

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 Data Model

In the previous chapter, you learned how to install MongoDB on two commonly used platforms (Windows and Linux), as well as how to extend the database with some additional drivers. In this chapter, you will shift your attention from the operating system and instead examine the general design of a MongoDB database. Specifically, you'll learn what collections are; what documents look like; how indexes work and what they do; and finally, when and where to reference data instead of embedding data. We touched on some of these concepts briefly in Chapter 1. To refresh your memory though, we'll address some of this material again in this chapter, but this time in more detail. Throughout this chapter, you will see code examples designed to give you get a good feel of what is being discussed. Do not worry too much about the commands you'll be looking at, however, because they will be discussed extensively in Chapter 4.

Designing the Database

As you learned in the first two chapters, a MongoDB database is non-relational and schemaless. This means that a MongoDB database isn't bound to any predefined columns or datatypes the way that relational databases are (such as MySQL). The biggest benefit of this implementation is that working with data is extremely flexible because there is no actual predefined structure required in your documents.

To put it more simply: you are perfectly capable of having one collection that contains hundreds or even thousands of documents that all carry a different structure—without breaking any of the MongoDB databases rules.

One of the benefits of this flexible schemaless design is that you won't be restricted when programming in a dynamically typed programming language (e.g., Python or PHP). Indeed, it would be a severe limitation if your extremely flexible and dynamically capable programming language couldn't be used to its full potential because of the innate limitations of your database.

Let's take another glance at what the data design of a document in MongoDB looks like, paying particular attention to how flexible data in MongoDB is compared to data in a relational database. In MongoDB, a *document* is an item that contains the actual data, comparable to a row in SQL. In the following example, you will see how two completely different types of documents can co-exist in a single collection called *Media* (note that a *collection* is equivalent to a table in the world of SQL):

```
{
  "Type": "CD",
  "Artist": "Nirvana",
  "Title": "Nevermind",
  "Genre": "Grunge",
  "Releasedate": "1991.09.24",
  "Tracklist": [
    {
```

```

    "Track" : "1",
    "Title" : "Smells like teen spirit",
    "Length" : "5:02"
  },
  {
    "Track" : "2",
    "Title" : "In Bloom",
    "Length" : "4:15"
  }
]

{
  "type": "Book",
  "Title": "Definite Guide to MongoDB: The NoSQL
  Database for Cloud and Desktop Computing, the",
  "ISBN": "987-1-4302-3051-9",
  "Publisher": "Apress",
  "Author": [
    "Plugge, Eelco",
    "Membrey, Peter",
    "Hawkins, Tim"
  ]
}

```

As you might have noticed when looking at the pair of preceding documents, most of the fields aren't closely related to one another. Yes, they both have fields called `Title` and `Type`; but apart from that similarity, the documents are completely different. Nevertheless, these two documents are contained in a single collection called `Media`.

MongoDB is called a *schemaless* database, but that doesn't mean MongoDB's data structure is completely devoid of schema. For example, you do define collections and indexes in MongoDB (you will learn more about this later in the chapter). Nevertheless, you do not *need* to predefine a structure for any of the documents you will be adding, as is the case when working with MySQL, for example.

Simply stated, MongoDB is an extraordinarily dynamic database; the preceding example would never work in a relational database, unless you also added each possible field to your table. Doing so would be a waste of both space and performance, not to mention highly disorganized.

Drilling Down on Collections

As mentioned previously, a collection is a commonly used term in MongoDB. You can think of a collection as a container that stores your documents (i.e., your data), as shown in Figure 3–1.

