbook and want to find all recipes calling for pears (maybe we have a lot of pears at home and don't want them to go bad). The time-consuming approach would be to page through every recipe, checking each ingredient list for pears. Of course, most of us would prefer to check the book's index for the 'pears' entry, which would give us an immediate list of all the recipes containing pears. Database indexes are essentially data structures that provide the same service.

Secondary indexes in MongoDB are implemented as B-trees. This sort of index, also the default for most relational databases, is optimized for a variety of queries, including range scans and queries with sort clauses. Thus, permitting multiple secondary indexes, MongoDB allows users to optimize for a wide variety of queries.

But keep in mind that not all databases use B-trees; nor do they necessarily allow for multiple indexes. Key-value stores typically use a single hash-based index only. Such an index configuration is designed for single-key lookups only.

With MongoDB, you can create up to 64 secondary indexes per collection. The kinds of indexes supported include all the ones you'd find in an RDMBS: ascending, descending, unique, compound-key, and even geospatial index are supported. Because MongoDB and MySQL use the same data structure for their indexes, advice for managing indexes in both these systems is quite compatible. We'll begin looking at indexes in the next chapter, and because an understanding of

indexing is so crucial to efficiently operating a database, we devote all of Chapter 7 to the topic.

1.2.4 Replication

MongoDB supports two kinds of replication: traditional master-slave replication and a more sophisticated variant known as replica sets. In a typical master-slave scenario, we set up two machines, designating one as master and the other as slave. All reads and writes go to the master node, and any updates to master are reflected on the slave. This configuration is usually used for failover; should the master database server crash, our slave database can be manually promoted to master, ensuring minimal downtime and minimal loss of data. [3]

Footnote 3 In some cases, replication can also be used to scale database reads. If we have a read-intensive application, as is commonly the case on the web, then it's possible to direct all database reads to a slave and direct all writes to a single master node. Note that it's never possible to write to slave; if extra write-throughput is needed, a horizontally-scaled architecture is best, as described in section 1.2.6.

Production deployments requiring just a single database node should always have a slave for redundancy. One possible issue with this configuration is that failover requires manual intervention, and sysadmins certainly don't want to have wake up in the middle of the night just to promote a slave to master. That's why, in the interest of automatic failover, MongoDB also supports a replica set architecture.

Replica sets still consist of a master node replicating to one or more *n* slaves, but if the master node fails, the cluster is able to pick a slave and automatically promote it to master. When the former master comes back online, it will do so as a slave. An illustration of this process is provided in figure 1.3.

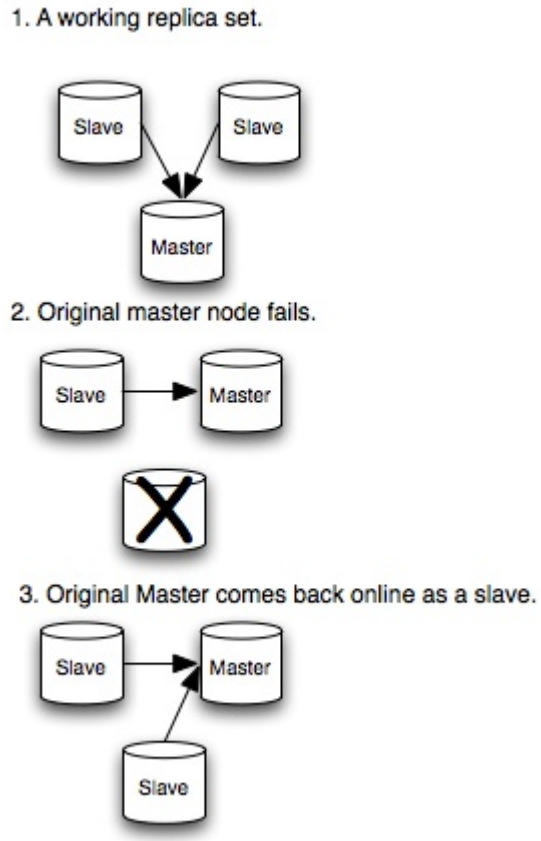

Figure 1.3 Automated failover with a replica set.

Replica sets will be discussed thoroughly in Chapter 8.

