Introduction

With billions of people using the Internet today, traditional RDBMS database solutions have difficulty meeting the rapidly growing need to handle large amounts of data. The growing trend is to introduce specialized databases that are not restricted to the conventions and the legacy overhead of traditional SQL databases. These databases are given the term NoSQL, meaning “Not Only SQL.” They are designed not to replace SQL databases, but to provide a different perspective in storing data.

This book teaches you the concepts of NoSQL through the MongoDB perspective. MongoDB is a NoSQL database that has a reputation for being easy to implement while still robust and scalable. It is currently the most popular NoSQL database in use. MongoDB has matured into a stable platform that several companies have leveraged to provide the data scalability they require.

Each hour in the book provides fundamentals for implementing and using MongoDB as back-end storage for high-performing applications. As you complete the 24 one-hour lessons in this book, you will gain practical understanding of how to build, use, and maintain a MongoDB database.

So pull up a chair, sit back, and enjoy the process of learning NoSQL through the perspective of MongoDB development.

How This Book Is Organized

This book is organized into four main parts:

**Part I**, “Getting Started with NoSQL and MongoDB,” covers the basic concepts of NoSQL, why you might want to use it, and available database types. It also covers MongoDB data structures and design concepts and explores what it takes to get MongoDB installed and running.

**Part II**, “Implementing NoSQL in MongoDB,” discusses the fundamental basics for implementing MongoDB. The hours in this part focus on creating databases and collections. They also cover the different methods of storing, finding, and retrieving data from the MongoDB database.

**Part III**, “Using MongoDB in Applications,” introduces you to the MongoDB drivers for some of the most common programming environments. A MongoDB driver is a library that provides the necessary tools to programmatically access and use the MongoDB database. This section covers
the drivers for Java, PHP, Python, and Node.js. Each programming language section is isolated, so if you have no interest in a particular language, you can skip its corresponding hour.

**Part IV, “Additional MongoDB Concepts,”** rounds out your knowledge of MongoDB by teaching you additional MongoDB concepts. In this part, you learn some of the basics of administrating MongoDB databases and look at more advanced MongoDB concepts such as replication, sharding, and GridFS storage.

### Code Examples

Two types of code examples appear in this book. The most common are code snippets that appear in-line with the text to illustrate talking points. Try It Yourself sections also provide code examples. These examples are more robust and are designed to run as standalone mini applications. To keep the code examples small and easy to follow, they are compressed, with little or no error checking, for example.

The Try It Yourself examples are presented in listings that include line numbers to make them easier to follow. They also include a filename in the listing title to indicate which file the listing came from. If the code listing in the Try It Yourself section has specific output, a follow-up listing shows you the console output of the code so that you can follow along as you are reading the book.

### Special Elements

As you complete each lesson, margin notes help you immediately apply what you just learned to your own web pages.

Whenever a new term is used, it is clearly highlighted—no flipping back and forth to a glossary.

---

**TIP**

Tips and tricks to save you precious time are set aside in Tip boxes so that you can spot them quickly.

---

**NOTE**

Note boxes highlight interesting information you want to be sure not to miss.

---

**CAUTION**

When you need to watch out for something, you’re warned about it in Caution boxes.
Q&A, Quiz, and Exercises

Every hour ends with a short question-and-answer session that addresses the kind of “dumb questions” all readers wish they dared to ask. A brief but complete quiz lets you test yourself to be sure you understand everything presented in the hour. Finally, one or two optional exercises give you a chance to practice your new skills before you move on.
What You’ll Learn in This Hour:

- How MongoDB structures data
- What data types MongoDB supports
- When to normalize and denormalize data
- How to plan your data model
- How capped collections work
- When to use indexing, sharding, and replication
- How to determine data life cycles

At the core of most large-scale applications and services is a high-performance data storage solution. The back-end data store is responsible for storing important data such as user account information, product data, accounting information, and blogs. Good applications require the capability to store and retrieve data with accuracy, speed, and reliability. Therefore, the data storage mechanism you choose must be capable of performing at a level that satisfies your application’s demand.

Several data storage solutions are available to store and retrieve the data your applications need. The three most common are direct file system storage in files, relational databases, and NoSQL databases. The NoSQL data store chosen for this book is MongoDB because it is the most widely used and the most versatile.

The following sections describe NoSQL and MongoDB and discuss the design considerations to review before deciding how to implement the structure of data and the database configuration. The sections cover the questions to ask and then address the mechanisms built into MongoDB that satisfy the resulting demands.
What Is NoSQL?

A common misconception is that the term NoSQL stands for “No SQL.” NoSQL actually stands for “Not only SQL,” to emphasize the fact that NoSQL databases are an alternative to SQL and can, in fact, apply SQL-like query concepts.

NoSQL covers any database that is not a traditional relational database management system (RDBMS). The motivation behind NoSQL is mainly simplified design, horizontal scaling, and finer control over the availability of data. NoSQL databases are more specialized for types of data, which makes them more efficient and better performing than RDBMS servers in most instances.

NoSQL seeks to break away from the traditional structure of relational databases, and enable developers to implement models in ways that more closely fit the data flow needs of their system. This means that NoSQL databases can be implemented in ways that traditional relational databases could never be structured.

Several different NoSQL technologies exist, including the HBase column structure, the Redis key/value structure, and the Virtuoso graph structure. However, this book uses MongoDB and the document model because of the great flexibility and scalability offered in implementing backend storage for web applications and services. In addition, MongoDB is by far the most popular and well-supported NoSQL language currently available. The following sections describe some of the NoSQL database types.

Document Store Databases

Document store databases apply a document-oriented approach to storing data. The idea is that all the data for a single entity can be stored as a document, and documents can be stored together in collections.

A document can contain all the necessary information to describe an entity. This includes the capability to have subdocuments, which in RDBMS are typically stored as an encoded string or in a separate table. Documents in the collection are accessed via a unique key.

Key-Value Databases

The simplest type of NoSQL database is the key-value stores. These databases store data in a completely schema-less way, meaning that no defined structure governs what is being stored. A key can point to any type of data, from an object, to a string value, to a programming language function.

The advantage of key-value stores is that they are easy to implement and add data to. That makes them great to implement as simple storage for storing and retrieving data based on a key. The downside is that you cannot find elements based on the stored values.
Column Store Databases

Column store databases store data in columns within a key space. The key space is based on a unique name, value, and timestamp. This is similar to the key-value databases; however, column store databases are geared toward data that uses a timestamp to differentiate valid content from stale content. This provides the advantage of applying aging to the data stored in the database.

Graph Store Databases

Graph store databases are designed for data that can be easily represented as a graph. This means that elements are interconnected with an undetermined number of relations between them, as in examples such as family and social relations, airline route topology, or a standard road map.

Choosing RDBMS, NoSQL, or Both

When investigating NoSQL databases, keep an open mind regarding which database to use and how to apply it. This is especially true with high-performance systems.

You might need to implement a strategy based on only RDBMS or NoSQL—or you might find that a combination of the two offers the best solution in the end.

With all high-performance databases, you will find yourself trying to balance speed, accuracy, and reliability. The following is a list of just some considerations when choosing a database:

- **What does my data look like?** Your data might favor a table/row structure of RDBMS, a document structure, or a simple key-value pair structure.

- **How is the current data stored?** If your data is stored in an RDBMS database, you must evaluate what it would take to migrate all or part to NoSQL. Also consider whether it is possible to keep the legacy data as is and move forward with new data in a NoSQL database.

- **How important is the guaranteed accuracy of database transactions?** A downside of NoSQL is that most solutions are not as strong in ACID (Atomic, Consistency, Isolation, Durability) as in the more well-established RDBMS systems.

- **How important is the speed of the database?** If speed is the most critical factor for your database, NoSQL might fit your data well and can provide a huge performance boost.

- **What happens when the data is not available?** Consider how critical it is for customers when data is not available. Keep in mind that customers view situations in which your database is too slow to respond as unavailability. Many NoSQL solutions, including MongoDB, provide a good high availability plan using replication and sharding.
How is the database being used? Specifically, consider whether most operations on the database are writes to store data or whether they are reads. You can also use this exercise as an opportunity to define the boundaries of how to split up data, enabling you to gear some data toward writes and other data toward reads.

Should I split up the data to leverage the advantages of both RDBMS and NoSQL? After you have looked at the previous questions, you might want to consider putting some of the data, such as critical transactions, in an RDBMS while putting other data, such as blog posts, in a NoSQL database.