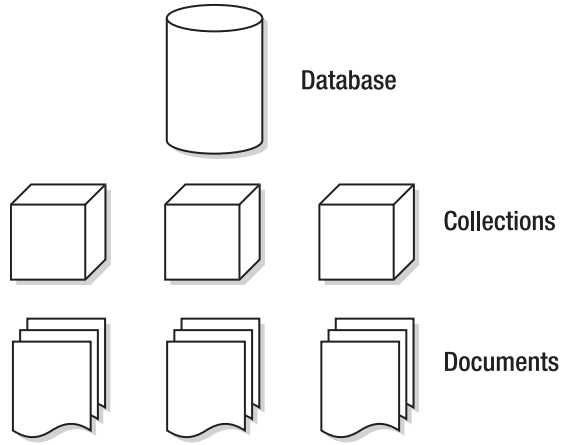


Figure 3-1. The MongoDB database model

Now compare the MongoDB database model to a typical model for a relational database (see Figure 3-2).

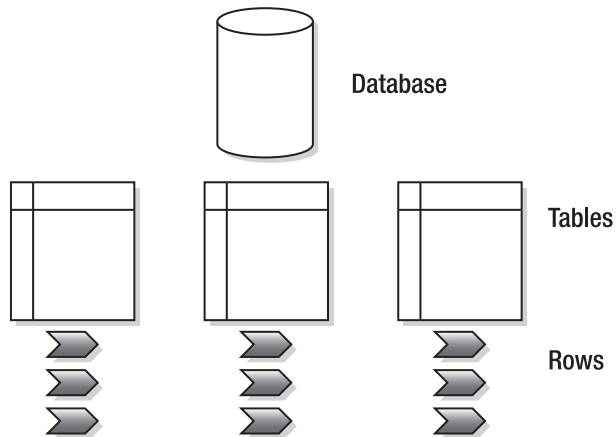


Figure 3-2. A typical relational database model

As you can see, the general structure is the same between the two types of databases; nevertheless, you do not use them in even remotely similar manners. There are several types of collections. The default collection type is expandable in size: the more data you add to it, the larger it becomes. There are also collections available that are *capped*. These *capped collections* can only contain a certain amount of data before the oldest document gets replaced by a newer document (you will learn more about these collections in Chapter 4).

Every collection in MongoDB has a unique name. This name should begin with a letter, or optionally, an underscore (`_`) when created using the `createCollection` function. The name can contain

numbers and letters; however, the \$ symbol is reserved by MongoDB. Generally, it's recommended that you keep the collection's name simple and short (to around nine characters or so); however, the maximum number of allowed characters in a collection name is 128. Obviously, there isn't much practical reason to create such a long name.

A single database has a default limit of 24,000 namespaces per database. Each collection accounts for at least two namespaces: one for the collection itself and one more for the first index created in the collection. If you were to add more indexes per collection, however, another namespace would be used. In theory, this means that each database can have up to 12,000 collections by default, assuming each collection only carries one index. However, this limit on the number of namespaces can be increased by providing the `nssize` parameter when executing the MongoDB service application (*mongod*).

Using Documents

Recall that a document consists of key-value pairs. For example, the pair "type" : "Book" consists of a key named `type`, and its value, `Book`. Keys are written as *strings*, but the values in them can vary tremendously. Values can be any of a rich set of datatypes, such as arrays or even binary data. Remember: MongoDB stores its data in BSON format (see Chapter 1 for more information on this topic).

Next, let's look at the other possible types of data you can add to a document, and what you use them for:

- *String*: This commonly used datatype contains a string of text (or any other kind of characters). This datatype is used mostly for storing text values (e.g., "Country" : "Japan").
- *Integer (32b and 64b)*: This type is used to store a numerical value (e.g., { "Rank" : 1 }). Note that there are no quotes placed before or after the integer.
- *Boolean*: This datatype can be set to either TRUE or FALSE.
- *Double*: This datatype is used to store floating point values.
- *Min / Max keys*: This datatype is used to compare a value against the lowest and highest BSON elements, respectively.
- *Arrays*: This datatype is used to store arrays (e.g., ["Membrey, Peter", "Plugge, Eelco", "Hawkins, Tim"]).
- *Timestamp*: This datatype is used to store a timestamp. This can be handy for recording when a document has been modified or added.
- *Object*: This datatype is used for embedded documents.
- *Null*: This datatype is used for a Null value.
- *Symbol*: This datatype is used identically to a string (see above); however, it's generally reserved for languages that use a specific symbol type.
- *Date **: This datatype is used to store the current date or time in UNIX time format (POSIX time).
- *Object ID **: This datatype is used to store the document's ID.
- *Binary data **: This datatype is used to store binary data.