1.2.5 Speed and Durability

To understand MongoDB's approach to durability, it pays to consider a few ideas first. In the realm of database systems there exists an inverse relationship between write speed and durability. Write speed, here understood, is volume of inserts, updates, and deletes that a database can process in a given time frame. Durability refers to level of assurance that these write operations have been made permanent.

For instance, suppose we write one-hundred records of 50KB each to a database and then immediately cut the power on the server. Will those records be recoverable when we bring the machine back online? The answer is maybe, and it depends on both our database system and the hardware hosting it. The problem here is that writing to a magnetic hard drive is orders of magnitude slower than writing to RAM. Certain databases, like memcached, write exclusively to RAM, which makes them extremely fast but completely volatile. On the other hand, few

databases write exclusively to disk because the low performance of such an operation is unacceptable. Therefore, database designers often need to make compromises to provide the best balance of speed and durability. [4]

Footnote 4 One such compromise can be seen in MySQL's InnoDB. InnoDB is a transactional storage engine, which by definition must guarantee durability. It accomplishes this by writing its updates in two places, once to a transaction log and again to an in-memory buffer pool. The transaction log is synced to disk immediately while the buffer pool is only eventually synced by a background thread. The reason for this dual write is that, generally speaking, random I/O is much slower that sequential I/O. Since writes to the main data files would constitute random I/O, it's faster to write these changes the RAM first, allowing the sync to disk to happen later. But since some sort of write to disk is necessary to guarantee durability, it's important that that write be sequential; this is exactly what the transaction log provides. Now, in the event of a power outage, InnoDB can replay its transaction log and update the main data files accordingly. (footnote on hardware) This provides an acceptable level of performance while guaranteeing a high level of durability.

MongoDB doesn't support transactions proper, nor does it, by default, guarantee that changes are flushed to disk the moment they're written to the database. This is because MongoDB's design philosophy makes scalability and performance paramount. Transactions reduce performance because distributing them across machines requires complicated locks and introduces network latency. Likewise, ensuring that all writes are synced to disk, even sequentially as with the transaction log described above, is expensive.

Happily, MongoDB provides users a high level of control over the speed/durability tradeoff. By default, all writes are "fire-and-forget," which means that these writes are sent across the socket without waiting for a database response. If users want a response, they can issue a write using their driver's "safe mode," which will, at minimum, wait for a response, ensuring that the write has been received by the server with no errors. MongoDB emphasizes durability through replication, and thus safe mode can also be used to ensure that any write has been replicated to *n* servers before returning. Finally, for the ultimate assurance, users can start MongoDB with journaling enabled. In the event of an unclean shutdown, the journal will be replayed automatically to ensure that the data files are in a consistent state. Clearly, the topics of replication and durability are vast; we'll provide a detailed exploration of them in Chapter 8.

1.2.6 Scaling

The easiest way to scale most databases is to upgrade the hardware. If our application is running on a single node, it's usually possible to add some combination of disk space, memory, and CPU to ease any database bottlenecks. The technique of augmenting a single node's hardware for scale is known as vertical scaling or "scaling up." Vertical scaling has the advantages of being simple, reliable, and cost-effective, but only to a certain point. If you're running on virtualized hardware (e.g., Amazon's EC2), then you may find that a sufficiently large instance simply isn't available. If you're running on physical hardware, there may come a point where the cost of a more powerful server becomes prohibitive.

At this point it may make sense to consider scaling horizontally, or "scaling out." Instead of beefing up a single node, scaling horizontally means distributing the database across multiple machines. Because a horizontally scaled architecture can make use of commodity hardware, costs can be significantly reduced. What's more, the consequences of failure are mitigated by the distribution of machines. Machines will unavoidably fail from time to time. If we've scaled vertically, then we have to deal with the failure of a machine upon which most of our system depends. This may not be an issue if a replica of the data exists on a replicated slave. But it's still the case that only a single server need fail to bring the entire system down. Contrast that with failure within a horizontally-scaled architecture. This is less catastrophic since a single machine represents a much smaller percentage of the system as a whole. And if that machine's data is properly replicated, it's possible to fail over with zero downtime.