Understanding MongoDB

MongoDB is an agile and scalable NoSQL database. The name *Mongo* comes from the word *humongous*. MongoDB is based on the NoSQL document store model, in which data objects are stored as separate documents inside a collection instead of in the traditional columns and rows of a relational database. The documents are stored as binary JSON or BSON objects.

The motivation of the MongoDB language is to implement a data store that provides high performance, high availability, and automatic scaling. MongoDB is extremely simple to install and implement, as you will see in upcoming hours. MongoDB offers great website back-end storage for high-traffic websites that need to store data such as user comments, blogs, or other items because it is fast, scalable, and easy to implement.

The following are some additional reasons MongoDB has become the most popular NoSQL database:

- **Document oriented**: Because MongoDB is document oriented, the data is stored in the database in a format that is very close to what you will be dealing with in both server-side and client-side scripts. This eliminates the need to transfer data from rows to objects and back.

- **High performance**: MongoDB is one of the highest-performing databases available. Especially in today’s world, where many people interact with websites, having a back end that can support heavy traffic is important.

- **High availability**: MongoDB’s replication model makes it easy to maintain scalability while keeping high performance and scalability.

- **High scalability**: MongoDB’s structure makes it easy to scale horizontally by sharding the data across multiple servers.

- **No SQL injection**: MongoDB is not susceptible to SQL injection (putting SQL statements in web forms or other input from the browser that compromises the DB security) because objects are stored as objects, not by using SQL strings.
Understanding Collections

MongoDB groups data through collections. A collection is simply a grouping of documents that have the same or a similar purpose. A collection acts similarly to a table in a traditional SQL database. However, it has a major difference: In MongoDB, a collection is not enforced by a strict schema. Instead, documents in a collection can have a slightly different structure from one another, as needed. This reduces the need to break items in a document into several different tables, as is often done in SQL implementations.

Understanding Documents

A document is a representation of a single entity of data in the MongoDB database. A collection consists of one or more related objects. A major difference exists between MongoDB and SQL, in that documents are different from rows. Row data is flat, with one column for each value in the row. However, in MongoDB, documents can contain embedded subdocuments, providing a much closer inherent data model to your applications.

In fact, the records in MongoDB that represent documents are stored as BSON, a lightweight binary form of JSON. It uses field:value pairs that correspond to JavaScript property:value pairs that define the values stored in the document. Little translation is necessary to convert MongoDB records back into JSON strings that you might be using in your application.

For example, a document in MongoDB might be structured similar to the following, with name, version, languages, admin, and paths fields:

```json
{
    name: "New Project",
    version: 1,
    languages: ["JavaScript", "HTML", "CSS"],
    admin: {name: "Brad", password: "****"},
    paths: {temp: "/tmp", project:"/opt/project", html: "/opt/project/html"}
}
```

Notice that the document structure contains fields/properties that are strings, integers, arrays, and objects, just as in a JavaScript object. Table 11.1 lists the different data types for field values in the BSON document.

The field names cannot contain null characters, dots (.), or dollar signs ($). In addition, the _id field name is reserved for the Object ID. The _id field is a unique ID for the system that consists of the following parts:

- A 4-byte value representing the seconds since the last epoch
- A 3-byte machine identifier
- A 2-byte process ID
- A 3-byte counter, starting with a random value
The maximum size of a document in MongoDB is 16MB, to prevent queries that result in an excessive amount of RAM or intensive hits to the file system. You might never come close to this, but you still need to keep the maximum document size in mind when designing some complex data types that contain file data into your system.

**MongoDB Data Types**

The BSON data format provides several different types used when storing the JavaScript objects to binary form. These types match the JavaScript type as closely as possible. It is important to understand these types because you can actually query MongoDB to find objects that have a specific property with a value of a certain type. For example, you can look for documents in a database whose timestamp value is a String object or query for ones whose timestamp is a Date object.

MongoDB assigns each data type an integer ID number from 1 to 255 when querying by type. Table 1.1 lists the data types MongoDB supports, along with the number MongoDB uses to identify them.

<table>
<thead>
<tr>
<th>Type</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>1</td>
</tr>
<tr>
<td>String</td>
<td>2</td>
</tr>
<tr>
<td>Object</td>
<td>3</td>
</tr>
<tr>
<td>Array</td>
<td>4</td>
</tr>
<tr>
<td>Binary data</td>
<td>5</td>
</tr>
<tr>
<td>Object ID</td>
<td>7</td>
</tr>
<tr>
<td>Boolean</td>
<td>8</td>
</tr>
<tr>
<td>Date</td>
<td>9</td>
</tr>
<tr>
<td>Null</td>
<td>10</td>
</tr>
<tr>
<td>Regular expression</td>
<td>11</td>
</tr>
<tr>
<td>JavaScript</td>
<td>13</td>
</tr>
<tr>
<td>Symbol</td>
<td>14</td>
</tr>
<tr>
<td>JavaScript (with scope)</td>
<td>15</td>
</tr>
<tr>
<td>32-bit integer</td>
<td>16</td>
</tr>
<tr>
<td>Timestamp</td>
<td>17</td>
</tr>
<tr>
<td>64-bit integer</td>
<td>18</td>
</tr>
</tbody>
</table>
Another point to be aware of when working with the different data types in MongoDB is the order in which they are compared when querying to find and update data. When comparing values of different BSON types, MongoDB uses the following comparison order, from lowest to highest:

1. Min key (internal type)
2. Null
3. Numbers (32-bit integer, 64-bit integer, double)
4. Symbol, String
5. Object
6. Array
7. Binary data
8. Object ID
9. Boolean
10. Date, timestamp
11. Regular expression
12. Max key (internal type)

### Planning Your Data Model

Before you begin implementing a MongoDB database, you need to understand the nature of the data being stored, how that data will be stored, and how it will be accessed. Understanding these concepts helps you make determinations ahead of time and structure the data and your application for optimal performance.

Specifically, you should ask yourself the following questions:

- What basic objects will my application be using?
- What is the relationship between the different object types—one-to-one, one-to-many, or many-to-many?
How often will new objects be added to the database?

How often will objects be deleted from the database?

How often will objects be changed?

How often will objects be accessed?

How will objects be accessed—by ID, property values, comparisons, or other?

How will groups of object types be accessed—common ID, common property value, or other?

When you have the answers to these questions, you are ready to consider the structure of collections and documents inside MongoDB. The following sections discuss different methods of document, collection, and database modeling you can use in MongoDB to optimize data storage and access.

**Normalizing Data with Document References**

Data normalization is the process of organizing documents and collections to minimize redundancy and dependency. This is done by identifying object properties that are subobjects and that should be stored as a separate document in another collection from the object’s document. Typically, this is useful for objects that have a one-to-many or many-to-many relationship with subobjects.

The advantage of normalizing data is that the database size will be smaller because only a single copy of objects will exist in their own collection instead of being duplicated on multiple objects in single collection. Additionally, if you modify the information in the subobject frequently, then you need to modify only a single instance instead of every record in the object’s collection that has that subobject.

A major disadvantage of normalizing data is that, when looking up user objects that require the normalized subobject, a separate lookup must occur to link the subobject. This can result in a significant performance hit if you are accessing the user data frequently.

An example of when normalizing data makes sense is a system that contains users who have a favorite store. Each User is an object with name, phone, and favoriteStore properties. The favoriteStore property is also a subobject that contains name, street, city, and zip properties.

However, thousands of users might have the same favorite store, so you see a high one-to-many relationship. Therefore, storing the FavoriteStore object data in each User object doesn’t make sense because it would result in thousands of duplications. Instead, the FavoriteStore object should include an _id object property that can be referenced from documents in the user’s
stores collection. The application can then use the reference ID `favoriteStore` to link data from the `Users` collection to `FavoriteStore` documents in the `FavoriteStores` collection.

Figure 1.1 illustrates the structure of the `Users` and `FavoriteStores` collections just described.

![Diagram showing the structure of data access and processing](image)

**FIGURE 1.1**
Defining normalized MongoDB documents by adding a reference to documents in another collection.

## Denormalizing Data with Embedded Documents

Denormalizing data is the process of identifying subobjects of a main object that should be embedded directly into the document of the main object. Typically, this is done on objects that have mostly one-to-one relationships or that are relatively small and do not get updated frequently.

