Appendix P

Working with MongoDB

Preview

This appendix gives you some step-by-step illustrations of using MongoDB, a popular document database. Among the NoSQL databases currently available, MongoDB has been one of the most successful in penetrating the database market. Therefore, learning the basics of working with MongoDB can be quite useful for database professionals.

Data Files and Available Formats

File name	Format/Description
Ch14_FACT.json	JavaScript Object Notation file (also used in Chapter 14)

Data Files Available on cengagebrain.com

Note

MongoDB is a product of MongoDB, Inc. In this appendix, we use the Community Server v.3.4.6 edition, which is open source and available free of charge from MongoDB, Inc. New versions are released regularly. This version of MongoDB is available from the MongoDB website for Windows, MacOS, and Linux.

P-1 Introduction to MongoDB

The name, MongoDB, comes from the word *humongous* as its developers intended their new product to support extremely large data sets. It is designed for:

- High availability
- High scalability
- High performance

As a document database, MongoDB is schema-less and aggregate aware. Recall from Chapter 14, Big Data and NoSQL, that being schema-less means that all documents are not required to conform to the same structure, and the structure of documents does not have to be declared ahead of time. Aggregate aware means that the documents encapsulate all relevant data related to a central entity within the same document. Data is stored in documents, documents of a similar type are stored in collections, and related collections are stored in a database. Documents are formatted using BSON for storage, which is a lightweight, binary representation of JSON, but with support for added features like a greater range of data types. To the users, the documents appear as JSON files, which makes them easy to read and easy to manipulate in a variety of programming languages. Understanding the basic structure of a JSON document is essential for working with MongoDB.

P-2 JSON Documents

JavaScript Object Notation (JSON) is a data interchange format that represents data as a logical object. Objects are enclosed in curly brackets {} that contain key-value pairs. When discussing JSON documents, the key-value pairs are typically written in the format *key:value*, so we will follow that convention in the discussion of MongoDB. In different circumstances, you may see the terminology *key:value*, *tag:value*, *field:value*, *id:value*, or *property:value* used, but they all mean the same thing, so do not allow these changes in terminology to throw you off. This variation in terminology is a testament to the wide range of contexts and programming languages that have adopted the JSON format.

A single JSON object can contain many *key:value* pairs separated by commas. A simple JSON document to store data on a book might look like this:

{_id: 101, title: “Database Systems”}

This document contains two *key:value* pairs:

- *_id* is a key with 101 as the associated value
- *title* is a key with “Database Systems” as the associated value

The *value* component may have multiple values that would be appropriate for a given key. In the previous example, adding a *key:value* pair for authors could have the values

"Coronel" and "Morris." When there are multiple values for a single key, an array is used. Arrays in JSON are placed inside square brackets []. For example, the above document could be expanded to:

{_id: 101, title: "Database Systems", author: ["Coronel", "Morris"]}

When JSON documents are intended to be read by humans, they are often displayed with each *key:value* pair on a separate line to improve readability, such as:

```
{
_id: 101,
title: "Database Systems",
author: ["Coronel", "Morris"]
}
```

Objects can also have other objects embedded as a value. When a JSON document has an embedded object, the embedded object is often referred to as a **subdocument**. Consider another simple document with data about a publisher that is related to the book in the previous example.

```
{
Name: "Cengage",
Address: "500 Topbooks Avenue",
City: "Boston",
State: "MA"
}
```

The book document and the publisher document are related. Remember, document databases are aggregate aware, so documents are typically arranged around a central entity. If we are constructing documents that have a book as the central entity, then we may wish to include data about the publisher inside that document. To include the publisher information with the book, we could embed the publisher as a subdocument inside the book document, as follows:

```
{
_id: 101,
title: "Database Systems",
author: ["Coronel", "Morris"],
publisher: {
        name: "Cengage",
        street: "500 Topbooks Avenue",
        city: "Boston",
        state: "MA"
        }
}
```

subdocument
A JSON document that is embedded as an object within another JSON document.

In this case, we have avoided a situation that would have required a join in a relational environment. In a relational environment, we would have used a BOOK table and a PUBLISHER table with a 1:M relationship. In essence, we have pre-joined the

book and publisher data into a single document. Although this increases redundancy, NoSQL databases often sacrifice redundancy to improve scalability. Remember, with document databases, we are attempting to avoid the need for joins, making documents independent of each other so they can be easily scaled out to many computers in a cluster.

In addition to pre-joining data, subdocuments can be useful when a value comprises smaller meaningful components. In the previous example, the street, city, and state pairs all work together to provide an address. Therefore, an address object could be created that includes these properties:

```
{
_id: 101,
title: "Database Systems",
author: ["Coronel", "Morris"],
publisher: {
        name: "Cengage",
        address: {
                street: "500 Topbooks Avenue",
                city: "Boston",
                state: "MA"
                }
        }
}
```

This document is organized around a book. The book document has four *key:value* pairs. The third *key:value* pair (author) has an array of multiple values. The fourth pair (publisher) has an object as the value. The publisher object is composed of two *key:value* pairs, the second of which (address) is an object that is composed of three *key:value* pairs. The JSON format allows an indefinite number of objects to be embedded inside each other, and it is possible to even have arrays of objects. For example, the author array in our example could be expanded as:

```
{
_id: 101,
title: "Database Systems",
author: [
        {
        name: "Coronel",
        email: "ccoronel@mtsu.edu",
        phone: "6155551212"
        },
        {
        name: "Morris",
        email: "smorris@mtsu.edu",
        office: "301 Codd Hall"
```

```
        }
        ],
publisher: {
        name: "Cengage",
        address: {
                street: "500 Topbooks Avenue",
                city: "Boston",
                state: "MA"
                }
        }
}
```

JSON format is very flexible and very powerful. It can represent the data in many different ways so that programmers can find the representation that best suits the needs of each individual application. JSON is well-suited to schema-less data models because there is no requirement for each document in a collection to have the same *key:value* pairs. Notice, for example, that there are differences in the *key:value* pairs for the two author objects within the author array in our illustration.

As you work to get comfortable with document databases, drawing connections to the relational topics that you are already familiar with can be helpful. A collection in MongoDB can be thought of as being similar to a table in a relational database in that they both contain data about a given topic. Obviously, the data in a collection is aggregate aware so it is a very different set of data than would appear in the relational table. A document is like a row of data in a relational table. The *key:value* pairs can be thought of as being like a column and a value in the column. Because each document (row) can have different *key:value* pairs, it is like allowing each row in a table to have different columns (thus, document databases are schema-less). An array in a *key:value* pair is the document database way of handling multivalued attributes. Embedded objects in a document can correspond to composite attributes or to pre-joined tables. The use of arrays and embedded objects allow the document database to be aggregate aware and avoid using multiple tables that would require joins to combine for reporting purposes.

P-3 Creating Databases and Collections in MongoDB

Just as relational databases can be accessed through a wide range of application programming languages such as JavaScript, Java, Python, and PHP, so can MongoDB. MongoDB also provides a built-in MongoDB shell program that provides an interface for working directly with the data. Using the shell, data is queried using Mongo Query Language (MQL). MQL is based on JavaScript, and many of the programming features of JavaScript, such as IF statements, variable declaration and manipulation, and looping, are available directly in MongoDB. Although full JavaScript programming is beyond the scope of this book, it is important to consider basic data manipulation in MQL.

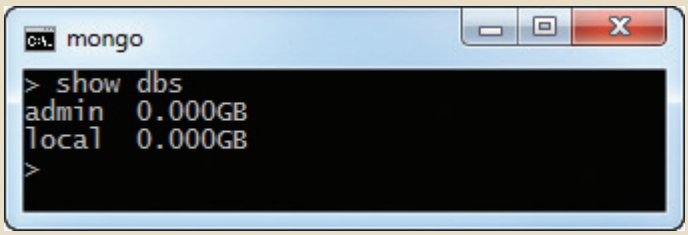
MongoDB databases comprise collections of documents. Each MongoDB server can host many databases. Within the MongoDB server (also called the host), each database is a type of object. As we know from the previous discussion of JSON format, objects can contain other objects. A database object contains collections. Collections are also

objects. Collection objects contain document objects. In addition to holding data content, an object can also have **methods**, which are programmed functions for manipulating the object. When connected to the MongoDB server, the first task is to specify with which database object you want to work. A list of the databases available on the server can be retrieved with the command:

```
show dbs
```

By default, a new installation of MongoDB includes an admin database and a local database as shown in Figure P.1. The admin database is used to record data about database administration issues such as users, roles, and privileges for the databases hosted on this server. The local database is used for storing data about the server's start-up process and the server's role in sharding operations. Recall that aggregate aware databases try to break up the data into pieces called shards. As a database grows in volume, MongoDB may adjust the way shards are distributed among the cluster nodes to balance the workload. Unless you are administering the MongoDB server, you are not likely to need to worry about the admin and local databases, as they should never be used for storing end user data.

FIGURE P.1 MONGODB DEFAULT DATABASES



All data manipulation commands in MongoDB must be directed to a particular database. Creating a new database in MongoDB is as easy as issuing the *use* command. The use command informs the server which database is to be the target of the commands that follow. If there is a database with the name specified, then that database will be used for the subsequent commands. If there is not a database with that name, then one is created automatically. For example, the following command creates a database named demo that we will use for some of the later code examples:

```
use demo
```

This command creates a database named "demo" and sets a special MongoDB variable named *db* to have the value "demo". In subsequent commands, we will use the variable *db* to specify the database that is targeted by our commands. The name of the database that is being used can be displayed using the **getName()** method on the db variable, as follows:

```
db.getName()
```

method
A programmed function within an object used to manipulate the data in that same object.

getName()
A MongoDB method for returning the name of the current database.

Note

Notice that the *show dbs* command will not include the new database in the list of available databases until the database contains at least one collection.

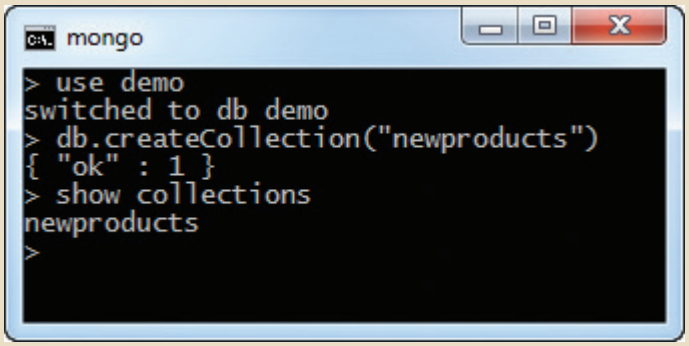
Although there are conceptual similarities between collections and relational tables, the differences become much more obvious when creating a collection. Creating a table in a relational database requires specifying a great deal of metadata. The name and data type for each attribute, any column constraints, primary key constraints, foreign key constraints, and check constraints are all specified at the time of table creation. Because document databases are schema-less with no predefined structure, none of these parameters are required. Creating a collection requires only specifying the database it belongs to and the name of the collection. Using the **createCollection()** method with the *db* variable creates a collection with the specified name. The following command creates a "newproducts" collection inside the previously defined demo database:

```
db.createCollection("newproducts")
```

To display the collections that exist in a database, the following command can be used, as shown in Figure P.2:

```
show collections
```

FIGURE P.2 MONGODB COLLECTIONS



Now that the demo database has at least one collection, demo shows as one of the available databases returned by the *show dbs* command. Although it was not necessary to specify parameters when creating the collection, there are a few options that are available. For instance, there is an autoIndexID option that acts as a surrogate key. When autoIndexID is set to true for a collection, the DBMS automatically adds a "_id" field to each document to act as a unique identifier and automatically create an index on that field to speed retrievals. AutoIndexID is set to true by default.

Using collection creation options, it is also possible to create a **capped collection** in MongoDB. A capped collection is a MongoDB collection that can only grow to a specified maximum size; then documents are automatically deleted from it. When a capped collection reaches its maximum size and new documents are added, the oldest documents are automatically deleted to make room for the new documents. Capped collections are often used for log files where only a limited number of recent actions are of interest. Capped collections can be limited by total storage size and/or by number of documents. For example, the following command creates a collection named userlog that is capped to 1,000 documents and 1 megabyte in size:

```
db.createCollection("userlog", {capped: true, size: 1048576, max: 1000})
```

createCollection()
A MongoDB method for creating a collection inside a database.

capped collection
A MongoDB collection that has a specified limit on the size and/or number of documents within the collection. Older documents are automatically deleted to make room for new documents once the limit is reached.

As you might be able to see, most MQL commands borrow their syntax from JavaScript and the JSON format.

