Brad Dayley

Sams Teach Yourself

NoSQL with MongoDB

in 24 Hours

SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Brad Dayley

Sams **Teach Yourself**

NoSQL with MongoDB

in 24 Hours

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself NoSQL with MongoDB in 24 Hours

ISBN-13: 9780672337130

ISBN-10: 0672337134

Library of Congress Control Number: 2014942748

Printed in the United States of America

First Printing: September 2014

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Pearson cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside of the U.S., please contact international@pearsoned.com.

Acquisitions Editor
Mark Taber

Managing Editor
Kristy Hart

Project Editors
Melissa Schirmer
Elaine Wiley

Copy Editor
Krista Hansing
Editorial
Services, Inc.

Indexer
WordWise
Publishing Services

Proofreader
Kathy Ruiz

Technical Editor
Russell Kloepfer

Publishing Coordinator
Vanessa Evans

Cover Designer
Mark Shirar

Compositor
Gloria Schurick

Contents at a Glance

Table of Contents

Part III: Using MongoDB in Applications

About the Author

Brad Dayley is a senior software engineer with more than 20 years of experience developing enterprise applications. He has designed and developed large-scale business applications, including SAS applications with NoSQL database back ends and rich Internet web applications as front ends. He is the author of the *jQuery and JavaScript Phrasebook, Sams Teach Yourself jQuery and JavaScript in 24 Hours,* and *Node.js, MongoDB and AngularJS Web Development*.

Dedication

For D!

A & F

Acknowledgments

I'd like to take this page to thank all those who made this title possible. First, I thank my wonderful wife and boys for giving me the inspiration and support I need. I'd never make it far without you. Thanks to Mark Taber for getting this title rolling in the right direction, Russell Kloepfer for his technical review, and Melissa Schirmer for managing everything on the production end.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title, edition number, and author, as well as your name and contact information.

Email: feedback@samspublishing.com

Mail: Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at **informit.com/register** for convenient access to any updates, downloads, or errata that might be available for this book.

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.

This page intentionally left blank

HOUR 1

Introducing NoSQL and MongoDB

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 back-end 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:

```
{
  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 of 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 1.1 MongoDB Data Types and Corresponding ID Number

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object ID	7
Boolean	8
Date	9
Null	10
Regular expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18

Type	Number
Min key	255
Max key	127

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.

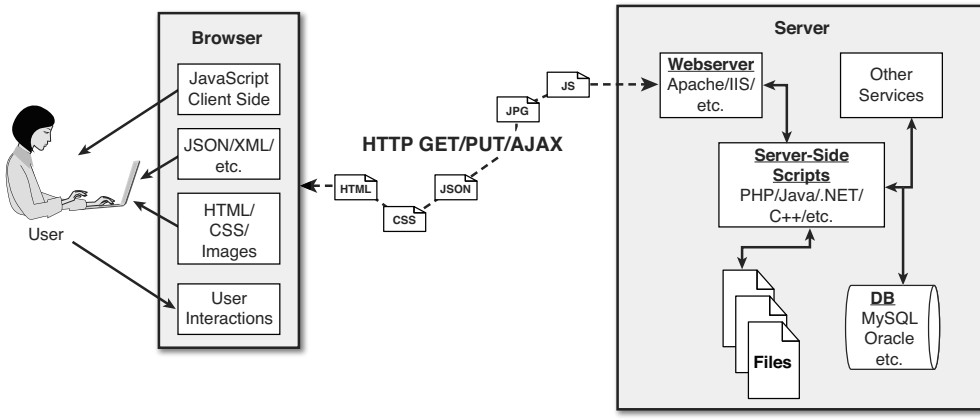

FIGURE 1.1
Defining normalized MongoDB documents by adding a reference to documents in another collection.

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.

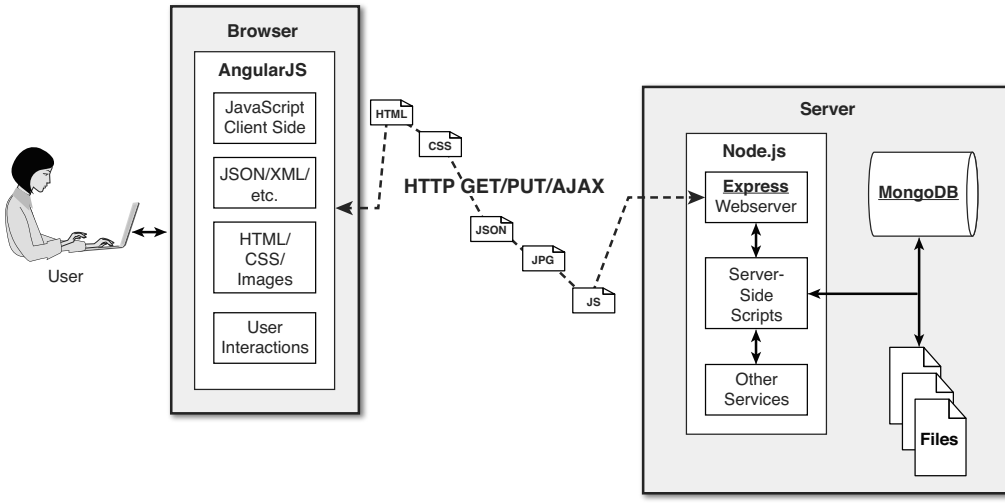

FIGURE 1.2
Defining denormalized MongoDB documents by implementing embedded objects inside a document.

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

Index

SYMBOLS

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W