The major advantage of denormalized documents is that you can get the full object back in a single lookup without needing to do additional lookups to combine subobjects from other collections. This is a major performance enhancement. The downside is that, for subobjects with a one-to-many relationship, you are storing a separate copy in each document; this slows insertion a bit and takes up additional disk space.

An example of when normalizing data makes sense is a system that contains users’ home and work contact information. The user is an object represented by a `User` document with `name`, `home`, and `work` properties. The `home` and `work` properties are subobjects that contain `phone`, `street`, `city`, and `zip` properties.

The `home` and `work` properties do not change often for the user. Multiple users might reside in the same home, but this likely will be a small number. In addition, the actual values inside the subobjects are not that big and will not change often. Therefore, storing the home contact information directly in the `User` object makes sense.
The `work` property takes a bit more thinking. How many people are you really going to get who have the same work contact information? If the answer is not many, the `work` object should be embedded with the `User` object. How often are you querying the `User` and need the `work` contact information? If you will do so rarely, you might want to normalize `work` into its own collection. However, if you will do so frequently or always, you will likely want to embed `work` with the `User` object.

Figure 1.2 illustrates the structure of `Users` with the `home` and `work` contact information embedded, as described previously.

**FIGURE 1.2**
Defining denormalized MongoDB documents by implementing embedded objects inside a document.

### Using Capped Collections
A great feature of MongoDB is the capability to create a capped collection. A capped collection is a collection that has a fixed size. When a new document needs to be written to a collection that exceeds the size of the collection, the oldest document in the collection is deleted and the new document is inserted. Capped collections work great for objects that have a high rate of insertion, retrieval, and deletion.

The following list highlights the benefits of using capped collections:

- Capped collections guarantee that the insert order is preserved. Queries do not need to use an index to return documents in the order they were stored, eliminating the indexing overhead.
Capped collections guarantee that the insertion order is identical to the order on disk by prohibiting updates that increase the document size. This eliminates the overhead of relocating and managing the new location of documents.

Capped collections automatically remove the oldest documents in the collection. Therefore, you do not need to implement deletion in your application code.

Capped collections do impose the following restrictions:

- You cannot update documents to a larger size after they have been inserted into the capped collection. You can update them, but the data must be the same size or smaller.

- You cannot delete documents from a capped collection. The data will take up space on disk even if it is not being used. You can explicitly drop the capped collection, which effectively deletes all entries, but you also need to re-create it to use it again.

A great use of capped collections is as a rolling log of transactions in your system. You can always access the last $X$ number of log entries without needing to explicitly clean up the oldest.

**Understanding Atomic Write Operations**

Write operations are atomic at the document level in MongoDB. Thus, only one process can be updating a single document or a single collection at the same time. This means that writing to documents that are denormalized is atomic. However, writing to documents that are normalized requires separate write operations to subobjects in other collections; therefore, the write of the normalized object might not be atomic as a whole.

You need to keep atomic writes in mind when designing your documents and collections to ensure that the design fits the needs of the application. In other words, if you absolutely must write all parts of an object as a whole in an atomic manner, you need to design the object in a denormalized way.

**Considering Document Growth**

When you update a document, you must consider what effect the new data will have on document growth. MongoDB provides some padding in documents to allow for typical growth during an update operation. However, if the update causes the document to grow to a size that exceeds the allocated space on disk, MongoDB must relocate that document to a new location on the disk, incurring a performance hit on the system. Frequent document relocation also can lead to disk fragmentation issues. For example, if a document contains an array and you add enough elements to the array to exceed the space allocated, the object needs to be moved to a new location on disk.
One way to mitigate document growth is to use normalized objects for properties that can grow frequently. For example instead of using an array to store items in a Cart object, you could create a collection for CartItems; then you could store new items that get placed in the cart as new documents in the CartItems collection and reference the user’s cart item within them.

**Identifying Indexing, Sharding, and Replication Opportunities**

MongoDB provides several mechanisms to optimize performance, scale, and reliability. As you are contemplating your database design, consider the following options:

- **Indexing:** Indexes improve performance for frequent queries by building a lookup index that can be easily sorted. The _id property of a collection is automatically indexed on because looking up items by ID is common practice. However, you also need to consider other ways users access data and implement indexes that enhance those lookup methods as well.

- **Sharding:** Sharding is the process of slicing up large collections of data among multiple MongoDB servers in a cluster. Each MongoDB server is considered a shard. This provides the benefit of utilizing multiple servers to support a high number of requests to a large system. This approach provides horizontal scaling to your database. You should look at the size of your data and the amount of request that will be accessing it to determine whether to shard your collections and how much to do so.

- **Replication:** Replication is the process of duplicating data on multiple MongoDB instances in a cluster. When considering the reliability aspect of your database, you should implement replication to ensure that a backup copy of critical data is always readily available.

**Large Collections vs. Large Numbers of Collections**

Another important consideration when designing your MongoDB documents and collections is the number of collections the design will result in. Having a large number of collections doesn’t result in a significant performance hit, but having many items in the same collection does. Consider ways to break up your larger collections into more consumable chunks.

An example of this is storing a history of user transactions in the database for past purchases. You recognize that, for these completed purchases, you will never need to look them up together for multiple users. You need them available only for users to look at their own history. If you have thousands of users who have a lot of transactions, storing those histories in a separate collection for each user makes sense.
Deciding on Data Life Cycles

One of the most commonly overlooked aspects of database design is the data life cycle. How long should documents exist in a specific collection? Some collections have documents that should be kept indefinitely (for example, active user accounts). However, keep in mind that each document in the system incurs a performance hit when querying a collection. You should define a Time To Live (TTL) value for documents in each of your collections.

You can implement a TTL mechanism in MongoDB in several ways. One method is to implement code in your application to monitor and clean up old data. Another method is to utilize the MongoDB TTL setting on a collection, to define a profile in which documents are automatically deleted after a certain number of seconds or at a specific clock time.

Another method for keeping collections small when you need only the most recent documents is to implement a capped collection that automatically keeps the size of the collection small.

Considering Data Usability and Performance

The final point to consider—and even reconsider—is data usability and performance. Ultimately, these are the two most important aspects of any web solution and, consequently, the storage behind it.

Data usability describes the capability of the database to satisfy the functionality of the website. You need to make certain first that the data can be accessed so that the website functions correctly. Users will not tolerate a website that does not do what they want it to. This also includes the accuracy of the data.

Then you can consider performance. Your database must deliver the data at a reasonable rate. You can consult the previous sections when evaluating and designing the performance factors for your database.

In some more complex circumstances, you might find it necessary to evaluate data usability, then consider performance, and then look back to usability in a few cycles until you get the balance correct. Also keep in mind that, in today’s world, usability requirements can change at any time. Be sure to design your documents and collections so that they can become more scalable in the future, if necessary.

Summary

At the core of most large-scale web applications and services is a high-performance data storage solution. The back-end data store is responsible for storing everything from user account information, to shopping cart items, to blog and comment data. Good web applications require the capability to store and retrieve data with accuracy, speed, and reliability. Therefore, the data storage mechanism you choose must perform at a level to satisfy user demand.
Several data storage solutions are available to store and retrieve data your web applications need. The three most common are direct file system storage in files, relational databases, and NoSQL databases. The data store chosen for this book is MongoDB, which is a NoSQL database.

In this hour, you learned about the design considerations to review before deciding how to implement the structure of data and configuration of a MongoDB database. You also learned which design questions to ask and then how to explore the mechanisms built into MongoDB to answer those questions.

Q&A

Q. What types of distributions are available for MongoDB?
A. General distributions for MongoDB support Windows, Linux, Mac OS X, and Solaris. Enterprise subscriptions also are available for professional and commercial applications that require enterprise-level capabilities, uptime, and support. If the MongoDB data is critical to your application and you have a high amount of DB traffic, you might want to consider the paid subscription route. For information on the subscription, go to https://www.mongodb.com/products/mongodb-subscriptions.

Q. Does MongoDB have a schema?
A. Sort of. MongoDB implements dynamic schemas, enabling you to create collections without having to define the structure of the documents. This means you can store documents that do not have identical fields.

Workshop

The workshop consists of a set of questions and answers designed to solidify your understanding of the material covered in this hour. Try answering the questions before looking at the answers.

Quiz

1. What is the difference between normalized and denormalized documents?
2. True or false: JavaScript is a supported data type in a MongoDB document.
3. What is the purpose of a capped collection?

Quiz Answers

1. Denormalized documents have subdocuments within them, whereas subdocuments of normalized documents are stored in a separate collection.
2. True.
3. A capped collection enables you to limit the total size or number of documents that can be stored in a collection, keeping only the most recent.