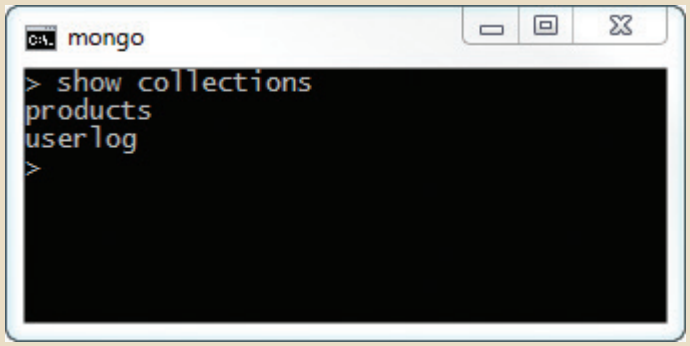
P-4 Renaming and Dropping Collections

Occasionally, it might be necessary to rename collections within MongoDB. To rename a collection you use the **renameCollection()** method of the collection, and provide the new name for the collection as a parameter. For example, the following command renames the newproducts collection to products:

```
db.newproducts.renameCollection("products")
```

Using the *show collections* command, you can see that the collection name has changed (see Figure P.3).

FIGURE P.3 RENAMING A MONGODB COLLECTION



Dropping a collection removes all documents in the collection and deletes any indexes that have been created with that collection. Dropping the collection is done with the **drop()** method of the collection. The drop() method does not require any options or parameters. It drops whichever collection that it was called with. For example, the following command drops the userlog collection that was created earlier:

```
db.userlog.drop()
```

P-5 Inserting Documents in MongoDB

renameCollection()
A MongoDB method to change the name of an existing collection within a database.

drop()
A MongoDB method to drop a collection and all of its documents from a database.

insert()
A MongoDB method to add new documents to an existing collection.

Once a collection has been created, documents can be added to it. The **insert()** method of the collection is used to add new documents to the collection. The syntax for the insert() method in MongoDB is as follows:

```
db.<collection name>.insert({document})
```

Consider the following JSON document:

```
{name: "standard desk chair",
price: 150,
brand: "CheapCo",
type: "chair"}
```

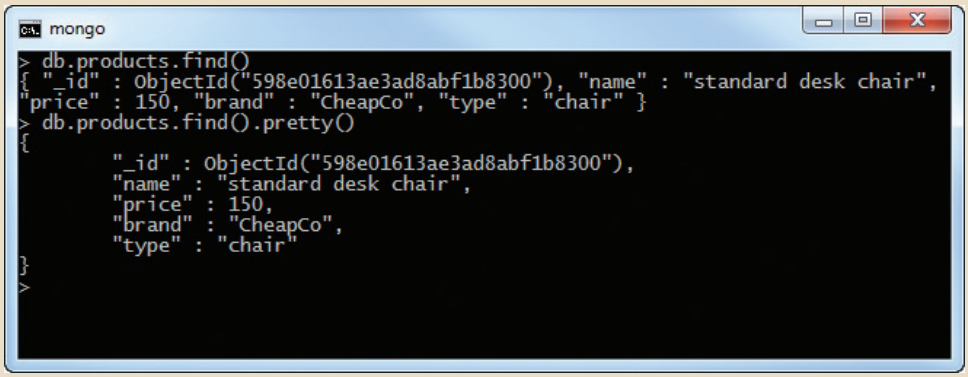
In order to insert the previous document into the products collection that we created earlier, we call the insert() method of the collection and provide the document as a parameter, as follows:

```
db.products.insert ({name: "standard desk chair,"
                     price: 150,
                     brand: "CheapCo",
                     type: "chair"})
```

Retrieving documents in MongoDB will be explored in greater detail later in this chapter; however, for now we can retrieve the documents in our collection using the **find()** method. Adding the pretty() method, discussed later, improves the readability of the returned document. The following command displays all of the documents in the product collection, as shown in Figure P.4:

```
db.products.find()
```

FIGURE P.4 RETRIEVING A MONGODB DOCUMENT



Notice that each document has a key named *_id* that contains an ObjectID. As discussed previously, the default for new collections is to have the autoIndexID property set to true so the _id column is generated automatically when documents are inserted. The _id values generated in your collections will differ from the ones shown in the figures because characteristics of the host computer become encoded as part of the _id field, but they will always be unique within your collection.

Next, let's insert another document in the same collection, but with a slightly different structure. Remember, document databases are schema-less so all of the documents in a collection are not required to have the same structure.

```
db.products.insert({name: "cushioned desk chair",
                    brand: "RoughRider",
                    type: "chair",
                    keywords: ["chair", "office", "cushioned"],
                    price: 300})
```

The insert() method can be used with JSON documents of any complexity. The previous example includes an array of values for the *keywords* key.

find()
A MongoDB method to retrieve documents from a collection.

P-6 Updating Documents in MongoDB

There are many options when updating documents in MongoDB. At the most basic level, updates can either be **operator updates** or **replacement updates**. An operator update makes changes to some of the content of a document, but leaves the remainder of the document unchanged. This is similar to an update in a relational database. A replacement update completely removes the original document and replaces it with the new document while preserving only the value of the _id primary key. Both types of updates are performed with the **update()** method of the collection. The syntax for the update() method is:

db.<collection name>.update({<query>}, {<change>}, <upsert>, <multi>)

The update() method can have four parameters. First is a *query* object that is used to determine which documents to update. This is like the WHERE clause of a SQL UPDATE command. Only documents that match the query are updated.

The second parameter is a *change* object that specifies what changes to make to the document. If a replacement update is being performed, then the change object will be the new version of the document to be stored. If an operator update is being performed, then the change object will contain the functions that specify what changes should be made to the document.

The third parameter is the *upsert* flag. The **upsert** flag is Boolean, that is, it is either true or false. By default, upsert is false. If upsert is false, then if no documents match the criteria in the query object, then no changes will be made to the collection. On the other hand, if upsert is set to true, then if no documents match the criteria in the query object, then a new document will be inserted with the data in the change object. Upsert combines an update and an insert, hence the name. If there is a match, it performs an update; if there is not a match, it performs an insert.

The fourth parameter is the *multi* flag. Like upsert, multi is Boolean and false by default. If multi is false, then only the first document found to match the query object will be updated. If multi is set to true, then all documents found to match the query object will be updated. Both the upsert and multi flags can be omitted to apply the default values of false for those flags.

operator update
In MongoDB, an update that changes one or more values inside a document while leaving the remainder of the document unchanged.

replacement update
In MongoDB, an update that completely replaces an existing document with a new document while keeping only the _id (primary key) unchanged.

update()
A MongoDB method for changing the contents of a document.

upsert
In MongoDB, a flag for use with updates that combines an insert operation and an update operation. When set to true, if no matching document is found for the criteria given, then a new document that matches the specified values will be created. If a matching document is found, then the existing document will be updated with the specified values.

P-6a Replacement Updates

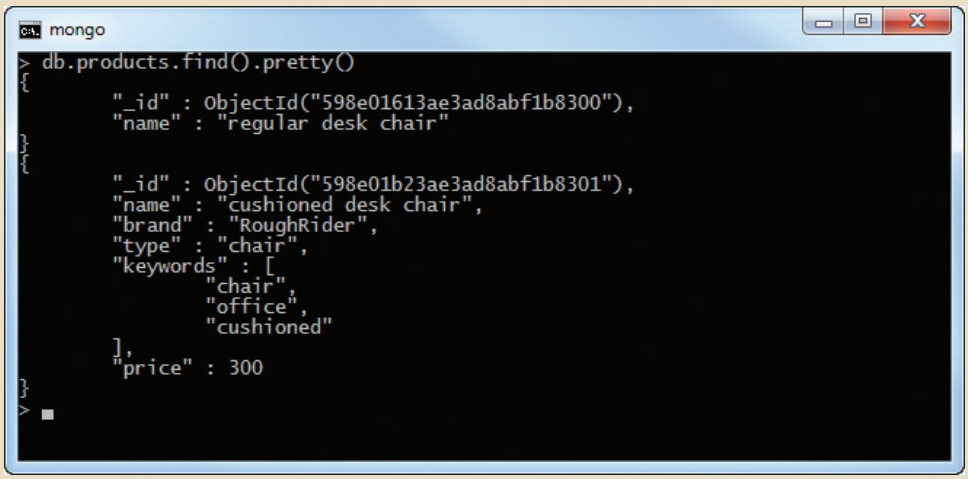
The following command performs a replacement update on the "standard desk chair" document that was first inserted into the products collection:

```
db.products.update({name: "standard desk chair"},
                   {name: "regular desk chair"},
                   false,
                   false)
```

Looking at the first document returned in Figure P.5 shows that the entire document for "standard desk chair" has been replaced with a document that contains only the *name* property. Comparing with Figure P.4 shows that the _id remains the same for the document. In the previous code, the query object states that the update should match documents that have the value "standard desk chair" for the key *name*. The change object indicates that the new version of the document should contain only a name key with the value "regular desk chair". Upsert and multi flags are both false, so a new document will not be created if the query object does not match any existing documents, and only the first document found to match will be updated.

The change object can include multiple *key:value* pairs, including complex values such as arrays and embedded objects. The following command replaces the same document with a new version:

FIGURE P.5 A REPLACEMENT UPDATE



db.products.update({name: ”regular desk chair”},

{name: “basic desk chair”, price: 100, brand: “RoughRider”, type: “chair”})

Note that because both the upsert and multi flags are omitted, the default value of false is used for both.

P-6b Operator Updates with $set, $unset, and $inc

Operator updates follow the same syntax as replacement updates, except in the change object. Operator updates use one or more update functions, or operators, in the change object. Common update operators are shown in Table P.1. Notice that MongoDB operators start with a $, and each operator can accept an object as a value. The syntax for an operator is:

<operator>: {<object>}

TABLE P.1

MONGODB COMMON UPDATE OPERATORS

OPERATOR	DESCRIPTION
$addToSet	Adds a value to an array if the value does not already exist in the array
$each	Modifies $push and $addToSet to allow adding an array of values to an array
$inc	Increments or decrements a value by the amount specified
$pull	Removes all occurrences of a value from an array
$pullAll	Removes all occurrences of an array of values from an array
$push	Adds a value to an array even if adding the value creates duplicates
$rename	Changes the value of the key portion of a key:value pair
$set	Places a value in a key:value pair
$unset	Removes a key:value pair

One of the most commonly used update operators is the **$set** operator. The $set operator works much like the SET clause of a SQL UPDATE command. With $set, the programmer specifies the key:value pair to change in the object parameter. For example, the name of the "basic desk chair" from the previous example can be changed using the $set operator. Because this is an operator update instead of a replacement update, the other key:value pairs in the document are not affected.

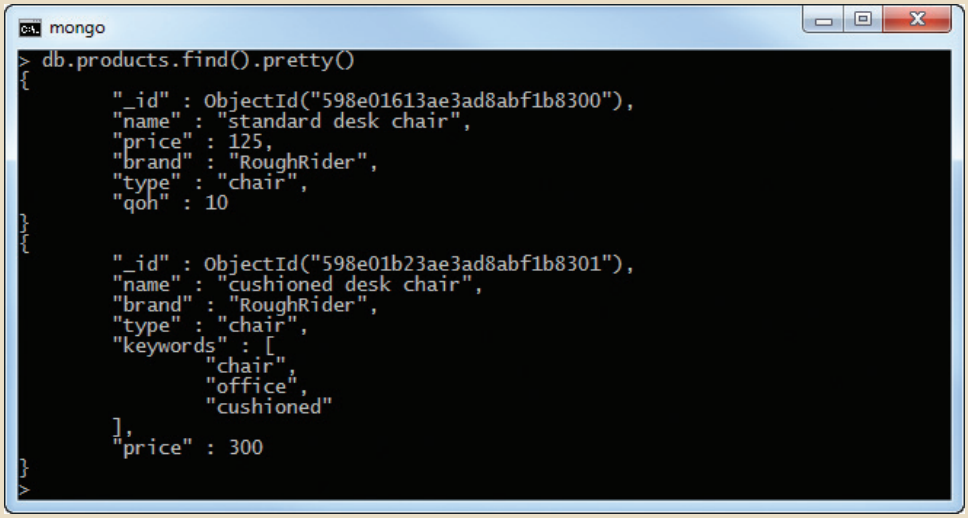
```
db.products.update({name: "basic desk chair"},
                   {$set:
                       {name: "standard desk chair"}
                   })
```

If the key specified in the $set operator's object parameter exists, then it is updated. If it does not exist, then the key is added to the document with the value specified. The following command is used to change the price to 125 and add quantity on hand (qoh) to the document. Figure P.6 shows the cumulative effect of these updates.