- *Regular expression* *: This datatype is used for regular expressions. All options are represented by specific characters provided in alphabetical order. You will learn more about regular expressions in Chapter 4.
- *JavaScript Code* *: This datatype is used for JavaScript code.

The last five datatypes (date, object id, binary data, regex, and JavaScript code) are non-JSON datatypes; specifically, they are special datatypes that BSON allows you to use. In Chapter 4, you will learn how to identify your datatypes by using the \$type operator.

In theory, this all probably sounds straightforward. However, you might wonder how you go about actually “designing” the document itself, including what information to put in it. Because a document can contain any type of data, you might think there is no need to reference information from inside another document. In the next section, we’ll look at the pros and cons of embedding information in a document vs. referencing that information from another document.

Embedding vs. Referencing Information in Documents

You can choose either to embed information into a document or reference that information in another document. Embedding information simply means that you place a certain type of data (e.g., an array containing more data) into the document itself. Referencing information simply means that you create a reference to another document that contains that specific data. Typically, you reference information when you use a relational database. For example, assume you wanted to use a relational database to keep track of your CDs, DVDs, and books. In this database, you might have one table for your CD collection and another table that stores the tracklists of your CDs. Thus, you would probably need to query multiple tables to acquire a list of tracks from a specific CD.

With MongoDB (and other non-relational databases), however, it would be much easier to embed such information instead. After all, the documents are natively capable of doing so. Adopting this approach keeps your database nice and tidy, ensures that all related information is kept in one single document, and even works much faster because the data is then co-located on the disk.

Now let’s look at the differences between embedding and referencing information by looking at a real-world scenario: storing CD data in a database.

In the relational approach, your data structure might look something like this:

```
|_media
  |_cds
    |_id, artist, title, genre, releasedate
  |_ cd_tracklists
    |_cd_id, songtitle, length
```

In the non-relational approach, your data structure might look something like this:

```
|_media
  |_items
    |_<document>
```

In the non-relational approach, the document might look something like the following:

```
{
  "Type": "CD",
  "Artist": "Nirvana",
```

```

    "Title": "Nevermind",
    "Genre": "Grunge",
    "Releasedate": "1991.09.24",
    "Tracklist": [
      {
        "Track" : "1",
        "Title" : "Smells Like Teen Spirit",
        "Length" : "5:02"
      },
      {
        "Track" : "2",
        "Title" : "In Bloom",
        "Length" : "4:15"
      }
    ]
  }
}

```

In the preceding example, the tracklist information is actually embedded in the document itself. This approach is both incredibly efficient and well organized. All the information that you wish to store regarding this CD is added to a single document. In the relational version of the CD database, this requires at least two tables; in the non-relational database, this requires only one collection and one document.

When retrieving information for a given CD, the information only needs to be loaded from one document into RAM, not from multiple documents. Remember that every reference requires another query in the database.

■ **Tip** The rule of the thumb when using MongoDB is to embed data whenever you can. This approach is far more efficient and almost always viable.

At this point, you might be wondering about the use case where an application has multiple users. Generally speaking, a relational database version of the aforementioned CD app would require that you have one table that contains all your users and two tables for the items added. For a non-relational database, it would be good practice to have separate collections for the users and the items added. For these kinds of problems, MongoDB allows you to create references in two ways: manually or automatically. In the latter case, you use the DBRef specification, which provides more flexibility in case a collection changes from one document to the next. You will learn more about these two approaches in Chapter 4.

Creating the `_id` Field

Every object within the MongoDB database contains a unique identifier to distinguish that object from every other object. This unique identifier is called the `_id` key, and it is added automatically to every document you create in a collection.