**Exercises**

1. Go to the MongoDB documentation website and browse the FAQ page. This page answers several questions on a variety of topics that can give you a good jump-start. You can find the FAQ page at http://docs.mongodb.org/manual/faq/.
This page intentionally left blank
This page intentionally left blank
**SYMBOLS**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!==</code></td>
<td>(both value and type are not equal) operator</td>
<td>42</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>(is not equal) operator</td>
<td>42</td>
</tr>
<tr>
<td><code>!</code></td>
<td>(not) operator</td>
<td>42</td>
</tr>
<tr>
<td><code>$add</code></td>
<td>operator</td>
<td>175</td>
</tr>
<tr>
<td><code>$addToSet</code></td>
<td>operator</td>
<td>147, 173</td>
</tr>
<tr>
<td><code>$all</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$and</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$avg</code></td>
<td>operator</td>
<td>173</td>
</tr>
<tr>
<td><code>$bit</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>$concat</code></td>
<td>operator</td>
<td>175</td>
</tr>
<tr>
<td><code>$divide</code></td>
<td>operator</td>
<td>175</td>
</tr>
<tr>
<td><code>$each</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>$elemMatch</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$exists</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$first</code></td>
<td>operator</td>
<td>173</td>
</tr>
<tr>
<td><code>$group</code></td>
<td>operator</td>
<td>172, 287</td>
</tr>
<tr>
<td><code>$gte</code></td>
<td>operator</td>
<td>172</td>
</tr>
<tr>
<td><code>$gt</code></td>
<td>operator</td>
<td>172</td>
</tr>
<tr>
<td><code>$inc</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>$in</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>$limit</code></td>
<td>operator</td>
<td>172, 287</td>
</tr>
<tr>
<td><code>$lte</code></td>
<td>operator</td>
<td>172</td>
</tr>
<tr>
<td><code>$lt</code></td>
<td>operator</td>
<td>172</td>
</tr>
<tr>
<td><code>$match</code></td>
<td>operator</td>
<td>172</td>
</tr>
<tr>
<td><code>$max</code></td>
<td>operator</td>
<td>173</td>
</tr>
<tr>
<td><code>$min</code></td>
<td>operator</td>
<td>173</td>
</tr>
<tr>
<td><code>$mod</code></td>
<td>operator</td>
<td>110, 175</td>
</tr>
<tr>
<td><code>$multiply</code></td>
<td>operator</td>
<td>175</td>
</tr>
<tr>
<td><code>$ne</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$nin</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$nor</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$not</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$operator</code></td>
<td></td>
<td>147</td>
</tr>
<tr>
<td><code>$or</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$pop</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>$project</code></td>
<td>operator</td>
<td>172</td>
</tr>
<tr>
<td><code>$pullAll</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>$pull</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>$push</code></td>
<td>operator</td>
<td>147, 173</td>
</tr>
<tr>
<td><code>$regex</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$rename</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>$setOnInsert</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>$set</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>$size</code></td>
<td>operator</td>
<td>110</td>
</tr>
<tr>
<td><code>$skip</code></td>
<td>operator</td>
<td>172</td>
</tr>
<tr>
<td><code>$slice</code></td>
<td>operator</td>
<td>147</td>
</tr>
<tr>
<td><code>%</code></td>
<td>(modulous) operator</td>
<td>40</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>(and) operator</td>
<td>42</td>
</tr>
<tr>
<td><code>*</code></td>
<td>(multiplication) operator</td>
<td>40</td>
</tr>
<tr>
<td><code>+</code></td>
<td>(addition) operator</td>
<td>40</td>
</tr>
<tr>
<td><code>++</code></td>
<td>(increment) operator</td>
<td>40</td>
</tr>
<tr>
<td><code>--</code></td>
<td>(decrement) operator</td>
<td>40</td>
</tr>
<tr>
<td><code>-</code></td>
<td>(subtraction) operator</td>
<td>40</td>
</tr>
<tr>
<td><code>/</code></td>
<td>(brackets)</td>
<td>45, 49</td>
</tr>
<tr>
<td><code>//</code></td>
<td>(division) operator</td>
<td>40</td>
</tr>
<tr>
<td><code>()</code></td>
<td>(parentheses)</td>
<td>50</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>(is less than) operator</td>
<td>42</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>(is less than or equal to) operator</td>
<td>42</td>
</tr>
<tr>
<td><code>===</code></td>
<td>(both value and type are equal) operator</td>
<td>42</td>
</tr>
</tbody>
</table>
accessing

access control, configuring, 78-82
documents, counting, 125-127
GridFS Stores, 484
  files, 492-493
  Node.js applications, 498, 500-501
  Python, 494, 496-497
HTTP interfaces, 26
Java applications, 209
  applying aggregation, 225-228
  finding distinct field values, 218-221
  grouping results, 221-225
  limiting result sets, 209-218
MongoDB shell clients, 27-31
MongoGridFS objects in PHP, 490
Node.js applications, 391
  applying aggregation, 406-409
  limiting result sets, 391-399
objects, 12
PHP applications, 273
  applying aggregation, 287-290
  grouping results, 283-287
  limiting result sets, 273-281
Python application, 331
  applying aggregation, 344-347
  grouping results, 341
  limiting result sets, 331-338
accounts
  authentication, starting, 79
database administrator, formatting, 79
users
  administrator, formatting, 78
  formatting, 72
  managing, 70-78
accuracy, 7, 17
ACID (Atomic, Consistency, Isolation, Durability), 7
adding
documents
  collections, 149-151, 232
  Java applications, 231-235
  Node.js applications, 411-416
  PHP applications, 293-297
Python application, 349-353
error handling, 65-67
files, GridFS Stores, 485, 491, 494, 498
indexes, 438-440
items to arrays, 63
objects, 12
shards to clusters, 473
addUser() method, 70, 87, 187, 370
add_user() method, 313
admin database
  access control, configuring, 78-82
  overview of, 69
  user accounts, managing, 70-78
aggregate() method, 89, 172, 188, 225, 254, 314, 345, 371
aggregation
  applying, 171-178
  Java, 225-228
  Node.js applications, 406-409
operators
  expression, 173-175
  framework, 174-172
PHP applications, 287-290
pipelines, 176
Python application, 344-347
analyzing queries, 449-451
and (&&) operator, 42
anonymous functions, 51
append() method, 192
applications
  Java. See Java applications
  Node.js. See Node.js applications
PHP. See PHP applications
Python. See Python applications
applied
aggregation, 171-178
Java, 225-228
Node.js applications, 406-409
PHP applications, 287-290
pipelines, 176
Python application, 344-347
anonymous functions, 51
arrays, 60-65
if statements, 43-44
indexes, 438-443
replication, 459-467
results, mapReduce() method, 178-183
arbiter servers, 460
arithmetic operators, 40-41
Array objects, PHP applications, 257
arrays, 39
applying, 60-65
combining, 62
fields
contents, 118
searching documents based on, 118
items
adding/deleting, 63
searching, 63
iterating through, 62
manipulating, 61
strings
converting, 62
splitting, 58
values, searching documents based on, 117
assigning
roles, 71
values to variables, 38
assignment operators, 41
Atomic, Consistency, Isolation, Durability. See ACID
atomic write operations, 15
authenticate() method, 187, 253, 313, 370
authentication, starting, 79
auth() method, 87
auth setting, 24
Booleans, 39
both value and type are equal
(==) operator, 42
both value and type are not equal
(!==) operator, 42
brackets (/), 45, 49
BSON (binary JSON), 9
C
callback functions, 368
capped collections, formatting, 14-15, 436-437
changeUserPassword() method, 87
characters, null, 9
clients, shells
accessing MongoDB from, 27-31
scripting, 33-34
cloneCollection() method, 87
clonedatabase() method, 87
close() method, 186, 191, 252, 312, 368
clusterAdmin role, 71
privileges, 91
clusters, sharding
adding, 473
deploying, 472
formatting, 475-479
code property, 145
collection_names() method, 313
Collection object, 89
Node.js applications, 370-371
Python application, 313