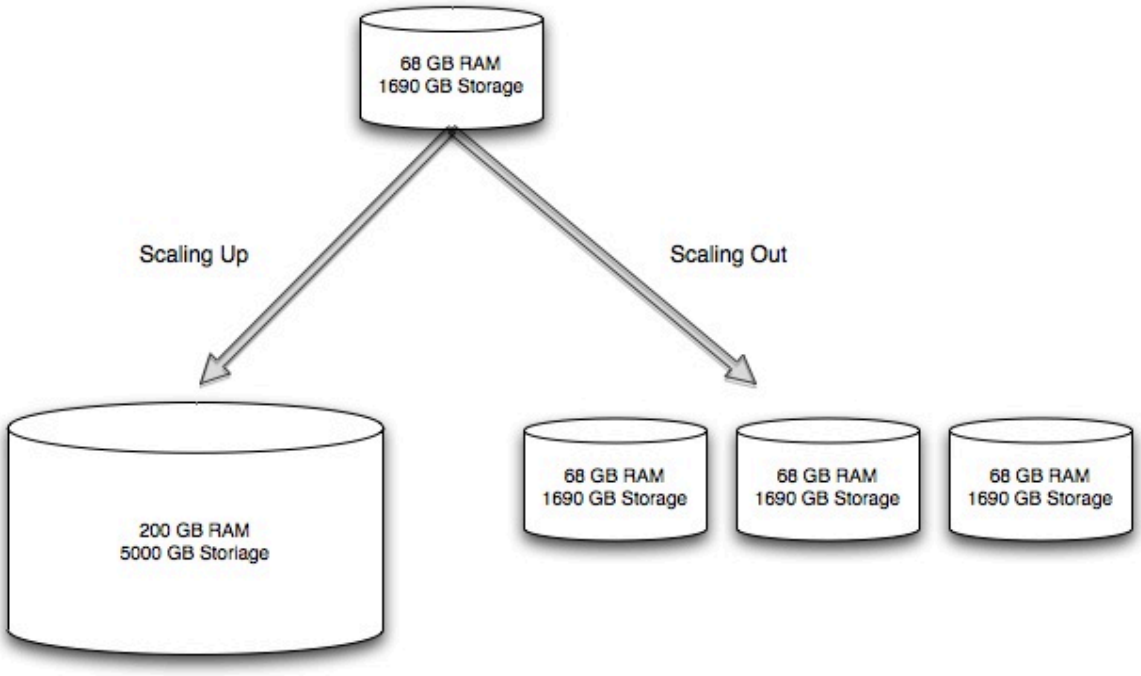

Figure 1.4 Horizontal vs. Vertical Scaling

MongoDB has been designed to make horizontal scaling manageable. It does so via a partitioning mechanism, known as auto-sharding, which automatically manages the distribution of data across nodes. In addition, the sharding system handles the addition of shard nodes, allowing you to add capacity as needed. The architecture also provides for automatic failover; individual shards are made up of a replica set consisting of at least two nodes, ensuring automatic recovery with no single point of failure. All this means that not a single line of application code has to handle these logistics; your application code communicates with a sharded cluster just as it speaks to a single node.

We've covered a lot of MongoDB's most compelling features, and in Chapter 2, we'll being to see how some of these work in practice. But at this point, we're going to take a more pragmatic look at the database. In the next section, we'll look at MongoDB in its environment, the tools that ship with the core server, and few basic ways of getting data in and out.

1.3 MongoDB's Core Server and Tools

MongoDB is written in C++ and is actively developed by 10gen. The project compiles on all major operating systems, including Mac OSX, Windows, and most flavors of Linux; pre-compiled binaries are available for each of these platforms at mongodb.org. MongoDB is open-source and licensed under the GNU-AGPL. The source code is freely available on GitHub, and contributions from the community are frequently accepted. But the project is guided by the 10gen core server team, and the overwhelming majority of commits come from this group.

NOTE **On the GNU-AGPL**

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

MongoDB v1.0 was released in November, 2009. Major releases appear approximately once every three months, with even point numbers for stable branches and odd numbers for development. As of this writing, the latest stable major release is 1.8.[5]

Footnote 5 You should always be using the latest point release, e.g., 1.8.1.

What follows is an overview of the components that ship with MongoDB along with a high-level description of the tools and language drivers for developing applications with the database.