The `_id` key is the first attribute added in each new document you create. This remains true even if you do not tell MongoDB to create this key. For example, none of the code in the preceding examples used the `_id` key. Nevertheless, MongoDB created an `_id` key for you automatically in each document. It did so because `_id` key is a mandatory element for each document in the collection.

If you do not specify the `_id` value manually, then the type will be set to a special BSON datatype that consists of a 12-byte binary value. Due to its design, this value has a reasonably high probability of being unique. The 12-byte value consists of a 4-byte timestamp (seconds since epoch), a 3-byte machine id, a 2-byte process id, and a 3-byte counter. It's good to know that the counter and timestamp fields are stored in *Big Endian*. This is because MongoDB wants to ensure that there is an increasing order to these values, and a Big Endian approach suits this requirement best.

■ **Note** Big Endian and Little Endian refer to how each individual bytes/bits are stored in a longer data word in the memory. Big Endian simply means that the highest value gets saved first. Similarly, Little Endian means that the smallest value gets saved first.

Figure 3–3 shows how the value of the `_id` key is built up and where the values come from.

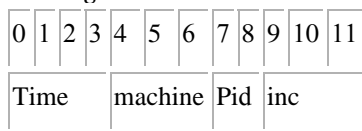


Figure 3–3. Creating the `_id` key in MongoDB

Every additional supported driver that you load when working with MongoDB (such as the PHP driver or the Python driver) supports this special BSON datatype and uses it whenever new data is created. You can also invoke `ObjectId()` from the MongoDB shell to create a value for an `_id` key. Optionally, you can specify your own value by using `ObjectId(string)`, where `string` represents the specified hex string.

Building Indexes

As mentioned in Chapter 1, an index is nothing more than a data structure that collects information about the values of specified fields in the documents of a collection. This data structure is used by MongoDB's query optimizer to quickly sort through and order the documents in a collection.

Remember that indexing ensures a quick lookup from data in your documents. Basically, you should view an index as a predefined query that was executed and had its results stored. As you can imagine, this enhances query-performance dramatically. The general rule of the thumb in MongoDB is that you should create an index for the same sort of scenarios where you would want to have an index in MySQL.

The biggest benefit of creating your own indexes is that querying for often-used information will be incredibly fast because your query won't need to go through your entire database to collect this information.

Creating (or deleting) an index is relatively easy—once you get the hang of it, anyway. You will learn how to do so in Chapter 4, which covers how to work with data. You will also learn some more advanced techniques for taking advantage of indexing in Chapter 10, which covers how to maximize performance.

Impacting Performance with Indexes

You might wonder why you would ever need to delete an index, rebuild your indexes, or even delete all indexes within a collection. The simple answer is that doing so lets you clean up some irregularities. For instance, sometimes the size of a database can increase dramatically for no apparent reason. Other times, the space used by the indexes might strike you as excessive.

Another good thing to keep in mind: you can have a maximum of 40 indexes per collection. Generally speaking, this is way more than you should need, but you could potentially hit this limit someday.

■ **Note** Adding an index increases query speed, but reduces insertion or deletion speed. It's best to consider only adding indexes for collections where the number of reads is higher than the number of writes. When more writes occur than reads, indexes may even prove to be counterproductive.

Finally, all index information is stored in the `system.indexes` collection in your database. For example, you can run the `indexes.find()` command to take a quick peek at the indexes that have been stored so far. The following line shows the sample data that has been added by default:

```
db.systems.indexes.find()
```

Implementing Geospatial Indexing

As was briefly mentioned in Chapter 1, MongoDB has implemented *Geospatial Indexing* since version 1.4. This means that, in addition to normal indexes, MongoDB also supports two-dimensional geospatial indexes that are designed to work in an optimal way with location-based queries. For example, you can use this feature to find a number of closest known items to your current location. Or you might further refine your search to query for a specified number of restaurants near your current location. This type of query can be particularly helpful if you are designing an application where you want to find the closest available branch office to a given customer's zipcode.