How can we make this index more useful? Email us at indexes@samspublishing.com
collections, 9

capped, formatting, 14-15, 436-437
databases, managing, 433-437
deleting, 98-100
design, 16
documents. See also documents
    adding, 149-151, 232
    configuring write concerns, 143-144
database update
    operators, 146-147
deleting, 161-163, 236
error handling, 144
manipulating, 143
paging, 136
PHP applications, 294
retrieving, 112-116
saving, 155-158, 239
status of database write requests, 145
updating, 151-155, 243
upserting, 158-160, 246
formatting, 96-98
lists, viewing, 96
managing, 96
reindexing, 441-443
renaming, 435-436
sharding, enabling, 474
statistics, viewing, 443-444
users, counting documents, 126
collections() method, 370
column store databases, 7
combining
    arrays, 62
    strings, 58
command-line parameters, 22-23
commands
    <database>, 87
    getLastError, 144-145
    mongoFiles, 482-483
    parameters, 30-31
    shells, 28, 32-33
top, 451-453
    use <new_database_name>, 92
comparison operators, 42
compound indexes, 439
don parameter, 168
config servers, 470, 472
configuring
    access control, 78-82
databases, 22
    error handling, 144
    MongoDB, 23-26
    PHP application write concerns, 257
    servers, 461
    sharding tag ranges, 475
    write concerns, 143-144
connect() method, 186, 252, 368
Connection objects, overview of, 86
connectionId property, 145
connections
    databases, configuring error handling, 144
    write concerns, configuring, 143-144
consoles, starting shells, 28
constructors, shells, 29
contents, searching, 118
converting
    arrays into strings, 62
    records, 9
copy() method, 191
copyDatabase() method, 87
copying databases, 434-435
copyTo() method, 89
count() method, 89, 108, 126,
    188, 202, 254, 256, 265, 314,
    316, 324, 371, 373, 383
counting documents, 125-127
    Java applications, 201-203
    Node.js applications, 383-385
    PHP applications, 265-267
    Python application, 324-326
countWords() method, 202
create_collection() method, 313
createCollection() method, 87, 97,
    187, 253, 370
createIndex() method, 89
current() method, 256
Cursor objects, 107-108
    documents, counting, 126
    Node.js applications, 373
    Python applications, 315,
        324, 327, 332
    results
        limiting, 130-138
        sorting, 128-130
customizing objects, defining, 54-55
data life cycles, 17
data types
    JavaScript, 38-39
    MongoDB, 10-11
<database> command, 87
Database object, 86-87
    Node.js applications, 369-370
    Python applications, 313
database_names() method, 312
databases
    access control, implementing, 80
    admin, overview of, 69
    administrator accounts, formatting, 79
    backing up, 454-455
    Collection object, 89
    collections, managing, 433-437
    column store, 7
    configuring, 22
    Connection objects, overview of, 86
    connections, configuring error handling, 144
    copying, 434-435
    deleting, 93-94
    document store, 6
    formatting, 92-93
    graph store, 7
    indexes, 438-443
    key-value, 6
    lists, viewing, 91
    managing, 91
    optimizing, 443-453
    repairing, 453-454
    modifying, 92
    profiling, 446-448
    queries, evaluating, 449-451
    results
        grouping, 167-171
        limiting, 130-138
        sorting, 128-130
    roles, assigning, 71
    selecting, 7-8
    sharding, enabling, 474
    shells, managing, 433
    statistics, viewing, 94-96, 443-444
    testing, 31
    top command, 451-453
    update operators, 146-147
    users, listing, 74
    validating, 444-446
    write concerns, configuring, 143-144
dataset examples, implementing, 100-103
dataSize() method, 89
DB object
    Java applications, 187
    PHP applications, 253
db() method, 368
dbAdminAnyDatabase role, 71
dbAdmin role, 71
DBCollection object
    Java applications, 188
    PHP applications, 279
DBCursor object, Java
    applications, 189-191, 202, 204, 210, 213
DBObject objects, 191-194
declaring variables, 38
decrement (--) operator, 40
defining
    documents, 13
    functions, 49
    objects, customizing, 54-55
    variables, 37-38
deleting
    collections, 98-100
    databases, 93-94
documents
    collections, 236
    Java applications, 236-238
    Node.js applications, 416-419
    PHP applications, 297-299
    Python application, 353-355
files, GridFS Stores, 486, 491, 495, 499
indexes, 441
items from arrays, 63
objects, 12
users, 77
denormalizing data, 13-14
deploying
    replica sets, 462-463
    sharding clusters, 472
design, 6, 16
diagnostics, managing databases, 443-453
Dictionary objects, PHP applications, 317

displayGroup() method, 284
displayWords() method, 131
distinct field values
  Node.js applications, 400-402
  PHP applications, 281-283
  Python applications, 339-341
  searching, 138-140, 218-221
distinct() method, 138, 188, 254, 314, 316, 339, 371
distinctField() method, 89
division (/) operator, 40
document store databases, 6
documents
  collections, 9. See also
  collections
    adding, 149-151, 232
    configuring write concerns, 143-144
  database update
    operators, 146-147
    deleting, 161-163, 236
    error handling, 144
    retrieving, 112-116
  saving, 155-158, 239
  status of database write requests, 145
  updating, 151-155, 243
  upserting, 158-160, 246
counting, 125-127
cursor objects, 107-108
distinct field values,
  retrieving, 139
embedding, denormalizing
  data, 13-14

Java applications
  adding, 231-235
  counting, 201-203
  deleting, 236-238
  saving, 239-241
  searching, 194-201
  sorting results, 203-206
  updating, 241-245
  upserting, 245-249
manipulating, 143
Node.js applications, 411
  adding, 411-416
  counting, 383-385
  deleting, 416-419
  grouping results, 402-406
  objects used as, 374
  paging, 397
  retrieving, 377-383
  saving, 419-423
  searching distinct field values, 400-402
  sorting results, 385-388
  updating, 423-427
  upserting, 427-431
overview of, 9-10
parameters, PHP Arrays
  objects as, 257
PHP applications, 293
  adding, 293-297
  counting, 265-267
  deleting, 297-299
  reviewing, 260-262
  saving, 299-302
  searching, 259-265
  sorting results, 267-270
updating, 302-305
upserting, 305-308
Python application, 349
  adding, 349-353
  counting, 324-326
  deleting, 353-355
  saving, 355-357
  searching, 318-324
  sorting results, 326-328
  updating, 358-361
  upserting, 361-364
references, normalizing data,
  12-13
results
  limiting, 130-138
  sorting, 128-130
shells, searching in, 112-116
sizing, 10
specific, retrieving (using
  PHP), 262-265
specific sets of, searching,
  117-122
updating, 15
values, searching, 118
do/while loops, 45
drivers
  Java applications, 185-194
  Node.js applications, 367-377
  PHP applications, 251-259
drop() method, 89, 188, 254,
  314, 371
drop_collection() method, 313
drop_database() method, 312
drop_index() method, 314
dropCollection() method, 370
dropDatabase() method, 87, 93,
  186-187
dropDups property, 440
dropIndex() method, 89, 188, 254
embedding documents, denormalizing data, 13-14
enabling sharding
  collections, 474
  databases, 474
engines, starting/stopping, 22
ensureIndex() method, 89, 188, 254, 314
err property, 145
errors
  handling
    adding, 65-67
    document collections, 144
    throwing, 66
escape codes, string objects, 56
  --eval command-line option, 31-32
eval() method, 87
evaluating
  queries, 449-451
  shells, expressions, 31-32
example datasets, implementing, 100-103
executing
  shell scripting, 32
  variables, 38
exit command, 28
expression operators, aggregation, 173-175
expressions, evaluating shells, 31-32
find operations
  PHP applications, 283-287
find() method, 89, 107, 112, 126, 188, 254, 259, 314, 371, 377
  fields, limiting, 132
find_and_modify() method, 314
find_one() method, 314
findAndModify() method, 89, 188, 254, 371
finding. See searching
findOne() method, 89, 188, 194, 254, 259, 371
findone() method, 112
results, grouping, 167-171, 221-225
for loops, 45-46
for/in loops, 46-47
forEach() method, 108
formatting. See also design
  capped collections, 14-15, 436-437
  collections, 96-98
  config servers, instances, 472
database administrator accounts, 79
databases, 92-93
database administrator accounts, 79
example datasets, 101
functions, 49-52
replica sets, 463-467
sharding clusters, 475-479
users
  accounts, 72
  administrator accounts, 78
frameworks, aggregation operators, 174-172
fromdb parameter, 434
fromhost parameter, 434
functions
 anonymous, applying, 51
callback, 368
defining, 49
formatting, 49-52
greeting(), 50
print(), 32
values, returning, 50
variables, passing, 50