1.3.1 The Core Server

The core database server runs via an executable called `mongod` (`mongodb.exe` on Windows). The `mongod` server process receives commands over a network socket using a special binary protocol. All the data files for a `mongod` process reside on the same machine and are stored by default in `/data/db`.

`mongod` can be run in several modes, the most common of which is as members of a replica set. Since replication is recommended, we generally see replica set configuration consisting of two replicas, plus an arbiter process residing on a third server.[6] For MongoDB's auto-sharding architecture, the components consist of `mongod` processes configured as per-shard replica sets, with special

meta-data servers, known as config servers, on the side. However, there's also a separate routing server called `mongos` used to send requests to the appropriate shard.

Footnote 6 These processes are very light-weight, and thus can easily be run on an app server, for instance.

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

1.3.2 The JavaScript Shell

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

```
use mongodb-in-action
db.users.insert({name: "Eliot"});
```

The first command, indicating which database we want to use, will be familiar to users of relational databases. The second command is a JavaScript expression that inserts a simple document. To see the results of our insert, we can execute a simple query:

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

The find method returns the inserted document, with the required object id added. All documents require a primary key stored in the `_id` field. You're allowed to enter a custom `_id` when creating a document, but if omitted, a

MongoDB object id will be inserted automatically.

In addition handling the sort of data management just described, the shell permits any administrative command to be run. Some examples include viewing the current database operation, checking the status of replication to a slave, and configuring a collection for sharding. Of course, the bulk of your work with MongoDB will be done through an application written in a given programming language; to see how that's done, we must say a few things about MongoDB's language drivers.

1.3.3 Database Drivers

If the notion of a database driver conjures up nightmares of low-level device hacking, don't fret. The MongoDB drivers are quite easy to use. Every effort is made to provide an API that matches the idioms of the given language while also maintaining relatively uniform interfaces across languages. For instance, all of the drivers implement similar methods for saving a document to a collection, but the representation of the document itself will usually be whatever is most natural to each language. A document in Ruby is represented using a Ruby hash, in Python a Python dictionary is used, and in Java, which lacks any primitive analogous to a document, a document-builder class is used. Because the drivers provide a rich, language-centric interface to the database, very little abstraction beyond the driver itself is required to build an application. This contrasts notably with building an application on an RDBMS, where a library that mediates between the relational data model of the database and the object-oriented model of most modern programming languages, is usually necessary. That said, even if the heft of an object-relational mapper isn't required, a thin wrapper to handle validations and type checking is frequently used atop a MongoDB driver.[7]

Footnote 7 A few popular wrappers at the time of this writing include Morphia for Java, Doctrine for PHP, and MongoMapper for Ruby.

10gen officially supports drivers for C, C++, C#, Erlang, Haskell, Java, Perl, PHP, Python, Scala, and Ruby. If you need support for another language, there's probably a community-supported driver for it. If no community-supported driver exists for your language, specifications for building a new driver are documented at mongodb.org. Since all of the officially-supported drivers are used heavily in production and provided under the Apache License, there are already plenty of good examples freely available for would-be driver authors.

Beginning in Chapter 3, we describe how the drivers work and how to use them to write programs.

1.3.4 Command-line Tools

MongoDB is bundled with several command-line utilities. They are briefly described below:

- `mongodump`, `mongorestore`: Standard utilities for backing up and restoring a database. `mongodump` saves the database's data in its native BSON format and thus is best used for backups only. mongodump has the advantage of being usable for hot backups and can be easily restored with `mongorestore`.
- `mongoexport`, `mongoimport`: These utilities export and import both JSON and CSV data; this is useful if you need your data in widely-supported formats. `mongoimport` can also be good for initial imports of large datasets, although we should note in passing that it's often desirable to adjust the data model to take best advantage of MongoDB. In those cases, it's frequently easier to import the data with a custom script using one of the drivers.
- `mongosniff`: A wire-sniffing tool for viewing operations sent to the database. Essentially translates the BSON going over the wire to human-readable shell statements.
- `mongostat`: Similar to `iostat`, constantly polls MongoDB and the system to provide helpful stats, including the number of operations per second (inserts, queries, updates, deletes, etc.), the amount of virtual memory allocated, and the number of connections to the server.