A document for which you want to add geospatial information must contain either a subobject or an array where the first two elements contain the x and y coordinates (or y,x), as in the following example:

```
{ loc : { lat : 52.033475, long: 5.099222 } }
```

Once the preceding information is added to a document, you can create the index (or even create the index beforehand, of course) and give the `ensureIndex()` function the `2d` parameter:

```
> db.places.ensureIndex( { loc: "2d" } )
```

■ **Note** The `ensureIndex()` function is used to add a custom index. Don't worry about the syntax of this function at this time—you will learn how to use this function in depth in the next chapter.

The `2d` parameter tells `ensureIndex()` that it's indexing a coordinate or some other form of two-dimensional information. By default, `ensureIndex()` assumes that a latitude/longitude key is given, and it uses a range of -180 to 180. However, you can overwrite these values using the `min` / `max` parameters:

```
> db.places.ensureIndex( { loc: "2d" }, { min : -500 , max : 500 } )
```

■ **Warning** At this time, you cannot insert values at the defined boundaries. For example, you cannot insert values such as (-180 -180) in the default boundaries or (-500 -500) in the example that used the `min` / `max` parameters.

You can also expand your geospatial indexes by using *secondary key* values (also known as *compound keys*). This can be useful when you intend to query on multiple values, such as a location (geospatial information) and a category (sort ascending):

```
> db.places.ensureIndex( { loc: "2d", category: 1 } )
```

■ **Note** At this time, the geospatial implementation is based on the idea that the world is perfectly flat. Thus, each degree of latitude and longitude is exactly 111km (69 miles) in length. However, this is only true exactly at the equator; the further you move away from the equator, the smaller the longitude becomes, approaching zero at the poles.

Querying Geospatial Information

In this chapter, we are concerned primarily with two things: how to model the data and how a database works in the background of an application. That said, manipulating geospatial information is increasingly important in a wide variety of applications, so we'll take a few moments to explain how to leverage geospatial information in a MongoDB database.

Once you've added data to your collection, and once the index has been created, you can do a geospatial query. For example, let's look at a few lines of simple yet powerful code that demonstrate how to use geospatial indexing.

Begin by starting up your MongoDB shell and selecting a database with the `use` function. In this case, the database is named `stores`:

```
> use stores
```

Once you've selected the database, you can define a few documents that contain geospatial information, and then insert them into the `places` collection (remember: you do not need to create the collection beforehand):

```
> db.places.insert( { name: "Su Shi's Sushi", loc: [52.12345, 6.749923] } )
```

```
> db.places.insert( { name: "Shi Su's Sushi", loc: [51.12345, 6.249923] } )
```

After you add the data, you need to tell the MongoDB shell to create an index based on the location information that was specified in the `loc` key, as in this example:

```
> db.places.ensureIndex ( { loc: "2d" } )
```

Once the index has been created, you can start searching for your documents. Begin by searching on an exact value (so far this is a “normal” query; it has nothing to do with the geospatial information at this point):

```
> db.places.find( { loc : [52,6] } )
>
```

The preceding search returns no results. This is because the query is *too* specific. A better approach in this case would be to search for documents that contain information *near* a given value. You can accomplish this using the `$near` operator, as in the following example:

```
> db.places.find( { loc : { $near : [52,6] } } )
{
  "_id" : ObjectId("4bc2de69b2571f7d62ee30a6"),
  "name" : "Su Shi's Sushi",
  "loc" : [ 52.12345, 6.749923 ]
}
{
  "_id" : ObjectId("4bc2de7cb2571f7d62ee30a7"),
  "name" : "Shi Su's Sushi",
  "loc" : [ 51.12345, 6.249923 ]
}
```

This set of results looks better. Using the `$near` operator causes the `find()` function to look for anything close to the coordinates of 52 and 6; the results are sorted by their distance from the point specified by the `$near` operator. The default output will be limited to one hundred results. If you feel this number is too few, then you can append the `limit` function to your query, as in this example:

```
> db.places.find( { loc : { $near : [52,6] } } ).limit(200)
```

■ **Note** There is a direct correlation between the number of results returned and how long a given query will take to execute.