G

generating new data results, 178-183
geospatial indexes, 439
getCollection() method, 87, 187
getCollectionNames() method, 96
getConnections() method, 252
getDatabaseNames() method, 186
getDB() method, 86, 186
getIndexes() method, 89
getLastError command, 144-145
getAddress() method, 187
getMongo() method, 87
getName() method, 87
getNext() method, 256
getReadPrefMode() method, 86
getReadPrefTagSet() method, 86
getSiblingDB() method, 87
getStats() method, 188
graph store databases, 7
greeting() function, 50
GridFS Stores
 files
 adding, 485, 491, 494, 498
deleting, 486, 491, 495, 499
manipulating, 492-493
retrieving, 486, 491, 495, 499
implementing, 481
Java, 484-489
Node.js applications
 accessing, 498, 500-501
implementing, 497-501
overview of, 481-482
PHP, 489-493
Python
 accessing, 496-497
implementing, 494-497
shells, implementing, 482-484

H
 handling errors, 65-67
hashed indexes, 439
hasNext() method, 108, 191, 256
Hello World, 49
help <option> command, 28
help() method, 87
high availability, replication, 460
hint() method, 108
horizontal scaling, 6
hostInfo() method, 87
HTTP interfaces, accessing, 26

I
 if statements, applying, 43-44
implementing, 6
 access control, 80
example datasets, 100-103
GridFS Stores, 481
 Java, 484-489
 Node.js applications, 497-501
 PHP, 489-493
 Python, 494-497
shells, 482-484

in Java applications, 185.
See also Java applications
looping, 44-49
replication, 459
sharding, 459, 468-479
strategies, 7

growth, documents, 15. See also updating
switch statements, 44
upsert, 158
increment (++) operator, 40
indexes, 16
adding, 438-440
collections, reindexing, 441-443
deleting, 441
indexOf() method, 58
initial parameter, 168
insert() method, 89, 158, 188, 231, 254, 314, 371
inserting, 149. See also adding
installing MongoDB, 22
instances, formatting config servers, 472
interfaces
HTTP, 26
REST, 26
interrupting loops, 47
is equal to (==) operator, 42
is greater than (> ) operator, 42
is greater than or equal to (>=) operator, 42
is less than (<) operator, 42
is less than or equal to (<=) operator, 42
is not equal (!=) operator, 42
isAuthenticated() method, 187
isCapped() method, 89
items, arrays
adding/deleting, 63
searching, 63
iterating through arrays, 62
iterator() method, 191
files, specifying, 32-33
functions, formatting, 49-52
looping, implementing, 44-49
operators, 40-44. See also operators
shells, 30, 40
strings, manipulating, 56-60
variables
defining, 37-38
scope, 52-53
journal setting, 24
jsMode option, 179
key-value databases, 6
keyf parameter, 168
keys
parameters, 168, 434
sharding, selecting, 470-471
values
grouping objects, 169, 284
keywords
function, 49
return, 50
var, 38
life cycles, data, 17
life cycles, data, 513
J
Java applications, 185
BasicDBObject object, 191-194
data access, 209
applying aggregation, 225-228
finding distinct field values, 218-221
grouping results, 221-225
limiting result sets, 209-218
DB object, 187
DBCollection object, 188
DBCursor object, 202, 204, 210
DBObject objects, 191-194
documents
adding, 231-235
counting, 201-203
deleting, 236-238
saving, 239-241
searching, 194-201
sorting results, 203-206
updating, 241-245
upserting, 245-249
driver objects, 185-194
GridFS Stores, implementing, 484-489
MongoClient object, 186
results, paging, 215
JavaScript
arrays, applying, 60-65
data types, 38-39
error handling, adding, 65-67
L
lastOp property, 145
life cycles, data, 17
limiting
  fields, 132, 212
  Node.js applications, 394
  PHP applications, 276
  Python applications, 334
  result sets, 130-138, 209-218
  results
    Node.js applications, 391-399
    PHP applications, 273-281
    Python applications, 331-338
limit option, 179
limitResults() method, 210
listCollections() method, 253
listDBs() method, 252
listing files, GridFS Stores
  Java, 485
  Node.js applications, 498
  PHP, 490-491
  Python, 494
lists
  collections, viewing, 96
databases, viewing, 91
  users, 74
literals, objects, 39
load() method, 32
log [name] : command, 28
logappend setting, 24
logout() method, 87
logpath setting, 24
lookups, denormalized documents, 13
looping
  do/while loops, 45
  for loops, 45-46
  for/in loops, 46-47
  implementing, 44-49
  interrupting, 47
  variables, 38
  while loops, 45
M
main() method, 202
managing
  access control, configuring, 78-82
  collections, 96
  configuration settings, 23
databases, 91
  collections, 433-437
  indexes, 438-443
  optimizing, 443-453
  repairing, 453-454
  shells, 433
  user accounts, 70-78
manipulating. See also modifying
  arrays, 61
documents, 143
GridFS Stores
  files, 492-493
  Node.js applications, 500-501
  Python, 496-497
requests, applying
  aggregation, 171-178
  results, 181
  strings, 56-60
map() method, 108, 131
mapReduce() method, 89, 188
  results, applying, 178-183
max() method, 108
maxConns setting, 24
methods
  addUser(), 313
  addUser(), 70, 87, 187, 370
  aggregate(), 89, 172, 188, 225, 254, 314, 345, 371
  append(), 192
  auth(), 87
  authenticate(), 187, 253, 313, 370
  batch_size(), 316
  batchInsert(), 254, 294
  batchSize(), 108, 188, 256, 373
  changeUserPassword(), 87
  cloneCollection(), 87
  cloneDatabase(), 87
  close(), 186, 191, 252, 312, 368
  Collection objects, 89
  collection_names(), 313
  collections(), 370
  connect(), 186, 252, 368
  Connection objects, 86
  copy(), 191
  copyDatabase(), 87
  copyTo(), 89
count(), 89, 108, 126, 188, 202, 254, 256, 265, 314, 316, 324, 371, 373, 383
countWords(), 202
createCollection(), 313
createIndex(), 89
current(), 256
cursor object, 108
database objects, 87
database_names(), 312
dataSize(), 89
db(), 368
DBCollection objects, 188
DBCursor objects, 191
displayGroup(), 284
displayWords(), 131
distinct(), 138, 188, 254, 314, 316, 339, 371
distinctField(), 89
drop(), 89, 188, 254, 314, 371
drop_collection(), 313
drop_database(), 312
drop_index(), 314
dropCollection(), 370
dropDatabase(), 87, 93, 186-187
dropIndex(), 89, 188, 254
ensureIndex(), 89, 188, 254, 314
eval(), 87
find(), 89, 107, 112, 126, 188, 254, 259, 314, 371, 377
find_and_modify(), 314
find_one(), 314
findAndModify(), 89, 188, 254, 371
findOne(), 89, 188, 194, 254, 259, 371
findOne(), 112
forEach(), 108
getCollection(), 87, 187
getCollectionNames(), 96
getConnections(), 252
getDatabaseNames(), 186
db(), 86, 186
dbIndex(), 89
getLastError(), 187
getMongo(), 87
get_name(), 87
gnext(), 256
getReadPrefMode(), 86
getReadPrefTagSet(), 86
gSiblingDB(), 87
getStats(), 188
group(), 89, 168, 188, 254, 314, 371
hasNext(), 108, 191, 256
help(), 87
hint(), 108
hostInfo(), 87
indexOf(), 58
insert(), 89, 158, 188, 231, 254, 314, 371
isAuthenticated(), 187
iscapped(), 89
iterator(), 191
limit(), 108, 130, 136, 191, 210, 216, 256, 274, 316, 332, 373
limitResults(), 210
listCollections(), 253
listDBs(), 252
load(), 32
logout(), 87
main(), 202
map(), 108, 131
mapReduce(), 89, 178-183, 188
max(), 108
min(), 108
MongoClient object, 186, 252
native, shells, 29
newMongo(), 86
next(), 108, 115, 256
objsLeftInBatch(), 108
open(), 368
partitions, selecting, 471-472
print(), 40
printjson(), 40
push(), 60
read_preference(), 312-314
readPref(), 108
reindex(), 89
remove(), 89, 161, 188, 236, 254, 297, 314, 371
remove_user(), 313
removeUser(), 87, 187, 370
rename(), 314, 371
renameCollection(), 89
repairDatabase(), 87
resetDoc(), 242
runCommand(), 87, 144
save(), 89, 155, 188, 239, 254, 299, 314, 371
selectCollection(), 252-253
selectDB(), 252
serverStatus(), 87
setReadPreference(),
   186-188, 252-254, 313
setReadPrefMode(), 86
setSlaveOk(), 86
setWriteConcern(), 186-188
showWord(), 300
shutdownServer(), 87
size(), 108, 191
skip(), 108, 136, 191, 216,
   256, 316, 373
snapshot(), 108
sort(), 108, 128, 191, 204,
   256, 267, 316, 326, 373,
   386
split(), 58
stats(), 89, 95
storageSize(), 89
String object, manipulating,
   56
toArray(), 108, 191, 373
totalIndexSize(), 89
totalSize(), 89
update(), 89, 188, 242, 254,
   302, 305, 314, 371
validate(), 92
version(), 87
write_concern(), 312-314
min() method, 108
models, planning, 11-17
modifying
databases, 92
objects, 12
modulus (%) operator, 40
MongoClient object
   Java applications, 186
   Node.js applications, 368
   PHP applications, 252
   Python applications, 312
MongoCollection object, PHP
   applications, 253-254
MongoCursor object, PHP
   applications, 256, 274-276
MongoDB. See also databases
   backing up, 454-455
   configuring, 23-26
   data types, 10-11
   HTTP interfaces,
      accessing, 26
   installing, 22
   Java applications. See Java
      applications
   overview of, 8-10
   shells
      accessing clients, 27-31
      scripting, 31-34
   starting, 22
   stopping, 25
MongoDB object, PHP
   applications, 253
mongofiles command, 482-483
MongoGridFS objects in PHP,
   accessing, 490
multikey indexes, 439
multiple documents, searching,
   115. See also documents
multiplication (*) operator, 40