$set
A MongoDB operator used with the update() method to specify new values for the listed key:value pairs in a document.

```
db.products.update({name: "standard desk chair"},
                   {$set:
                       {price: 125, qoh: 10}
                   })
```

FIGURE P.6 CUMULATIVE EFFECT OF UPDATES WITH $SET

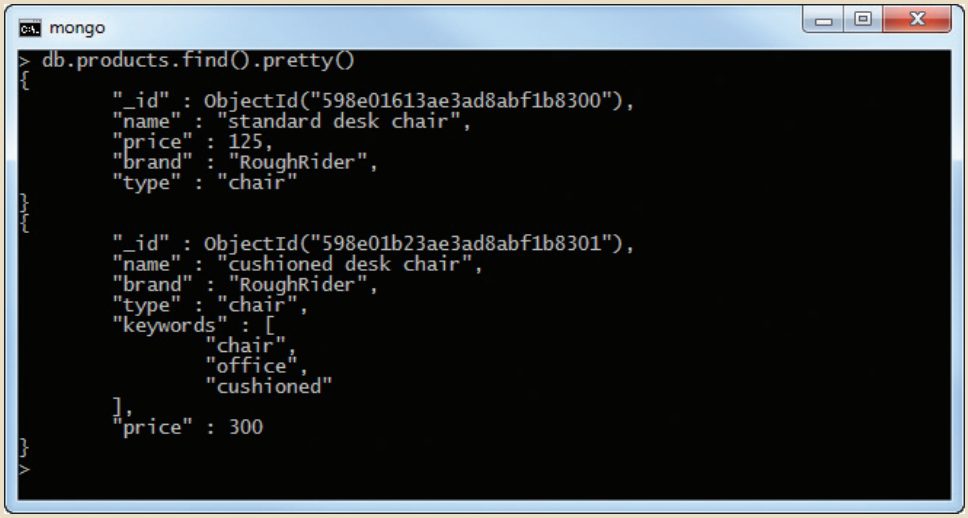


Just as the $set operator can be used to add a new key:value pair to a document, the **$unset** operator can be used to remove a selected key:value pair from a document. The following command removes the qoh key:value pair from the document. As shown in Figure P.7, the qoh key is removed.

db.products.update({name: "standard desk chair"},
{$unset:
{qoh: 1}
})

$unset
A MongoDB operator used with the update() method to remove a key:value pair from a document.

FIGURE P.7 REMOVING A KEY WITH $UNSET

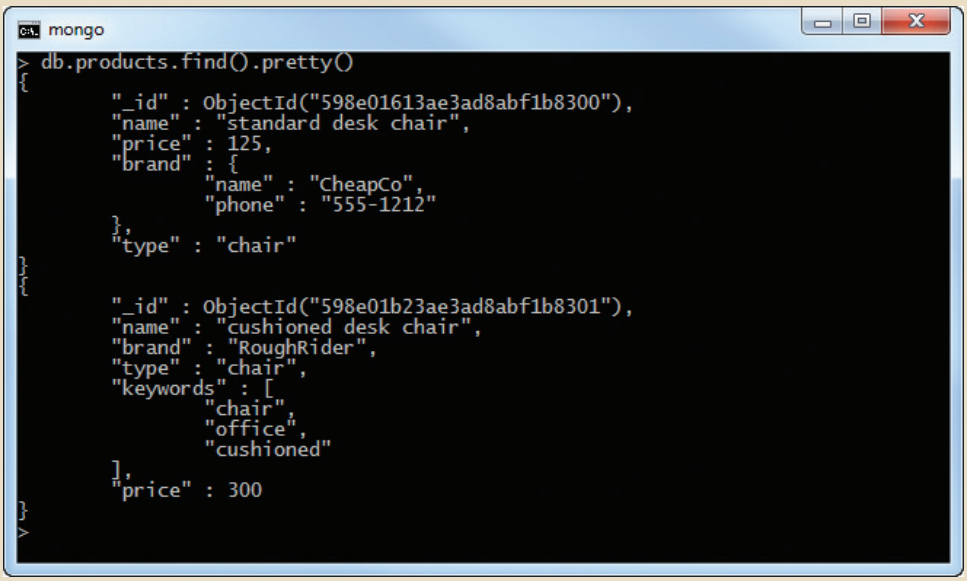


Remember, all operators accept an object parameter. The object parameter must include a minimum of a key:value pair. Since $unset is removing the pair, the actual value provided in the command is immaterial as long as some value is provided. In the previous example, the value "1" is provided for the qoh key. Using the value "1" with $unset is common, but any value would work.

The following command changes the brand key to contain a subdocument (see Figure P.8).

```
db.products.update({name: "standard desk chair"},
                   {$set:
                        {brand:
                             {name: "CheapCo",
                             phone: "555-1212"}
                        }
                   })
```

FIGURE P.8 A DOCUMENT WITH A SUBDOCUMENT



The **$inc** operator is used to increment or decrement a numeric value. The object parameter for the $inc operator specifies the key whose value is to be changed and the change amount. For example, if we wish to increase the price of chairs by $50, we could use the following command:

$inc
A MongoDB operator used with the update() method to increment or decrement values in a key:value pair without having to retrieve the existing value first.

```
db.products.update({type: "chair"},
                   {$inc:
                        {price: 50},
                   },
                   false, true)
```

FIGURE P.9 MULTI-UPDATE USING $INC

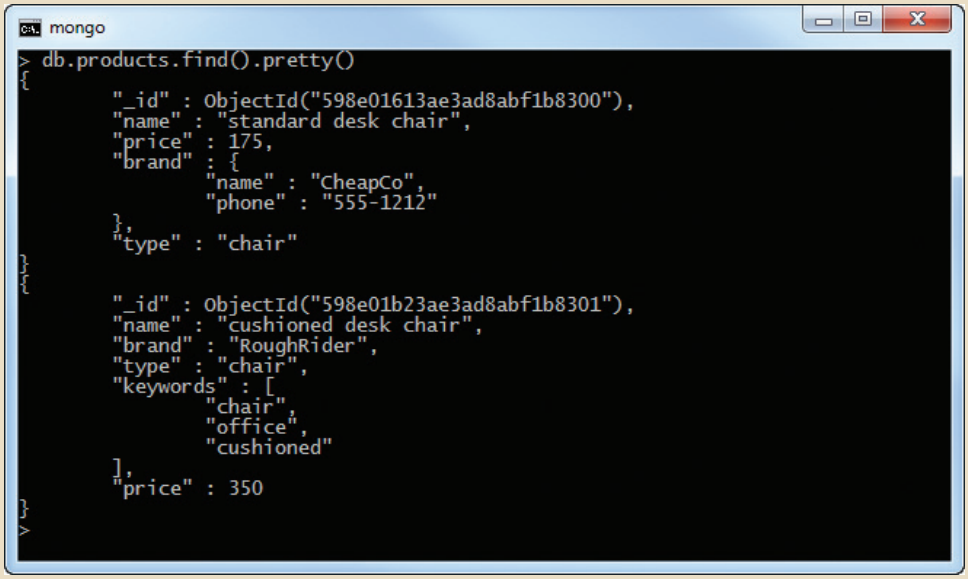


Figure P.9 shows that the previous command changes the price of both chairs by $50. Although the price of the either chair could have been changed using the $set operator, that would require knowing the existing price of the chair. The advantage of the $inc operator is that the current value does not have to be retrieved to calculate the new value. To decrease the value of an attribute, simply enter a negative value for the change amount.

Note

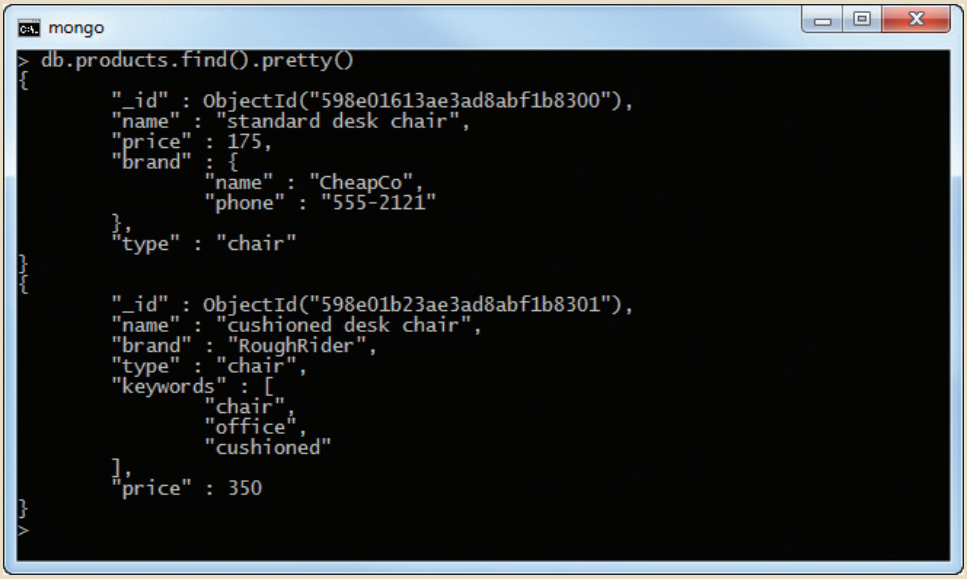
With the MongoDB update() method, a value for the upsert or multi flag can be set using an object containing a key:value pair instead of relying on the ordinal position within the command. For example, the previous update could have specified the value true for the multi flag as follows:

db.products.update({type: "chair"}, {$inc: {price: 50}}, {multi: true})

Updating the values in a subdocument can also be done using the $set, $unset, and $inc operators. The only modification is that the name of the subdocument must be added to the name of the key in the subdocument that is to be modified. Dot (.) notation is used when adding the subdocument name to the key name with the format "<*subdocument*>.<*key*>." To ensure that MongoDB correctly interprets the combined name as a single value, the combined name must be placed inside quotes. The following command changes the phone number for the embedded subdocument in the standard desk chair document (see Figure P.10).

```
db.products.update({name: "standard desk chair"},
                   {$set:
                          {"brand.phone": "555-2121"}
                   })
```

FIGURE P.10 UPDATING A SUBDOCUMENT



P-6c Updating Arrays with $push, $pull, and $addToSet

Changing values in an array can be slightly different. To change the values in an array, it is possible to use the $set operator, but the entire array would have to be set at the same time. The cushioned chair document that was previously inserted included a "keywords" key that contains the array ["chair," "office," "cushioned"]. The additional keyword "fancy" could be added to the array using the $set operator if the entire array is reentered, as follows:

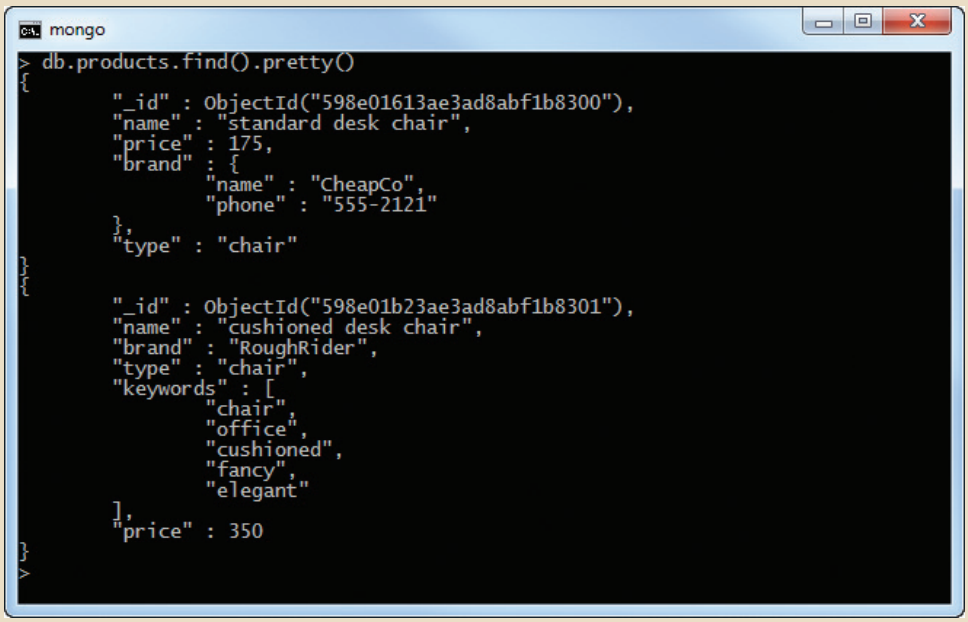
```
db.products.update({name: "cushioned desk chair"},
                   {$set:
                   {keywords: ["chair", "office", "cushioned", "fancy"]}
                   })
```

Although this produces the required result, it is not normal practice. In most cases, the programmer is attempting to add or remove values from the array, not replace every value in the array. Using $set in this situation is inefficient because it would require first retrieving the entire array to know which values already exist in the array so that they can be included in the update operation. Typically, values are added to or removed from the array without having to retrieve the array first. This is done using the $push and $pull operators.