1.4 Why MongoDB?

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

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

Of course, the aforementioned claims are rather bold. In order to substantiate them, we're going to take a broad look at the varieties of databases currently in use and contrast them with MongoDB. Next, we'll discuss MongoDB's use cases and

provide examples of them in production. Finally, we'll discuss some important practical considerations for using MongoDB.

1.4.1 MongoDB vs. Other Databases

There's been an explosion in the number of databases available, and it can be difficult to weigh one against another. Fortunately, most of these database fall under one of a few categories. In the sections that follow, we describe simple and sophisticated key-value stores, relational databases, and document databases, and show how these compare and contrast with MongoDB.

Table 1.1 Database Families

	Examples	Data Model	Scalability Model	Use Cases
Simple key-value stores	Memcached	Key-value, where is value is a binary blob.	Variable. Memcached can scale across nodes using available RAM to convert into a single, monolithic data store.	Caching. Web ops.
Sophisticated key-value stores	Cassandra, Project Voldemort, Riak	Variable. Cassandra uses a key-value structure known as a column. Voldemort stores binary blobs.	Eventually-consistent, multi-node distribution for high availability and easy failover.	High throughput verticals (activity feeds, message queues). Caching. Web ops.
Relational Databases	Oracle, MySQL, PostgreSQL	Tables.	Vertical scaling. Limited support for clustering and manual partitioning.	System requiring transactions (banking, finance) or SQL. Normalized data model.

SIMPLE KEY-VALUE STORES

Simple key-value stores do just what the name implies: they index values based on a supplied key. A common use case is caching. For instance, suppose you needed to cache an HTML page rendered by your app. The key in this case might be the page's URL, and the value would be the rendered HTML itself. Note that as far as a key-value store is concerned, the value is simply an array of bytes. There's no enforced schema, as you would find in a relational database, nor is there any concept of data types. This naturally limits the operations permitted by key-value stores: you can put a new value and then use its key either to retrieve that value or delete it. Systems with such simplicity are generally fast and scalable. And the best-known simple key-value store is memcached (pronounced "mem-cash-dee"). Memcached stores its data in memory only, so that while it's not persistent, it's very fast. The other great feature is that the memory can be distributed across multiple web servers and yet act as a single data store, eliminating the complexity of maintaining cache state across machines.

Compared with MongoDB, a simple key-value store like memcached will allow for faster reads and writes, but such a system can act as a primary data store only in rare cases. Simple key/value stores are best used as adjuncts, either as a caching layer atop a more traditional database or as a simple persistence layer for an ephemeral service such as a job queue.

SOPHISTICATED KEY-VALUE STORES

It's possible to refine the simple key-value model just described to handle complicated read/write schemes or to provide a more functional data model. In these cases, we end up with what we'll term a sophisticated key-value store. One example is Amazon's Dynamo, described in a widely-studied white paper entitled "Dynamo: Amazon's Highly Available Key-value Store." The aim of Dynamo is to provide a system robust enough to continue functioning in the face of network failures, data center outages, and other similar disruptions. This requires that the system can always be read from and written to, which essentially requires that data be automatically replicated across multiple nodes. If a node fails, a user of the system, perhaps in this case a customer with an Amazon shopping cart, won't experience any interruptions in service. Dynamo provides ways of resolving the inevitable conflicts that arise when a system allows the same data to be written to multiple nodes. At the same time, Dynamo is easily scaled. Because the system is master-less, that is, all nodes are equal, it's easy to understand the system as a whole, and nodes can be added easily. Although Dynamo is a proprietary system, the ideas used to build it have inspired many systems falling under the "NoSQL" umbrella, including Cassandra, Project Voldemort, and Riak.

Looking at who developed these sophisticated key-value stores, and how they've been used in practice, we can see just where these systems shine. Let's take Cassandra, which implements many of Dynamo's scaling properties while providing a column-oriented data model inspired by Google's BigTable. Cassandra is an open-sourced version of a data store built by Facebook for its inbox search feature. [8] The system scaled horizontally to index over 50TB of inbox data, allowing for searches on inbox keywords and recipients. Data was indexed by user id, where each record consisted of an array of search terms for keyword searches and an array of recipient ids for recipient searches.[9]

Footnote 8 It's worth noting that Facebook abandoned Cassandra for HBase in November 2010. See http://highscalability.com/blog/2010/11/16/facebooks-new-real-time-messaging-system-hbase-to-store-135.html for details.