N
n property, 145
name property, 440
naming
   fields, 9
   variables, 38
native methods, shells, 29
new Mongo () method, 86
next() method, 108, 115, 256
noauth setting, 24
Node.js applications, 367
   Collection object, 370-371
   Cursor objects, 373
data access, 391
   applying aggregation,
      406-409
   grouping results, 402-406
   limiting result sets,
      391-399
Database object, 369-370
documents, 411
   adding, 411-416
   counting, 383-385
   deleting, 416-419
   paging, 397
   retrieving, 377-383
   saving, 419-423
   searching distinct field
      values, 400-402
operators

517

operators

$, 148
$add, 175
$addToSet, 148, 173
$all, 110
$and, 110
$avg, 173
$bit, 148
$concat, 175
$divide, 175
$each, 148
$elemMatch, 110
$exists, 110
$first, 173
$group, 172, 287
$gt, 110
$gte, 110
$in, 110
$inc, 147
$last, 173
$limit, 172, 287
$lit, 110
$lte, 110
$match, 172
$max, 173
$min, 173
$mod, 110, 175
$multiply, 175
$ne, 110
$nin, 110
$nor, 110
$not, 110
$or, 110
$pop, 148
$project, 172
$pull, 148

operators

517

operators

$, 148
$add, 175
$addToSet, 148, 173
$all, 110
$and, 110
$avg, 173
$bit, 148
$concat, 175
$divide, 175
$each, 148
$elemMatch, 110
$exists, 110
$first, 173
$group, 172, 287
$gt, 110
$gte, 110
$in, 110
$inc, 147
$last, 173
$limit, 172, 287
$lit, 110
$lte, 110
$match, 172
$max, 173
$min, 173
$mod, 110, 175
$multiply, 175
$ne, 110
$nin, 110
$nor, 110
$not, 110
$or, 110
$pop, 148
$project, 172
$pull, 148

operators

517

operators

$, 148
$add, 175
$addToSet, 148, 173
$all, 110
$and, 110
$avg, 173
$bit, 148
$concat, 175
$divide, 175
$each, 148
$elemMatch, 110
$exists, 110
$first, 173
$group, 172, 287
$gt, 110
$gte, 110
$in, 110
$inc, 147
$last, 173
$limit, 172, 287
$lit, 110
$lte, 110
$match, 172
$max, 173
$min, 173
$mod, 110, 175
$multiply, 175
$ne, 110
$nin, 110
$nor, 110
$not, 110
$or, 110
$pop, 148
$project, 172
$pull, 148

operators

517

operators

$, 148
$add, 175
$addToSet, 148, 173
$all, 110
$and, 110
$avg, 173
$bit, 148
$concat, 175
$divide, 175
$each, 148
$elemMatch, 110
$exists, 110
$first, 173
$group, 172, 287
$gt, 110
$gte, 110
$in, 110
$inc, 147
$last, 173
$limit, 172, 287
$lit, 110
$lte, 110
$match, 172
$max, 173
$min, 173
$mod, 110, 175
$multiply, 175
$ne, 110
$nin, 110
$nor, 110
$not, 110
$or, 110
$pop, 148
$project, 172
$pull, 148

How can we make this index more useful? Email us at indexes@samspublishing.com
operators

- $pullAll, 148
- $push, 148, 173
- $regex, 110
- $rename, 147
- $set, 148
- $setOnInsert, 147
- $size, 110
- $skip, 172
- $slice, 148
- $sort, 148, 172
- $strcasecmp, 175
- $substr, 175
- $subtract, 175
- $sum, 173
- $toLower, 175
- $toUpper, 175
- $type, 110
- $unset, 148
- $unwind, 172
- addition (+), 40
- aggregation
  - expression, 173-175
  - framework, 174-172
- and (&&), 42
- arithmetic, 40-41
- assignment, 41
- both value and type are equal (===), 42
- both value and type are not equal (!===), 42
- comparison, 42
- decrement (--), 40
- division (/), 40
- fields: value, 110
- increment (++), 40
- is equal to (==), 42
- is greater than (>), 42
- is greater than or equal to (>=), 42
- is less than (<), 42
- is less than or equal to (<=), 42
- is not equal (!=), 42
- JavaScript, 40-44
- modulus (%), 40
- multiplication (*), 40
- not (!), 42
- or (||), 42
- query, 109-110
- subtraction (-), 40
- update, 146-147
- optimizing databases, 443-453
- options, --eval command-line, 31-32
- orders, sorting, 128-130
- or (||) operator, 42
- out option, 179
- outputting data in shells, 40

parameters

- command-line, 22, 23
- commands, 30-31
- documents, PHP Arrays
  - objects as, 257
- fields, 213
- group() method, 168
- Node.js applications, objects
  - used as, 374
- operator, 287
- projection, 133
- query, 109, 139, 161
- parentheses (()), 50
- partitions, selecting methods, 471-472
- passing variables to functions, 50
- patterns, prototyping objects, 55
- performance, 17
  - databases, managing, 443-453
  - models, planning, 11-17
  - replication, 460. See also replication
- PHP applications, 251
  - Array objects, 257
  - data access, 273
    - applying aggregation, 287-290
    - grouping results, 283-287
    - limiting result sets, 273-281
    - searching distinct field values, 281-283
- DB object, 253
- DBCollection object, 279
- Dictionary objects, 317
- documents, 293
repairDatabase() method

adding, 293-297
counting, 265-267
deleting, 297-299
reviewing, 260-262
saving, 299-302
searching, 259-265
sorting results, 267-270
updating, 302-305
upserting, 305-308
driver objects, 251-259
fields, limiting, 276
GridFS Stores, implementing, 489-493
MongoClient object, 252
MongoCollection object, 253-254
MongoCursor object, 256, 274-276
MongoDB object, 253
results, paging, 278
write concerns, configuring, 257
pipelines, applying aggregation, 176
planning models, 11-17
port setting, 24
primary servers, 460
print() function, 32
print() method, 40
printjson() method, 40
privileges, clusterAdmin role, 91
profiling databases, 446-448
projection parameter, 133
prototyping object patterns, 55
push() method, 60

Python applications, 311
Collection object, 313
Cursor objects, 315, 332
data access, 331
applying aggregation, 344-347
grouping results, 341
limiting result sets, 331-338
Database object, 313
distinct field values, 339-341
documents, 349
adding, 349-353
counting, 324-326
deleting, 353-355
saving, 355-357
searching, 318-324
sorting results, 326-328
updating, 358-361
upserting, 361-364
fields, limiting, 334
GridFS Stores
accessing, 496-497
implementing, 494-497
MongoClient object, 312
results, paging, 336