The **$push** operator is used to add a value to an array. The **$pull** operator is used to remove a value from an array. Other values in the array are unaffected by $push and $pull. Just like $set and $unset, $push and $pull accept an object parameter that describes the change to make. For example, to add the keyword "elegant" to the cushioned chair document, the following command could be used (results shown in Figure P.11):

```
db.products.update({name: "cushioned desk chair"},
                   {$push:
                   {keywords: "elegant"}
                   })
```

FIGURE P.11 ARRAY VALUE ADDED WITH $PUSH



The query object tells the update() method to find a document with the value "cushioned desk chair" in the name key. The change object tells the method to push, or add, the value "elegant" to the array of values for the keywords key. Since the upsert and multi flags were omitted, they take the default value of false.

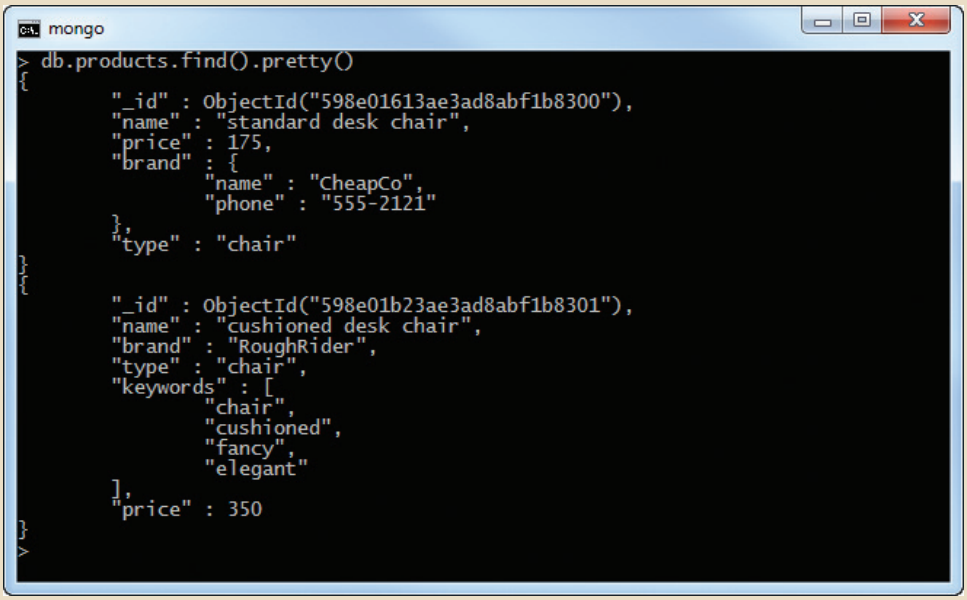
The $pull operator has the exact same syntax as the $push operator. The only difference is that $pull removes the value from the array instead of adding it to the array. The keyword "office" can be removed from the array using the $pull operator as follows (results shown in Figure P.12):

```
db.products.update({name: "cushioned desk chair"},
                   {$pull:
                   {keywords: "office"}
                   })
```

$push
A MongoDB operator used with the update() method to add values, possibly duplicates, to an array in a document.

$pull
A MongoDB operator used with the update() method to remove all occurrences of a value from an array in a document.

FIGURE P.12 ARRAY VALUE REMOVED WITH $PULL



As you examine the results of the update in Figure P.12, notice that the other values in the array were not impacted.

It is possible, and in some cases desirable, for an array to contain duplicate values. If an array contains a list of websites visited by a user, we may wish to know every time the user visited a given website even if that means the website appears in the array multiple times. Other times, we do not want duplicates to appear in an array. In the keywords for our cushioned chair, there is no advantage to storing the keyword "elegant" multiple times. The $push operator allows duplicate array values. If you were to re-execute the earlier command to push "elegant" to the keywords array, the value would appear more than once. To allow programmers to add a value to an array only if the value does not already exist in the array, the **$addToSet** operator is used. The $addToSet operator uses the same syntax and operates in the same manner as the $push operator except that it does not add a value to the array if the value already exists in the array. The following command can be used to add the value "expensive" to the keywords array:

```
db.products.update({name: "cushioned desk chair"},
                   {$addToSet:
                   {keywords: "expensive"}
                   })
```

This command adds "expensive" to the keyword array because the value "expensive" was not already in the array. If you use the $addToSet operator with a value that already exists in the array, the value will not be added again, as shown below.

```
db.products.update({name: "cushioned desk chair"},
                   {$addToSet:
                   {keywords: "cushioned"}
                   })
```

If a value appears in an array multiple times, the $pull operator will remove every occurrence of that value from the array.

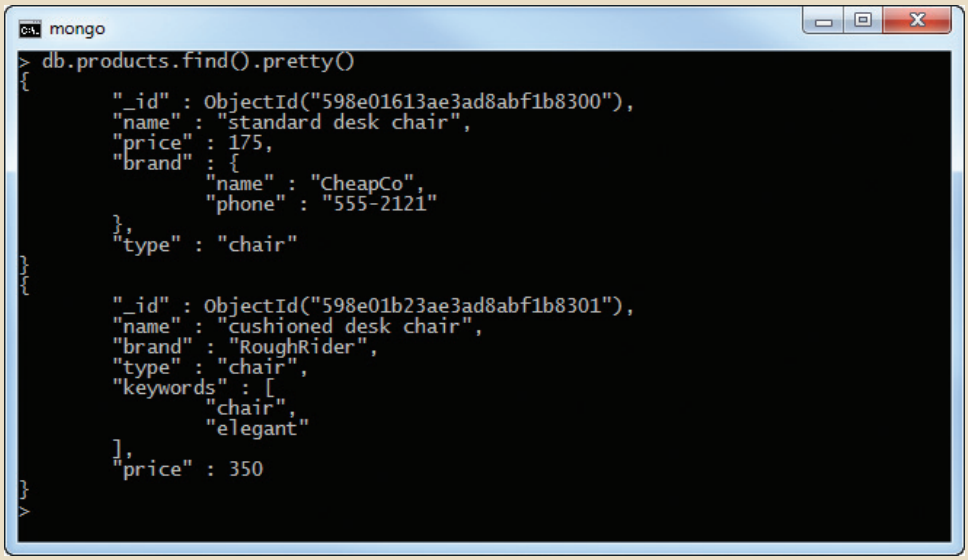
$addToSet
A MongoDB operator used with the update() method to add values, suppressing duplicates, to an array in a document.

P-6d Updating Arrays with Arrays using $each and $pullAll

The previous operators are very good at working with individual values in arrays, whether it is a single, simple value or an object value. However, since arrays are designed to hold multiple values, we often want to modify the contents of an array with multiple values simultaneously. It is not possible to update an array with an array of values using $push, $pull, and $addToSet. Luckily, only small modifications are needed to accomplish this task. To remove an array of values from an existing array, use the **$pullAll** operator instead of the $pull operator. The $pullAll operator syntax is the same as the $pull operator except that it accepts an array of values instead of a single value. The following command removes the values "fancy," "expensive," and "cushioned" from the keywords array, as shown in Figure P.13.

```
db.products.update({name: "cushioned desk chair"},
        {$pullAll:
            {keywords: ["fancy", "expensive", "cushioned"]}
        })
```

FIGURE P.13 REMOVING AN ARRAY FROM AN ARRAY



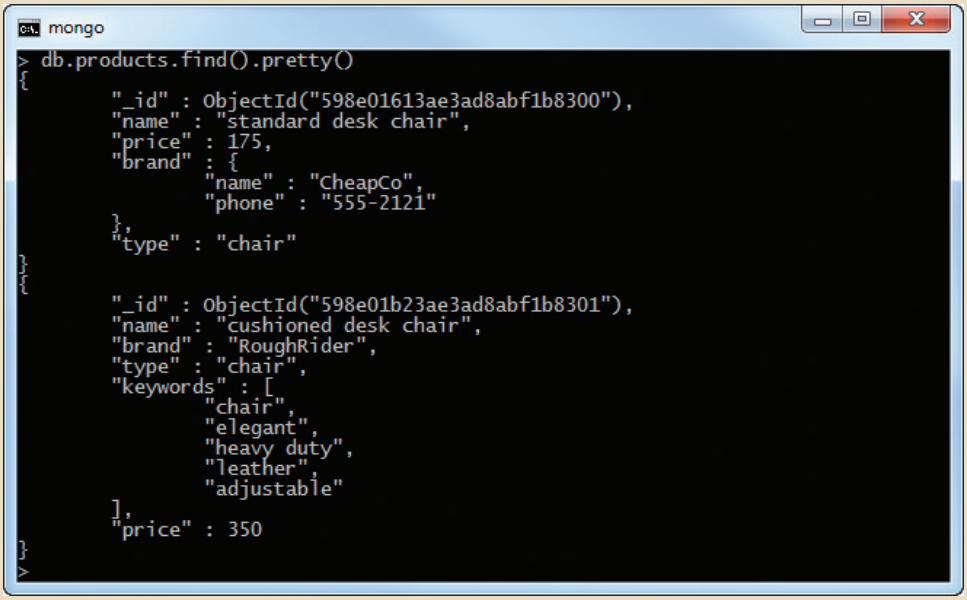
The $push and $addToSet operators can be used in conjunction with the **$each** operator. The $each operator modifies $push and $addToSet to iterate, or loop, over a set of values and act on each one separately. For example, if we want to add the values "heavy duty," "leather," and "adjustable" to the keywords array, the following command combines $push and $each to complete that task, as shown in Figure P.14.

```
db.products.update({name: "cushioned desk chair"},
        {$push:
            {keywords:
                {$each: ["heavy duty", "leather", "adjustable"]}
            }
        })
```

$pullAll
A MongoDB operator used with the update() method to remove an array of values from an array in a document.

$each
A MongoDB operator used with the update() method that modifies the function of the $push and $addToSet operators to allow an array of values to be added to an array in a document.

FIGURE P.14 ADDING AN ARRAY TO AN ARRAY

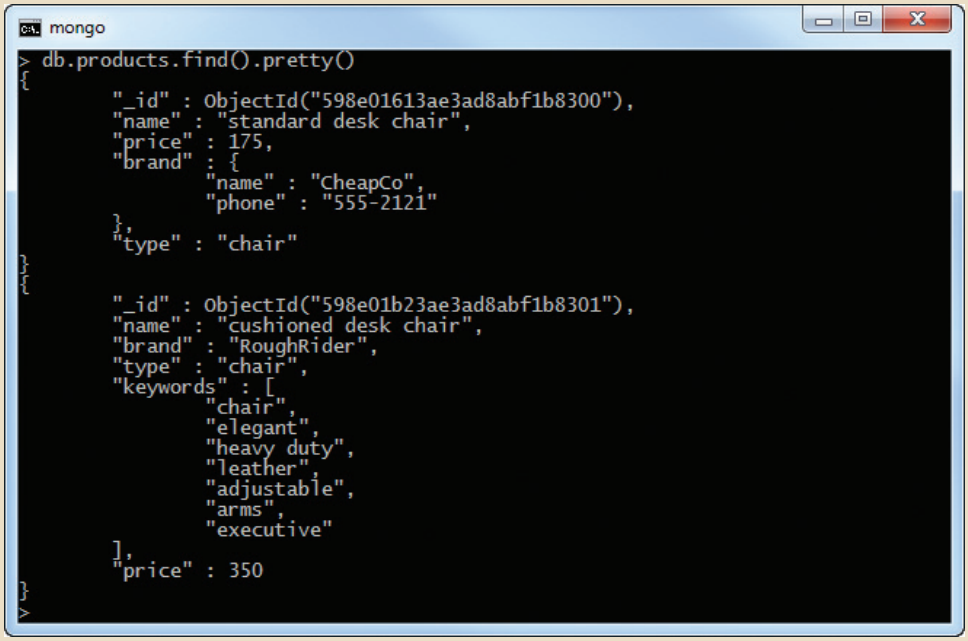


Just as before, the $push operator takes an object parameter. When adding a single value to the array, the object parameter is a simple key:value pair. When adding an array, the *value* in the key:value pair is an object that contains its own key:value pair with the $each operator as the key and an array as the value. This syntax is also used when the $each operator is used with the $addToSet operator. The following command attempts to add the values "arms," "heavy duty," and "executive" to the keywords array, with the results shown in Figure 4.26.

```
db.products.update({name: "cushioned desk chair"},
        {$addToSet:
                {keywords:
                        {$each: ["arms", "heavy duty", "executive"]}
                }
        })
```

Looking at Figure P.15, notice that the value "heavy duty" appears only once because $addToSet does not create duplicate values in the array.