Footnote 9 http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf

These sophisticated key-value stores were developed by major Internet companies like Amazon, Google, and Facebook, to manage cross sections of systems with extraordinarily large amounts of data. In other words, sophisticated key-value stores manage a relatively self-contained domain that demands significant storage and availability. Because of their master-less architecture, these

systems scale easily with the addition of nodes. They opt for eventual-consistency, which means that reads don't necessarily reflect the latest write. But what they get in exchange for weaker consistency is the ability to write in the face of any one node's failure.

This contrasts with MongoDB, which provides strong consistency, a richer data model, and secondary indexes. The last two of these attributes go hand-in-hand; if a system allows for the modeling of multiple domains, as for example what would be required to build a complete web application, then it's going to be necessary to query across the entire data model. And so secondary indexes become necessary. Compared with these sophisticated key-value stores, MongoDB can be considered a more general solution to the problem of large, scalable web applications. MongoDB's auto-sharded scaling architecture is sometimes criticized because it doesn't replicate all the ideas in Dynamo. But anyone who studies Yahoo's PNUTS data store, or even Google's BigTable, will see that the ideas in MongoDB have been successfully deployed for demanding web applications.

RELATIONAL DATABASES

Much has already been said of relational databases in this introduction, so in the interest of brevity, we need only discuss what RDBMSs have with common with MongoDB and where they diverge. MongoDB and MySQL[10] are both capable of representing a rich data model, although where MySQL uses fixed-schema tables, MongoDB has schema-free documents. MySQL and MongoDB both support B+ tree indexes, and those accustomed to working with indexes in MySQL can expect similar behavior in MongoDB. Of course, MySQL supports both joins and transactions, so if you must use SQL or if you require transactions, then you'll have to use MySQL or another RDBMS. That said, MongoDB document model is often rich enough to represent objects without requiring joins. And its updates can be atomically applied to individual documents, providing a light subset of what's possible with traditional transactions. Both systems support replication. As for scalability, MongoDB has been designed to scale horizontally, with sharding and failover handled automatically. Any sharding on MySQL has to be managed manually, and given the complexity involved, it's more common to see a vertically scaled MySQL system.

Footnote 10 We're using MySQL here somewhat generically, since the features we're describing apply to most relational databases.

DOCUMENT DATABASES

Few databases identify themselves as document databases. As of this writing, the only well-known document database apart from MongoDB is Apache's CouchDB. CouchDB's document model is similar, although data is stored in plain text as JSON, whereas MongoDB uses the BSON binary format. Like MongoDB, CouchDB supports secondary indexes; the difference is that the indexes are defined by writing map reduce functions, which is a bit more involved than the declarative syntax using by MySQL and MongoDB. They also scale differently. CouchDB doesn't partition data across machines; rather, each CouchDB node is a complete replica of every other.

1.4.2 Use Cases and Production Deployments

Let's be honest here. You're not going to choose a database solely on the basis of its features. You need to know that real business are using it successfully. Here we provide a few broadly-defined use cases for MongoDB and give some examples of its use in production.[11]

Footnote 11 For an up-to-date list of MongoDB production deployments, see http://www.mongodb.org/display/DOCS/Production+Deployments.

WEB APPLICATIONS

MongoDB is well-suited as a primary data store for web applications. Even a simple web application will require numerous data models for managing users, sessions, app-specific data, uploads, permissions, to say nothing of the overarching domain. Just as this aligns well with the tabular approach provided by relational databases, so too does it benefit from the collection and document model of MongoDB. And because documents can represent rich data structures, the number of collections needed will usually be much smaller than the number of tables required to model the same data using a fully-normalized relational model. In addition, dynamic queries and secondary indexes allow for the easy implementation of most queries familiar to SQL developers. Finally, as a web application grows, MongoDB provides a clear path for scale.