Q
queries
   evaluating, 449-451
routers, 469
starting, 473
query operators, 109-110
query options, 179
query parameters, 139, 161

R
RAM (random access memory), 10
random access memory. See RAM
RDBMSs (relational database management systems), 6-8
read role, 71
read_preference() method, 312-314
readAnyDatabase role, 71
readPref() method, 108
readWrite role, 71
readWriteAnyDatabase role, 71
records, converting, 9
reduce parameter, 168
references, normalizing data, 12-13
reIndex() method, 89
reindexing collections, 441-443
relational database management systems. See RDBMSs
reliability, 7
remove() method, 89, 161, 188, 236, 254, 297, 314, 371
remove_user() method, 313
removeUser() method, 87, 187, 370
removing. See deleting
rename() method, 314, 371
renameCollection() method, 89
renaming collections, 435-436
repairDatabase() method, 87
repairing databases, 453-454
replacing words in strings, 58
replica sets. See also replication
deploying, 462-463
formatting, 463-467
types of, 460
replication, 16, 459
applying, 459-467
strategies, applying, 461
requests
manipulating, applying
aggregation, 171-178
Node.js applications, 374
paging, 128
PHP applications, applying
aggregation, 287-290
status of database write,
retrieving, 145
resetDoc() method, 242
REST interfaces, 26
rest setting, 24
results, 30-31
  grouping, 167-171, 221-225
  Java applications
    paging, 215
    sorting, 203-206
  limiting
    Java applications, 209-218
    PHP applications, 273-281
  manipulating, 181
mapReduce() method,
applying, 178-183
Node.js applications
  grouping, 402-406
  limiting, 391-399
  sorting, 385-388
  paging, 136
  PHP applications
    grouping, 283-287
    paging, 278
    sorting, 267-270
Python application
  limiting, 331-338
  sorting, 326-328
  grouping, 341
  paging, 336
  sets, limiting, 130-138
  sorting, 128-130
retrieving
distinct field values, 139,
219, 282, 400
documents
  from collections, 112-116
  Node.js applications, 377-383
  Python applications, 319
using Java, 195
files, GridFS Stores, 483,
486, 491, 495, 499
specific documents (using
PHP), 262-265
status of database write
requests, 145
return keyword, 50
returning
  fields, limiting, 212
  objects, 132
  values from functions, 50
reviewing documents, PHP
applications, 260-262
roles
  assigning, 71
  clusterAdmin privileges, 91
routers
  queries, 469
  starting, 473
runCommand() method, 87, 144
S
save() method, 89, 155, 188,
239, 254, 299, 314, 371
saving
databases, 454-455
  documents
    collections, 155-158, 239
    Java applications, 239-241
    Node.js applications,
    419-423
    PHP applications, 299-302
    Python application,
    355-357
scalability, 6
scope
  options, 179
  variables, 52-53
scripting shells, 31-34
  clients, 33-34
  executing, 32
searching
  array items, 63
  contents, 118
distinct field values, 138-140, 218-221
Node.js applications, 400-402
PHP applications, 281-283
documents
in shells, 112-116
Java applications, 194-201
Node.js applications, 377-383
PHP applications, 259-265
Python application, 318-324
multiple documents, 115
results, grouping, 167-171
specific documents, Node.js applications, 380
specific sets of documents, 117-122
substrings in strings, 58
secondary servers, 460
selectCollection() method, 252-253
selectDB() method, 252
selecting
NoSQL, 7-8
partitioning methods, 471-472
RDBMS, 7-8
sharding keys, 470-471
servers
replication, 460. See also replication
sharding, types of, 469
serverStatus() method, 87
setReadPreference() method, 186-188, 252, 254, 313
setReadPrefMode() method, 86
sets, results. See also results
Java applications, 203-206
limiting, 130-138
limiting access in Java applications, 209-218
PHP applications, 273-281
setSlaveOk() method, 86
setup. See configuring
setWriteConcern() method, 186-188
shard servers, 469
sharding, 16, 459
clusters
adding, 473
deploying, 472
formatting, 475-479
collections, enabling, 474
databases, enabling, 474
implementing, 468-479
keys, selecting, 470-471
servers, types of, 469
tag ranges, configuring, 475
shells
aggregation, applying, 171-178
clients
accessing MongoDB from, 27-31
scripting, 33-34
commands, 28, 32-33
constructors, 29
databases, managing, 433
documents
adding to collections, 149-151
counting, 125-127
deleting, 161-163
limiting result sets, 130-138
saving, 155-158
searching in, 112-116
sorting results, 128-130
upserting, 158-160
expressions, evaluating, 31-32
GridFS Stores, implementing, 482-484
JavaScript, 30, 40
native methods, 29
objects, grouping, 222
results
grouping, 167-171
mapReduce() method, 178-183
scripting, 31-34
starting, 28
user accounts, formatting, 72
show <option> command, 28
showWord() method, 300
shutdownServer() method, 87
single field indexes, 439
size() method, 108, 191
sizing
collection design, 16
documents, 10
Node.js application results, limiting, 392-394
results, limiting by, 210
slaveOk parameter, 434

How can we make this index more useful? Email us at indexes@samspublishing.com
snapshot() method, 108
sorting results, 128-130
    Java applications, 203-206
    Node.js applications, 385-388
    PHP applications, 267-270
    Python application, 326-328
sort option, 179
sort() method, 108, 128, 191, 204, 256, 267, 316, 326, 373, 386
sparse property, 440
specific documents
    Node.js applications, 380
    PHP, 262-265
    Python applications, 321
specific sets of documents, searching, 117-122
specifying JavaScript files, 32-33
speed, 7
split() method, 58
splitting strings into arrays, 58
starting
    authentication, 79
    MongoDB, 22
    query routers, 473
    shells, 28
statements
    if, applying, 43-44
    return, 50
    switch, implementing, 44
statistics, viewing, 94-96, 443-444
stats() method, 89, 95
status of database write requests, retrieving, 145
stopping MongoDB, 25
storage, GridFS Stores, 483
storageSize() method, 89
strategies, 7
strategies, applying replication, 461
strings
    arrays
        converting, 62
        splitting, 58
    combining, 58
    manipulating, 56-60
    objects, escape codes, 56
    substrings, searching, 58
    words, replacing, 58
subdocuments, 9, 118. See also documents
subobjects, 13. See also objects
substrings, searching strings, 58
subtraction (-) operator, 40
switch statements
    implementing, 44
syntax, objects, 53-54
totalIndexSize() method, 89
totalSize() method, 89
troubleshooting
databases
    managing, 443-453
    repairing, 453-454
    top command, 451-453
try/catch blocks, 65
TTL (Time To Live), 17
TTL property, 440
types
    data
        JavaScript, 38-39
        MongoDB, 10-11
    of indexes, 438-440
    of loops, 44-49
    of replica sets, 460
    of sharding servers, 469
unique property, 440
update operators, databases, 146-147
update() method, 89, 188, 242, 254, 302, 305, 314, 371
updateExisting property, 145
updating documents, 15
    collections, 151-155, 243
    Java applications, 241-245
    Node.js applications, 423-427
    PHP applications, 302-305
    Python applications, 358-361
upserted property, 145
upserting documents
  collections, 158-160, 246
  Java applications, 245-249
  Node.js applications, 427-431
  PHP applications, 305-308
  Python application, 361-364
usability, 10
use <database> command, 28
use <new_database_name> command, 92
user accounts
  authentication, starting, 79
  formatting, 72
  managing, 70-78
user administrator accounts, formatting, 78
userAdminAnyDatabase role, 71
userAdmin role, 71
username parameter, 434
users
  deleting, 77
  documents, counting, 126
  lists, 74

fields
  searching based on, 117
  searching distinct, 138-140, 218-221
  functions, returning, 50
  key, grouping objects, 169
  null variables, 39
  subdocuments, searching, 118
var keyword, 38
variables
  defining, 37-38
  functions, passing, 50
  scope, 52-53
verbose option, 179
verbose setting, 24
version() method, 87
viewing
  collections, 96
  databases lists, 91
  stats, 94-96
  statistics, 443-444

V
validate() method, 92
validating databases, 444-446
values
  arrays, searching documents based on, 117
  documents, searching, 118

W
waited property, 145
while loops, 45
wnote property, 145
words, replacing strings, 58
write concerns
  configuring, 143-144
    Node.js applications, 374
  PHP applications, 257
  Python applications, 317

How can we make this index more useful? Email us at indexes@samspublishing.com