FIGURE P.15 ADDING ONLY NEW VALUES TO AN ARRAY WITH AN ARRAY



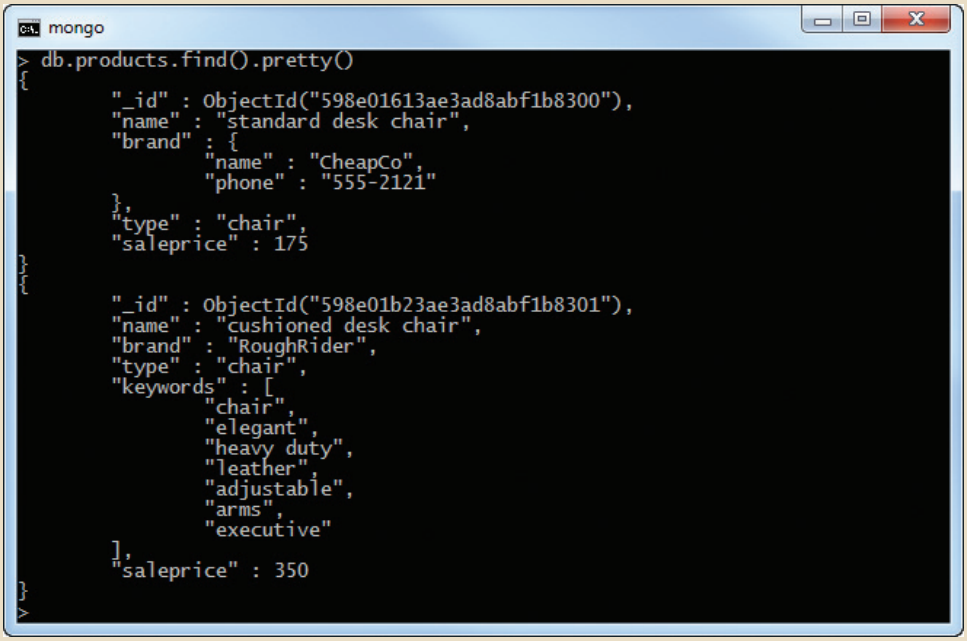
P-6e Updating to Rename Keys with $rename

All of the previous update operations have worked on changing the values in key:value pairs. It is also possible to change the key in the key:value pair by using the **$rename** operator. The $rename operator also uses an object parameter. In this case, the object parameter contains a simple, text key:value pair in which the key contains the name of the key to be renamed, and the value is the text that the key name should be changed to, as in {<old_name>:<"new_name">}. The value component that contains the new name is read by MongoDB as a value when it is checking the syntax of the command. Therefore, the new value must be inside quotes because it is a text value. For example, the following command renames the price key to saleprice for all documents in the collection, as shown in Figure P.16:

```
db.products.update({},
                   {$rename:
                          {price: "saleprice"}
                   },
                   {multi:true})
```

$rename
A MongoDB operator used with the update() method to change the name of the key portion of a key:value pair in a document.

FIGURE P.16 RENAMING THE KEY IN A PAIR



Notice that the query object in this command is empty. Including a query object is required, but the object is allowed to be empty. An empty query object matches every document, so combined with the *multi* flag being set to true, the update operation is performed on every document in the collection.

P-7 Deleting Documents in MongoDB

Deleting documents from a MongoDB collection is straightforward with few options. Documents are deleted using the **remove()** method of the collection. The remove() method accepts a single query object as a parameter. Documents that match the query object are deleted from the collection. As with the update() method, the query object can be an empty object, which would delete every document in the collection, but that is rarely used in practice. The following code deletes the standard desk chair document, as shown in Figure P.17:

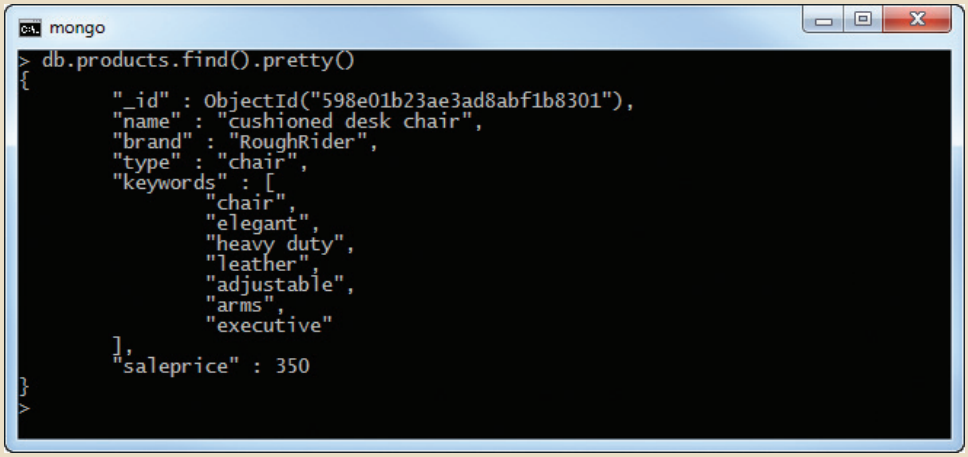
remove()
In MongoDB, a method to delete specified documents from a collection.

```
db.products.remove({name: “standard desk chair”})
```

Note

Documents cannot be manually deleted in a capped collection.

FIGURE P.17 REMOVING A DOCUMENT FROM MONGODB



P-8 Querying Documents in MongoDB

The collection of documents used to illustrate MongoDB queries is based on the data used in earlier chapters for a small library. The portion of the model that is being used here consists of documents with *patron* as the central entity. The documents have the following structure:

{_id: <system-generated ObjectID,

display: <the patron's full name as it will be displayed to users>,

fname: <patron's first name in all lowercase letters>,

lname: <patron's last name in all lowercase letters>,

type: <either "faculty" or "student">,

age: <patron's age in years only if the patron is a student>,

checkouts: <an array of objects for the patron's checkout history>

[id: <an assigned number for this checkout object>,

year: <the year in which this checkout occurred>,

month:<the month in which this checkout occurred>,

day: <the day of the month in which this checkout occurred>,

book:<the book number of the book for this checkout>,

title:<the title of the book>,

pubyear: <the year the book was published>,

subject:<the subject of the book>]

}

Online Content

The documents for the *fact* database are available as a collection of JSON documents that can be directly imported into MongoDB. The file is named Ch14_Fact.json and is available at *www.cengagebrain.com*.

Notice that the patron's name is stored twice, once with first and last name together with capitalization, and again with first name and last name in all lowercase letters in separate key:value pairs. All searches in MongoDB are case sensitive by default. Making a query case insensitive typically requires the use of a programming technique called regular expressions, which can be very detrimental to performance for MongoDB. Therefore, it is common in practice to store text values in which capitalization matters twice: once capitalized the way it should be displayed, and once in all uppercase or lowercase letters to simplify queries.

Note

The following section uses the *fact* database and the *patron* collection that was adapted from the Ch07_FACT database used in Chapter 7, Introduction to SQL. Free Access to Computer Technology (FACT) is a small library run by the Computer Information Systems department at Tiny College. The database can be created using the Ch14_Fact.json file by using the following command at an operating system command prompt (note that the command is for use at a command prompt in the OS, not inside the MongoDB shell).

```
mongoimport --db fact --collection patron --type json --file Ch14_Fact.json
```

Mongoimport is an executable program that is installed with MongoDB that is used to import data into a MongoDB database. The above command specifies that the imported documents should be placed in the fact database (if one does not exist, it will be created) and in the patron collection (if one does not exist, it will be created). Mongoimport can work with different file types such as CSV files and JSON files. The type parameter specifies that the imported documents are already in JSON format. The file parameter specifies the file name of the file to be imported. If your copy of the Ch14_Fact.json file is not in the current directory for your command prompt, you will need to provide an appropriate path for the file location.

P-8a Selection and Restriction with the find() Method

As shown previously, the basic method used to retrieve documents in MongoDB is the find() method. The find() method of a collection retrieves documents from the collection that match the restrictions provided. The method accepts two object parameters: the query object that specifies criteria that the documents must match to be included and a projection object that specifies which keys will be included in the returned documents. Both parameters are optional. The syntax for the find() method is:

```
find({<query>}, {<projection>})
```

Note

To work along with the following commands, you must have imported the Ch14_Fact.json file as described earlier. Then enter the Mongo shell using the command:

mongo

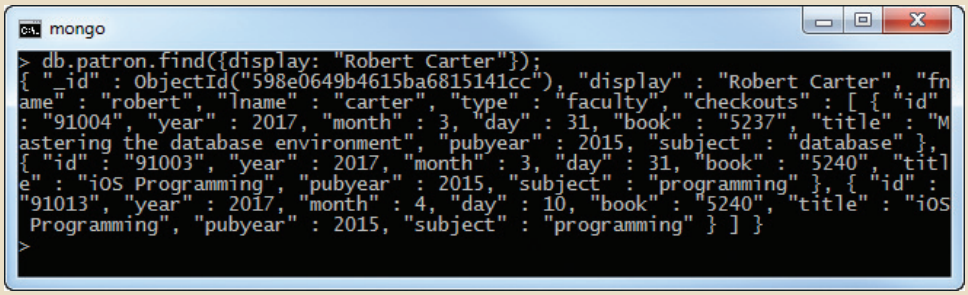
Then change to the fact database using the command:

use fact

The query object parameter functions similar to the WHERE clause of a SQL SELECT query. Providing a simple key:value pair in the query parameter is interpreted as an equality condition. For example, the following command retrieves the patron with the display name "Robert Carter," as shown in Figure P.18:

db.patron.find({display: "Robert Carter"})

FIGURE P.18 RETRIEVING A DENSE DOCUMENT



Note

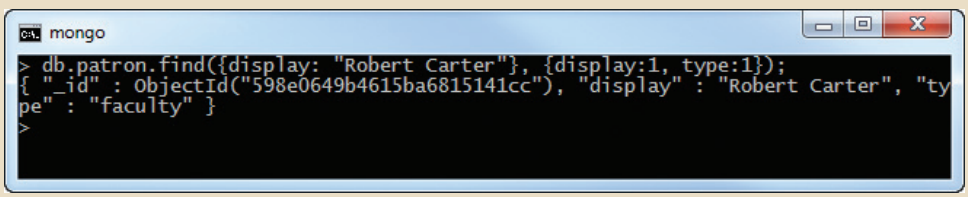
Notice that because the projection object parameter is optional, it can be omitted when the entire document should be returned.

In the previous command, the query object parameter is *{display: "Robert Carter"}*. This is the equivalent of *WHERE display = "Robert Carter"* in a SQL SELECT query.

The projection object parameter is used to specify which key:value pairs to return. The projection object contains one or more key:value pairs. The keys in the projection objects are the keys from the document to be projected. The value with each key in the projection object is either the value 0 or 1. Specifying the value 1 with a key indicates that that key:value pair from the document should be included in the results. Specifying the value 0 with a key indicates that that key:value pair from the document should be omitted. For example, the following command retrieves the document for patron "Robert Carter" but only returns the _id, display name, and type keys, as shown in Figure P.19:

db.patron.find({display: "Robert Carter"}, {display:1, type:1})

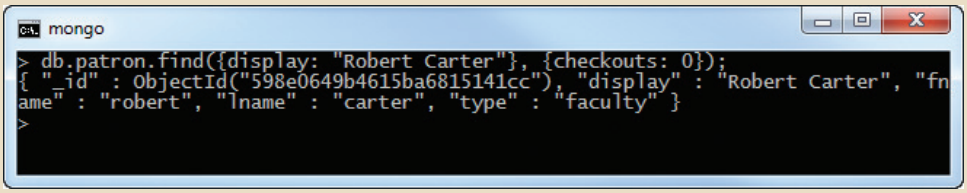
FIGURE P.19 PROJECTING KEY:VALUE PAIRS



Notice that the _id key is returned by default even when it is not specifically asked for. To return all keys except the ones specified, use the 0 value in the projection object. For example, the following command returns all keys in the Robert Carter document except the checkouts, as shown in Figure P.20:

db.patron.find({display: “Robert Carter”}, {checkouts: 0})