In production, MongoDB has proven capable of managing all aspects of an app, from the primary data domain to add-ons like logging and real-time analytics. This is the case with The Business Insider (TBE), which has used MongoDB as its primary data store since January, 2008. TBE is basically a news site, although it gets substantial traffic, serving over one million unique page views per day. What's interesting in this case is that in addition to handling the site's main content (posts,

comments, users, etc.), MongoDB also processes and stores real-time analytics data, where between 2 and 8 updates occur on the database for each page view. These analytics are used by TBE to generate dynamic heat-maps indicating click-through rates for the various news stories. The site doesn't host enough data to warrant sharding yet, but it does make use of replica sets to ensure automatic failover in the event of an outage.

AGILE DEVELOPMENT

Regardless of what you may thing about the agile development movement, none of us can deny that building an application quickly is desirable. A number of development teams, including those from Shutterfly and The New York Times, have chose MongoDB in part because they can develop application much more quickly on it than on relational databases. One obvious reason for this is that MongoDB has no fixed schema. So all the time spent committing, communicating, and applying schema changes is saved.

In addition, less time need be spent data modeling, since the structures of modern web applications are more naturally represented as MongoDB documents.

ANALYTICS AND LOGGING

We just alluded to the idea that MongoDB can work well for analytics and logging, and the number of application using MongoDB grows by the day. Often, a well-established site will begin its forays into the MongoDB world with special apps dedicated to analytics. These sites include GitHub, Disqus, Justin.tv, and Gilt Groupe, among many others.

MongoDB's relevance to analytics derives from its speed and from two key features: targeted atomic update and capped collections. Atomic updates let client efficiently increment counters and push values onto arrays. Capped collections, useful for logging, feature fixed allocation, which lets them age out automatically. Storing analytics and logging data in a database, as opposed to on the file system, provides easier organization and much greater query power. Now, instead of using `grep` or a custom log search utility, users can employ the MongoDB query language they know and love to examine log output.

CACHING

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

UNSTRUCTURED DATA

Have look at this code example[12]:

Footnote 12 This idea comes from http://eliothorowitz.com/post/459890033/streaming-twitter-into-mongodb

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

Here we're pulling down a small sample of the tweet stream and piping that directly into a MongoDB collection. Since the stream produces JSON documents, there's no need to munge the data before sending it to database. The `mongoimport` tool directly translates the data to BSON. This means that each tweet is stored with its structure intact, as a separate document in the collection. So we can immediately operate on the data, whether we want to query, index, or perform a map-reduce aggregation on it.

If your application needs to consume a JSON API, then having a system that so easily translates JSON is invaluable.

1.5 Tips and Limitations

For all these good features, it's always worthwhile keeping in mind a system's real trade-offs and limitations. And there are indeed some practical considerations that should be noted before building a real-world application on MongoDB and running in production. Most of these are consequences of MongoDB's use of memory-mapped files.

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

Footnote 13 64-bit architectures can theoretically address up to 16.3 billion GB of memory, which is for all intents and purposes unlimited.

A second consequence of using `mmap` is that MongoDB will allocate memory automatically, as needed. So it will be trickier running the database in a shared environment or on a non-dedicated instance. There are a few techniques for running MongoDB in such an environment, and we'll discuss these in Chapter 13. But do note that MongoDB runs best on a dedicated server.

Finally, it's recommended to run MongoDB with replication, especially if you're not running with journaling enabled. Because MongoDB uses memory-mapped files, any unclean shutdown of a `mongod` not running with journaling durability can result in corruption. Therefore, it's necessary to have a replicated backup available for failover. Of course, this is good advice for any database, and it would be imprudent not to do likewise with any serious MySQL deployment, for example.

1.6 Summary

We've covered a lot. To briefly summarize, MongoDB is an open-source, document-based database management system. Designed for the data and scalability requirements of modern Internet applications, MongoDB features dynamic queries and secondary indexes; fast atomic updates and complex aggregations; and support for both master-slave replication, replica sets with automatic failover, and automatically-managed sharding for distributing data across multiple machines.

That a mouthful; but if you've read this far, you should have a good sense for all these capabilities. You're probably itching to code. After all, it's one thing to talk about a database's features but quite another to use the database in practice. Fortunately, that's exactly what we'll be doing in the next two chapters. First, we'll get acquainted with the MongoDB JavaScript shell, which is an incredibly useful tool for interacting with the database. Then, in Chapter 3, we'll start experimenting with the driver and write a simple MongoDB-based application in Ruby.