In addition to the `$near` operator, MongoDB also includes a `$within` operator. You use this operator to find items in a particular shape. At this time, you can find items located in a `$box` or `$center` shape, where `$box` represents a rectangle and `$center` represents a circle. Let’s look at a couple additional examples that illustrate how to use these shapes.

To use the `$box` shape, you first need to specify the lower-left and the upper-right corners of the box, and then save these values into a variable. For example, the first line in the following code snippet stores the values in a variable called `box`, while the second line executes the query:

```
> box = [[40, 60], [4, 8]]
> db.places.find( { loc: { $within : { $box : box } } } )
```

The code to find in items in a `$circle` shape looks quite similar. In this case, you need to specify the center of the circle and its radius before executing the `find()` function:

```
> center = [50, 15]
> radius = 10
> db.places.find( { loc: { $within : { $center : [center, radius] } } } )
```

By default, the `find()` function is ideal for running queries. However, MongoDB also provides the `geoNear()` function, which functions like the `find()` function, but also displays the distance from the specified point for each item in the results. The `geoNear()` function also includes some additional diagnostics. The following example uses the `geoNear()` function to find the two closest results to the specified position:

```
> db.runCommand( { geoNear : "places", near : [52,6], num : 2 } )
{
  "ns" : "stores.places",
  "near" : "1100100000110000101110101001100000110000101110101001",
  "results" : [
    {
      "dis" : 0.7600121516405387,
      "obj" : {
        "_id" : ObjectId("4bc2de69b2571f7d62ee30a6"),
        "name" : "Su Shi's Sushi",
        "loc" : [
          52.12345,
          6.749923
        ]
      }
    },
    {
      "dis" : 0.911484268395441,
      "obj" : {
        "_id" : ObjectId("4bc2de7cb2571f7d62ee30a7"),
        "name" : "Shi Su's Sushi",
        "loc" : [
          51.12345,
          6.249923
        ]
      }
    }
  ],
  "stats" : {
    "time" : 0,
    "btreelocs" : 2,
    "nscanned" : 2,
    "objectsLoaded" : 2,
    "avgDistance" : 0.8357482100179898
  },
  "ok" : 1
}
```

That's all on this topic for now; however, you'll see a few more examples that show you how to leverage geospatial functions in this book's upcoming chapters.

Using MongoDB in the Real World

Now that you have MongoDB and its associated plug-ins installed, as well as having gained an understanding of the data model, it's time to get to work. In the remainder of the book, you will learn how to build, query, and otherwise manipulate a variety of sample MongoDB databases (see Table 3–1 for a quick view of the topics to come). Each chapter will stick primarily to using a single database that is unique to that chapter; we took this approach to make it easier to read this book in a modular fashion.

Table 3–1. MongoDB Sample Databases Covered in This Book

Chapter	Database Name	Topic
4	library	Working with data and indexes
5	Test	GridFS
6	contacts	PHP and MongoDB
7	Inventory	Python and MongoDB
8	blog	Blogs
9	blog	Database administration
10	blog	Optimization
11	blog	Replication
12	blog	Sharding

Summary

In this chapter, we looked at what's happening in the background of your database. We also explored the primary concepts of collections and documents in more depth; and we covered the datatypes supported in MongoDB, as well as how to embed data and reference data.

Next, we covered what indexes do, including when and why they should be used (or not).

We also touched on the concepts of geospatial indexing. For example, we covered how geospatial data can be stored; we also explained how you can search for such data using either the regular `find()` function or the more geospatially based `geoNear` database command.

In the next chapter, we'll take a closer look at how the MongoDB shell works, including which functions can be used to insert, find, update, or delete your data. We will also explore how conditional operators can help you with all of these functions.