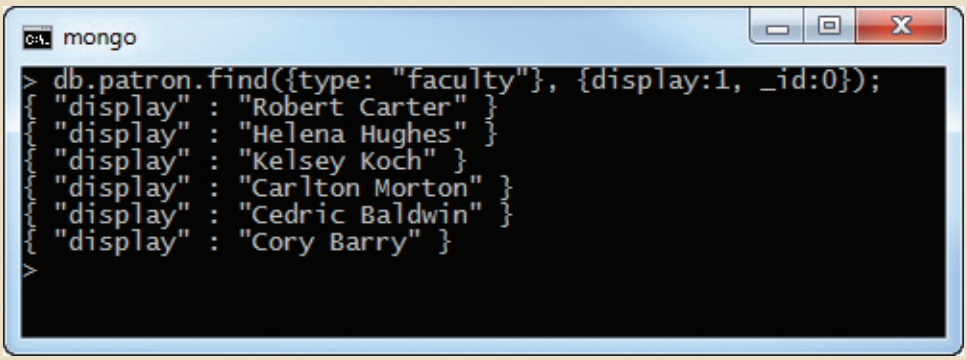
FIGURE P.20 EXCLUDING A PAIR FROM THE OUTPUT



Generally, it is not possible to specifying including and excluding keys in the same query. The exception is the _id key that is returned by default. When specifying keys to include, if the programmer wishes to suppress the _id key, it can be explicitly excluded. The following command returns just the display name key of all faculty patrons, as shown in Figure P.21:

db.patron.find({type: “faculty”}, {display:1, _id:0})

FIGURE P.21 MIXING INCLUSION AND EXCLUSION IN PROJECTIONS

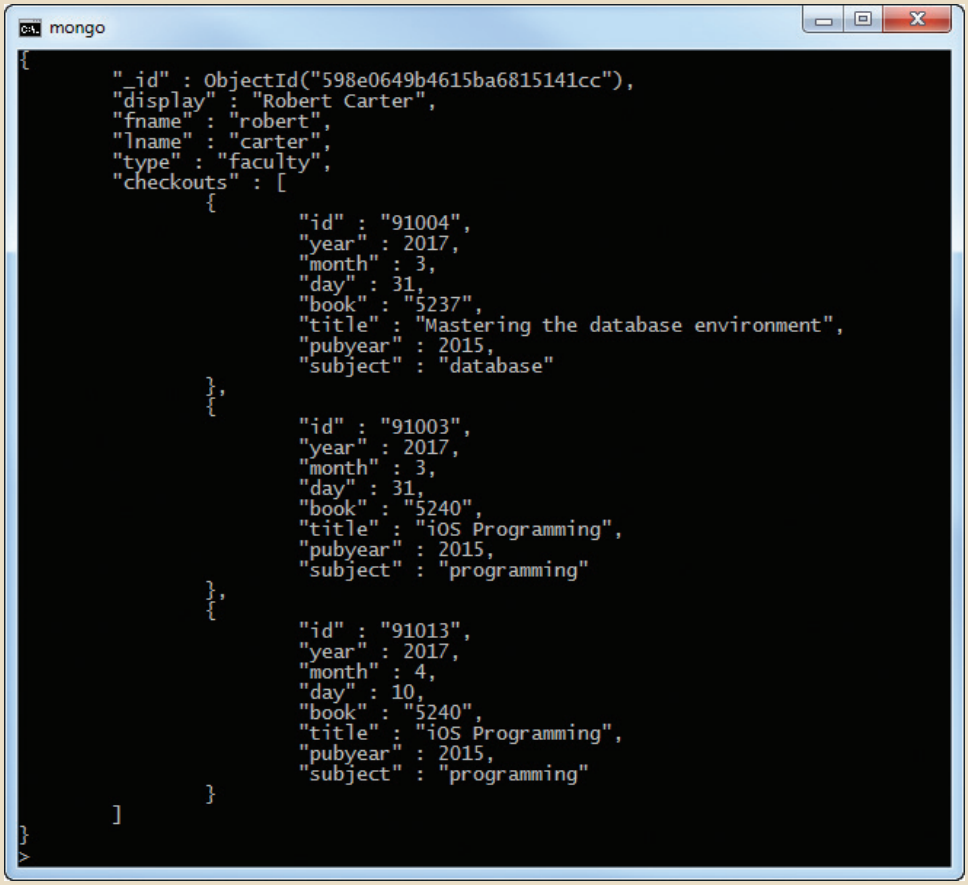


pretty()
In MongoDB, a method that can be chained to the find() method to improve the readability of retrieved documents through the use of line breaks and indention.

The JSON document is returned in dense format, where each key:value pair is separated by a space. This is the default view of documents and is appropriate when the documents are very simple or when the document is being returned to an application, not for human readability. To improve the readability of the returned document, the pretty() method can be used. The **pretty()** method is a MongoDB method to improve the readability of documents by placing key:value pairs on separate lines. The pretty() method can be added to the find() method, and it does not take any parameters. The following command combines the find() and pretty() methods, as shown in Figure P.22.

db.patron.find({display: “Robert Carter”}).pretty()

FIGURE P.22 USING THE PRETTY() METHOD

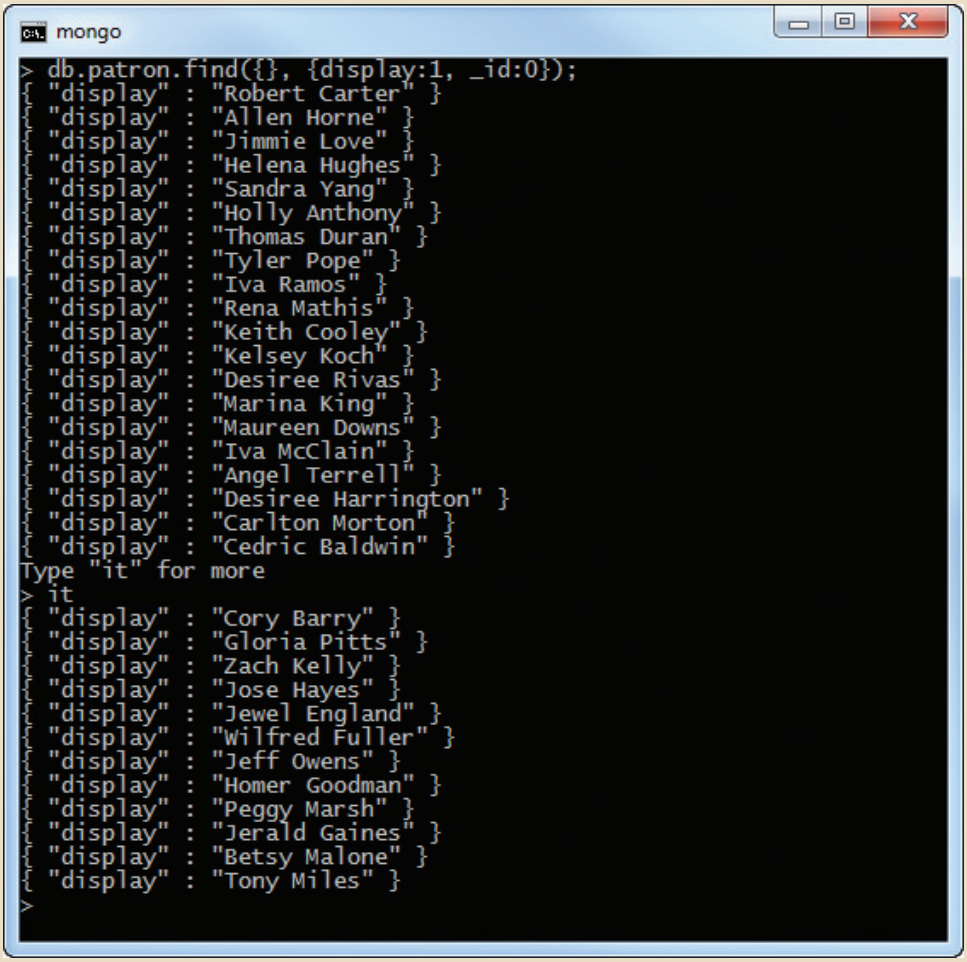


Recall that both the query object parameter and the projection object parameter are optional. If both are omitted, then the find() method is written with no parameters. Ifthe query object parameter is needed, but the projection object parameter is not, then just the projection object can be omitted. However, if the query object parameter is not needed, but the projection object is, then the programmer cannot simply omit the query object. If only one object parameter is provided, then MongoDB will assume that the one parameter is intended to be the query object parameter. In these cases, an empty object can be used for the query object similar to the way an empty object was used for updating all documents previously. For example, the following command retrieves only the name key of every document, as shown in Figure P.23:

```
db.patron.find({}, {display:1, _id:0})
```

Notice in Figure P.23 that when working in the MongoDB shell, if a query returns more than 20 documents, only the first 20 are displayed on screen followed by a prompt to type "it" to display the next 20. In these cases, all documents are found and returned, but the shell is performing a simple manipulation to reduce scrolling the screen with data.

FIGURE P.23 ITERATING OVER MANY DOCUMENTS



P-8b Document Pagination with limit() and skip()

method chaining
In MongoDB, serially listing multiple methods of a collection in the same command to perform multiple operations on the collection simultaneously.

limit()
In MongoDB, a method that can be chained to the find() method to restrict output to a specified number of documents to be retrieved. Often used for pagination of results.

The MongoDB shell's behavior of displaying only a few documents at a time can be helpful, but is rather limited and might not meet the programmer's needs. Other methods can be used with the results of the find() method to help improve the flow of documents both when viewing them interactively in the shell or when outputting them to an application.

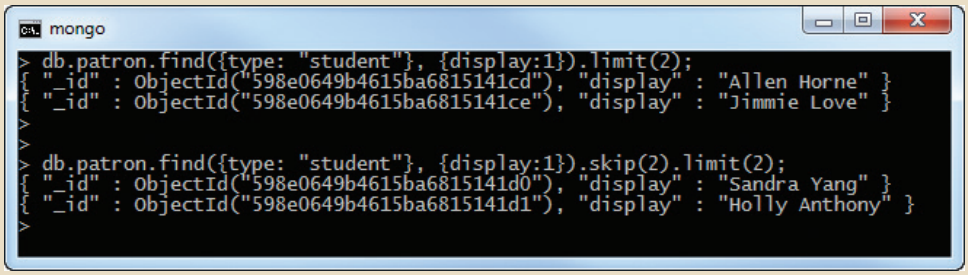
When multiple methods are applied to the same collection, this is referred to as **method chaining**. Using the pretty() method with find() in an earlier example was a simple example of method chaining. Other methods can also be chained with the find() method. Among the most common are limit(), skip(), and sort(). The **limit()** method is used to restrict the number of documents returned. Limit() takes a single numeric parameter to specify the number of documents to limit to. Unlike the query object parameter of the find() method, limit() is not a query based on the content of the documents. After the find() method has determined which documents to return, the limit() method simply restricts the result set to the number of documents specified. For example, the following command finds all documents for student patrons but limits the results to just the first two documents.

```
db.patron.find({type: "student"}, {display:1}).limit(2)
```

Limit() is often used with applications that are retrieving data for display to users so that the data can be paginated into chunks by the application or website. The **skip()** function is also used to support pagination by allowing retrieval of subsequent chunks of documents. For example, if we are retrieving student patrons so that they can be listed for display, the previous limit() command indicates that we are using chunks of two documents at a time. To retrieve the next page of two documents, we would want to skip the first two documents because they have already been displayed and then display the next two documents. The following command would do that, as shown in Figure P.24:

db.patron.find({type: "student"}, {display:1}).skip(2).limit(2)

FIGURE P.24 USING LIMIT() AND SKIP() FOR PAGINATION



Note

If you know that the document you want will be the first document returned, instead of chaining find() and limit(1), there is a shorthand method named findOne() that returns only the first document.

db.patron.findOne({type: "student"})

is the same as

db.patron.find({type: "student"}).limit(1)

To retrieve the third paginated chunk of documents, the skip parameter would be increased to 4.

P-8c Sorting Documents with sort()

Thus far, we have not concerned ourselves with the order of the documents that are returned. Due to the internal storage mechanisms used by MongoDB, documents are often returned in the order in which they were entered into the collection. However, just as with SELECT queries, we often want to control the sorted order of the data that is being returned. The **sort()** method allows us to do this. The sort() method of a collection orders the retrieved documents based on the values of one or more keys. Similar to the projection object parameter of the find() method, the sort() method takes an object of one or more keys paired with numeric values. In the case of projection, the values were 0 or 1. With sorting, the values are 1 or −1. If a key is associated with the value 1 in the sort() method, then the documents will be sorted on ascending values in that key. If the

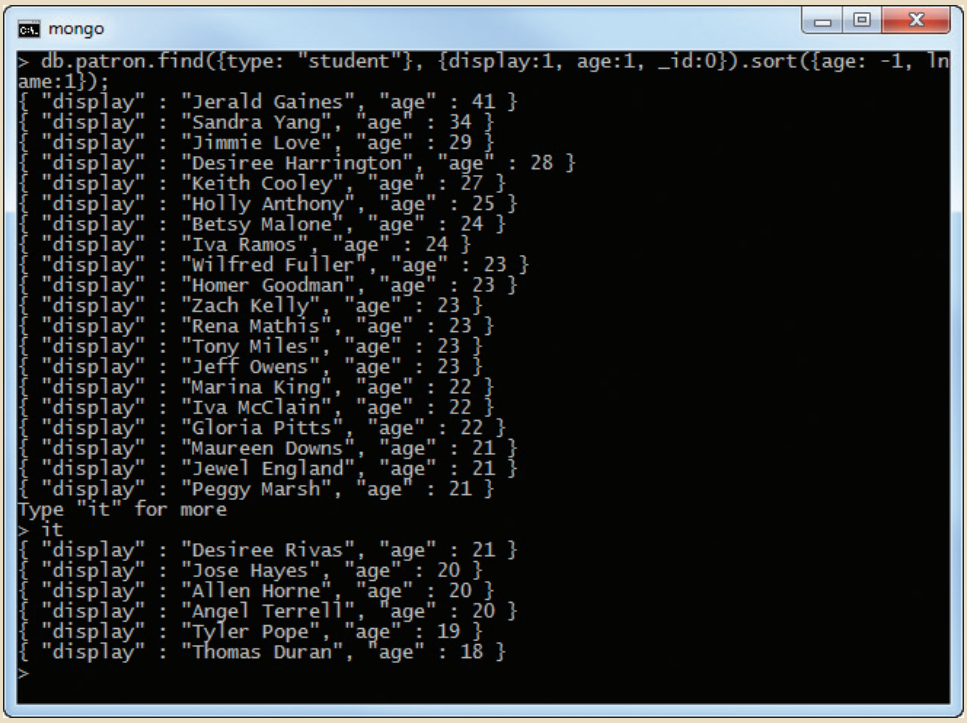
skip()
In MongoDB, a method that can be chained to the find() method to specify a number of documents to skip over in the output of documents that have been retrieved. Often used with limit() for pagination of results.

sort()
In MongoDB, a method that can be chained to the find() method to order documents being returned based on the value of one or more key:value pairs.

key is associated with the value −1 in the sort() method, then the documents will be sorted in descending order. Just as with the ORDER BY clause of a SQL SELECT query, when multiple sort attributes are specified, the results are sorted based on the attributes from left to right. For example, the following command retrieves the student patron documents, projects the display name and age, then sorts the results in descending order by age, and then within matching values for age the documents are sorted in ascending order by last name, as shown in Figure P.25:

db.patron.find({type: "student"}, {display:1, age:1, _id:0}).sort({age: −1, lname:1})

FIGURE P.25 SORTING DOCUMENTS



Interestingly, MongoDB does not care which order the limit(), skip(), and sort() methods are chained after the find() method. So the following three commands return the exact same result:

db.patron.find({type: "student"}, {display:1, age:1, _id:0}).sort({age: −1}).limit(5).skip(5)

db.patron.find({type: "student"},{display:1, age:1, _id:0}).limit(5).skip(5).sort({age: −1})

db.patron.find({type: "student"},{display:1, age:1, _id:0}).limit(5).sort({age: −1}).skip(5)

P-8d Queries Using Inequalities in MongoDB

Thus far, all of the queries that we've looked at use at most one criteria in the query object with a simple equality condition. Inequalities such as greater than, less than, and not equal to are also possible, but require the use of functions. Table P.2 lists the functions used in MongoDB for inequality comparisons.

TABLE P.2

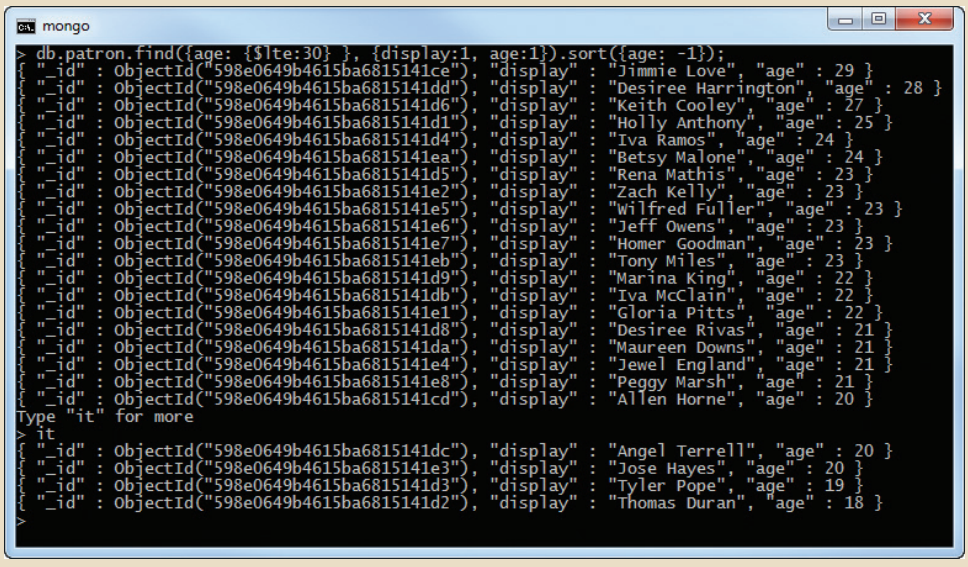
MONGODB INEQUALITY FUNCTIONS

FUNCTION	DESCRIPTION
$gt	Greater than ">"
$gte	Greater than or equal to "≥"
$lt	Less than "<"
$lte	Less than or equal to "≤"
$ne	Not equal "≠"

All of the inequality functions operate the same way and have the same syntax. The functions are used as key for a key:value pair. The value is the value that is being applied to the function. For example, the $gt function is used to match values that are greater than the value provided. Therefore, *{$gt:20}* would mean greater than 20. The function's key:value pair is used as an object value for the key that the function is to work within. If we want to limit to documents where *age > 20*, then we would place the function object *{$gt:20}* as the value in a pair with the key *age*, like *age:{$gt:20}* in the query object parameter. Similarly, if we want to find the documents where the age ≤ 30, the criteria would be written *age:{$lte:30}* because $lte is the *less than or equal to* function, 30 is the target value for the function, and the function is being used with the age key. The following command retrieves the display name and age from documents for patrons that are age 30 or less, as shown in Figure P.26:

```
db.patron.find({age: {$lte:30} }, {display:1, age:1}).sort({age: −1})
```

FIGURE P.26 USING INEQUALITIES



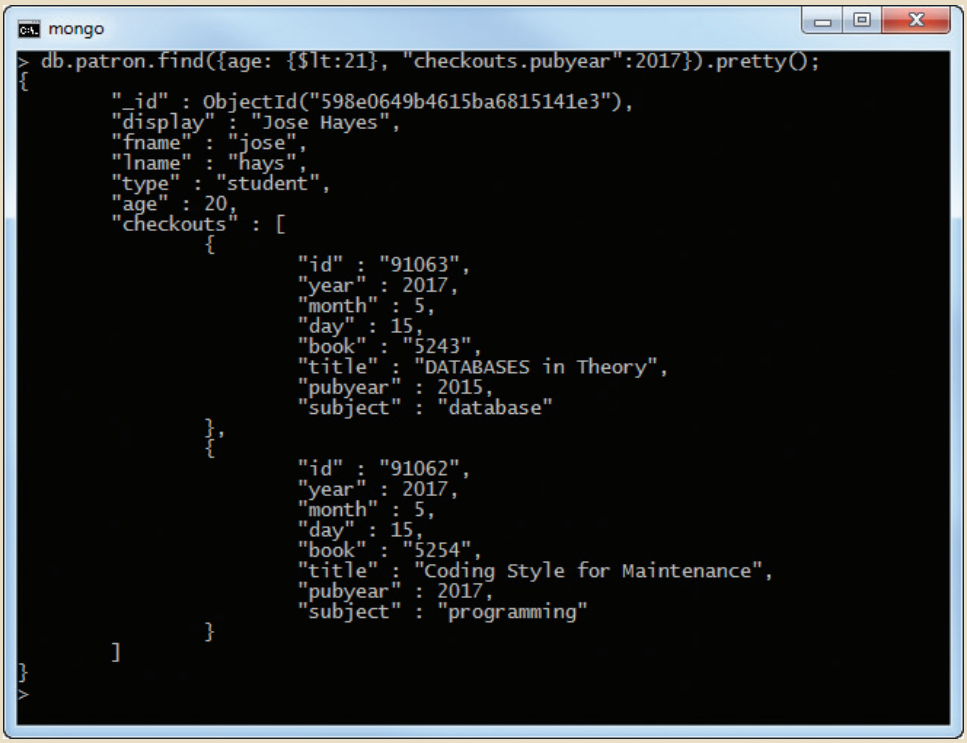
P-8e Combining Criteria with $and and $or

implicit *and*
In MongoDB, the combining of multiple criteria with a logical AND connector by separating the criteria with commas.

As you would expect, MongoDB provides the ability to combine multiple criteria in the query object parameter of the find() method using logical *and* and *or* operators. Combining criteria with a logical operator can be done both implicitly and explicitly. Separating criteria with commas in the query object employs an **implicit *and***. For example, the following query finds the patrons who are under 21 years of age and have checked out a book published in 2017. To improve readability, the result is formatted with the pretty() method, as shown in Figure P.27.

```
db.patron.find({age: {$lt:21}, "checkouts.pubyear":2017}).pretty()
```

FIGURE P.27 IMPLICIT LOGICAL AND



$and
A MongoDB operator to explicitly combine multiple criteria with a logical AND connector by placing the criteria as objects within an array of criteria.

explicit *and*
In MongoDB, the combining of multiple criteria with a logical AND connector through the use of the $and operator.

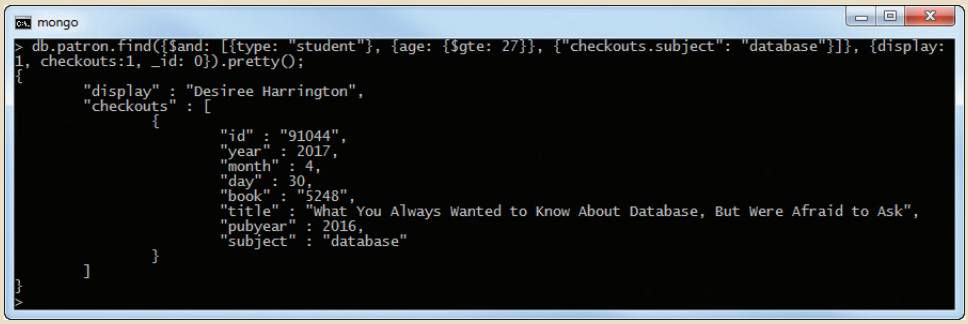
Remember, the year of publication is in a subdocument so the "checkouts.pubyear" notation is required. The previous command contained two criteria: *age:{$lt:21}* and *"checkouts.pubyear":2017*. Because these criteria are separated by a comma, MongoDB combines them with an implicit logical *and*.

Explicit logical *and* requires the use of the **$and** function. In practice, **explicit *and*** using the $and function tends to be preferred. With implicit *and*, all criteria in the list are always evaluated to true or false. With an explicit *and*, as soon as any criteria evaluates to false for a document, the remainder of the criteria are skipped for that document. This makes the explicit *and* faster and more efficient than an implicit *and* for most operations. The $and function appears as a key in a key:value pair that accepts an array of objects as the target value. For example, if *x* and *y* are criteria, then the syntax of $and would be

$and: [{x}, {y}]. Notice that each criterion is enclosed in {} to indicate that it is an object. The following command retrieves all documents that include a student over 26 years old checking out a book within the database subject, as shown in Figure P.28.

```
db.patron.find({$and: [{type: "student"}, {age: {$gte: 27}}, {"checkouts.subject":
"database"}]}, {display: 1, checkouts:1, _id: 0}).pretty()
```

FIGURE P.28 EXPLICIT LOGICAL AND USING $AND

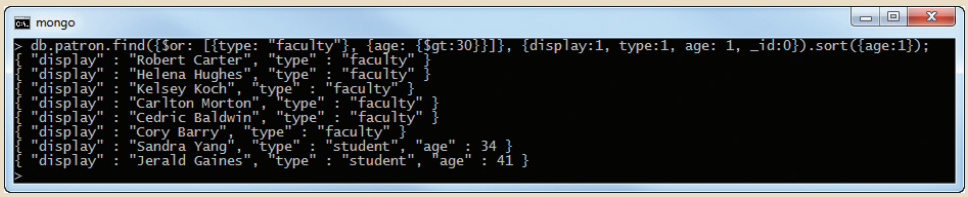


Again, notice that each criterion is contained inside its own set of brackets so that it is an object all to itself. The criteria objects are separated with a comma.

All logical *or* operations are explicit. The **$or** function has the same syntax as the $and function. The following command retrieves documents that are for faculty or for students over 30 years of age. Only the display name, type, and age will be displayed with the results sorted by age, as shown in Figure P.29:

```
db.patron.find({$or: [{type: "faculty"}, {age: {$gt:30}}]}, {display:1, type:1, age: 1,
_id:0}).sort({age:1})
```

FIGURE P.29 EXPLICIT LOGICAL OR USING $OR



As MQL queries become more complex, they can seem confusing because of all of the punctuation. Breaking down the previous query, the symbols become a little clearer.

- The query is based on the db.patron.find() method.
  - Within the find() method there are two objects, the query object and the projection object.
  - The query object is {$or: [{type: "faculty"}, {age: {$gt:30}}]}

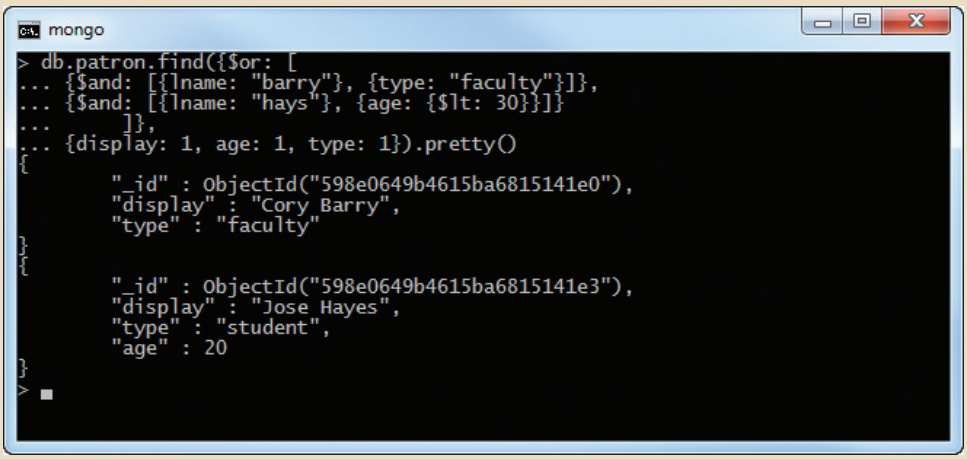
$or
A MongoDB operator to explicitly combine multiple criteria with a logical OR connector by placing the criteria as objects within an array of criteria.

  - The query object contains one key:value pair. $or is the key and [{type: “faculty”}, {age: {$gt:30}}] is the value.
  - The value for the $or key is an array composed of two objects. The first object is {type: “faculty”}, and the second object is {age: {$gt:30}}.
  - The first object in the array is a simple key:value pair, that is, type: “faculty.”
  - The second object in the array is a key:value pair with age as the key and an object for the value. The object value is {$gt:30}, which calls the *greater than* function with 30 as the target value of the function.
- The projection object contains a list of key:value pairs.
- The projection object is {display:1, type:1, age:1, _id:0}
  - The keys in the object associated with value 1 are included in the result. The key in the object associated with the value 0 is excluded from the result.
- The sort() method is chained to the find() method to specify a sorted order for the output.
  - The sort() method contains an object made of simple key:value pairs.
  - The keys in the sort object associated with the value 1 are sorted in ascending order. If any key has been associated with the value −1, it would have been sorted in descending order.

The $and and $or functions can be mixed within the same query object. The following query, with results shown in Figure P.30, retrieves the _id, display name, and age for patrons that either have the last name “barry” and are faculty, or have the last name “hays” and are under 30 years old:

```
db.patron.find({$or: [
                      {$and: [{lname: “barry”}, {type: “faculty”}]},
                      {$and: [{lname: “hays”}, {age: {$lt: 30}}]}
                      ]},
               {display: 1, age: 1, type: 1}).pretty()
```

FIGURE P.30 MIXING LOGICAL AND WITH LOGICAL OR IN THE SAME QUERY



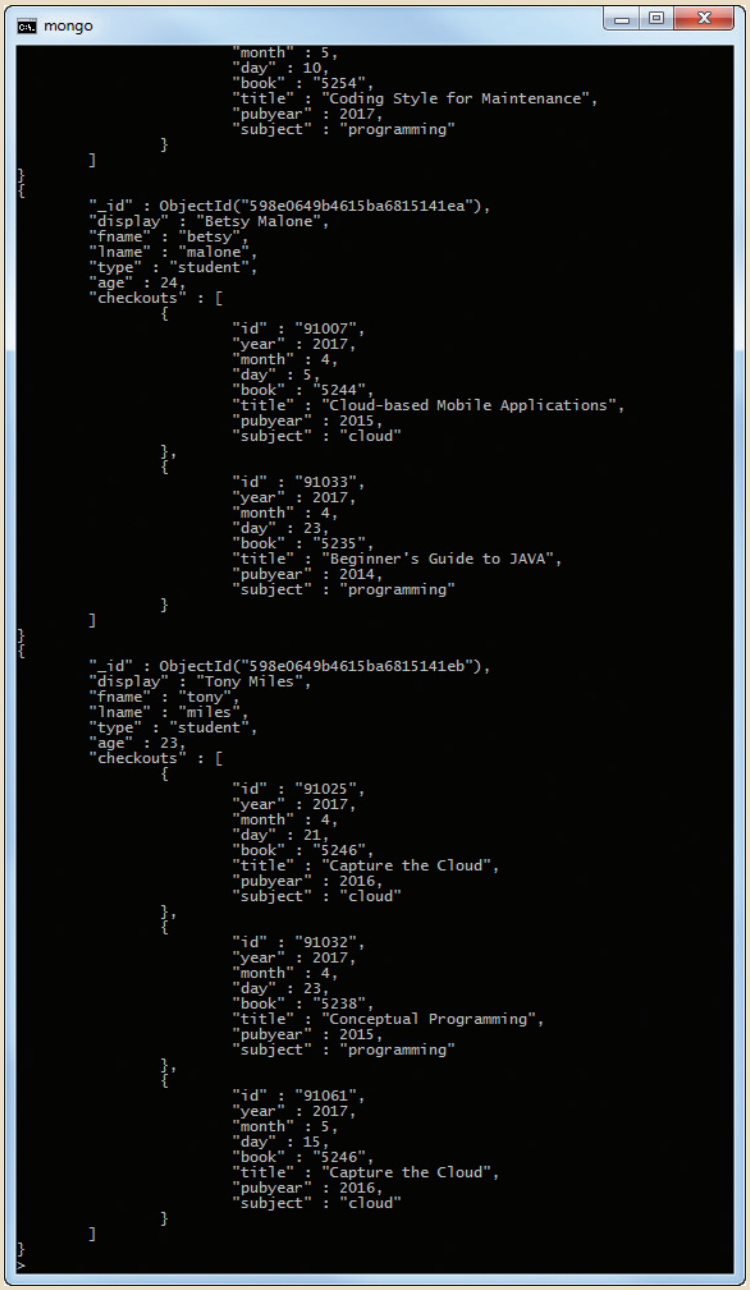
P-8f Matching Elements in Arrays with $elemMatch

As with performing updates, working with arrays can pose unique challenges when writing queries. Consider the following requirement: find the patrons who have checked out a book in the programming subject that was published in 2015. Based on our coverage of $and and using dot notation with subdocuments, we can craft the following query:

```
db.patron.find({$and: [{"checkouts.subject":
"programming"},{"checkouts.pubyear":2015}]}).pretty()
```

Carefully examining the results of that query (partial results shown in Figure P.31) will illustrate the problem of querying arrays.

FIGURE P.31 PARTIAL RESULTS OF CHECKOUTS FOR PROGRAMMING BOOKS



The results include documents that have checkouts of books in the programming subject, and books published in 2015, so both criteria are met. However, looking through the output we find a few patrons who checked out a book that was published in 2015 and who checked out a book on programming, but they were not the same book! Our intention was that the same book satisfies both criteria, not two separate books each satisfying one criteria apiece. This problem is solved with the **$elemMatch** function. In mathematical terminology, items in a set are called elements. It is, therefore, appropriate to refer to the values in an array as elements. The $elemMatch function accepts an object containing a list of criteria as a parameter. The $elemMatch function requires that all criteria included in the function be satisfied by the same element in an array. In the current database, the elements in the checkouts array are objects that represent each checkout by a patron. Using $elemMatch with the criteria in the previous command requires both criteria to be satisfied by the same checkout, as shown in the following command (see Figure P.32 for partial results).

$elemMatch
A MongoDB operator used to specify that all criteria in an array of criteria must evaluate to true for the same element in a document array.

```
db.patron.find({checkouts: {$elemMatch: {subject: "programming", pubyear:
2015}}}).pretty()
```

Examining the output of this command shows that within the checkouts of each patron returned is at least one checkout event that satisfies both criteria.

FIGURE P.32 USING $ELEMMATCH TO FIND CRITERIA IN THE SAME SUBDOCUMENT

MongoDB is a powerful document database that is being adopted by many organizations. It was originally designed to support web-based operations, and as such, it draws heavily on JavaScript for the structure of its documents and for MQL. The previous several sections have introduced you to the basic create, read, update, and delete operations that are central to database processing. This should give you a basic familiarity with MongoDB and MQL, but there is much more to learn if you are interested in pursuing a career in document databases.

Summary

- MongoDB is a document database that stores documents in JSON format. The documents can be created, updated, deleted, and queried using a JavaScript-like language, named MongoDB Query Language.
- Documents of a similar nature are stored together in a collection. Even though the documents are similar, they are not required to have exactly the same structures. A document database may contain many collections.
- Documents can be updated by either a replacement update or an operator update. A replacement update will replace an entire document with a new document that has the same unique id. An operator update can be used to modify specific key:value pairs in a document without impacting the remainder of the document.
- Data retrieval is done primarily through the find() method. Within the find() method, functions can be used to write complex logical criteria.
- Other methods can be chained to the find() method to support pagination, restricting results, and formatting output.

Key Terms

$addToSet, P-18
$and, P-32
capped collection, P-7
createCollection, P-7
drop(), P-8
$each, P-19
$elemMatch, P-36
explicit *and*, P-32
find(), P-9
getName(), P-6
implicit *and*, P-32
$inc, P-14
insert(), P-8
limit(), P-28
method, P-6
method chaining, P-28
$or, P-33
operator update, P-10
pretty(), P-26
$pull, P-17
$pullAll, P-19
$push, P-17
$rename, P-21
renameCollection, P-8
remove(), P-22
replacement update, P-10
$set, P-12
skip(), P-29
sort(), P-29
subdocument, P-3
$unset, P-13
update(), P-10
upsert, P-10

Review Questions

1. What is the difference between a replacement update and an operator update in MongoDB?
2. Explain what an upsert does.
3. Describe the difference between using $push and $addToSet in MongoDB.
4. Explain the functions used to enable pagination of results in MongoDB.
5. Explain the difference in processing when using an explicit *and* and an implicit *and* with MongoDB.

Problems

For the following set of problems, use the *fact* database and *patron* collection created in the text for use with MongoDB.

1. Create a new document in the patron collection. The document should satisfy the following requirements:

   First name is "Rachel"
   Last name is "Cunningham"
   Display name is "Rachel Cunningham"
   Patron type is student
   Rachel is 24 years old
   Rachel has never checked out a book

   Be certain to use the same keys as already exist in the collection. Be certain capitalization is consistent with the documents already in the collection. Do not store any keys that do not have a value (in other words, no NULLs).

2. Modify the document entered in the previous question with the following data. Do not replace the current document.

   Rachel has checked out two books on January 25, 2018.
   The id of the first checkout is "95000"
   The first book checked out was book number 5237
   Book 5237 is titled "Mastering the database environment"
   Book 5237 was published in 2015 and is in the "database" subject
   The id of the second checkout is "95001"
   The second book checked out was book number 5240
   Book 5240 is titled "iOS Programming"
   Book 5240 was published in 2015 and is in the "programming" subject

   Use the same keys as already exist within the collection. Conform to the existing documents in terms of capitalization.

Online Content

The documents for the *fact* database is available as a collection of JSON documents that can be directly imported into MongoDB. The file is named Ch14_Fact.json and is available at *www.cengagebrain.com*.

3. Write a query to retrieve the _id, display name and age of students that have checked out a book in the cloud subject.
4. Write a query to retrieve only the first name, last name, and type of faculty patrons that have checked out at least one book with the subject "programming".
5. Write a query to retrieve the documents of patrons that are faculty and checked out book 5235, or that are students under the age of 30 that have checked out book 5240. Display the documents in a readable format.
6. Write a query to display the only the first name, last name, and age of students that are between the ages of 22 and 26.