9 Text search

This chapter covers

- Why text search is important
- Text search basics
- Defining MongoDB text search indexes
- Using text search with MongoDB `find()`
- Using MongoDB text search with aggregation
- Using text search with different languages

In chapters 5 and 6, which explored constructing queries and using aggregation, you learned how to perform database queries using a fairly sophisticated query language. For many applications, searches using these types of queries may be sufficient. But when you're dealing with large amounts of unstructured data, or trying to support users finding the product they want to buy from a huge catalog of possible products, this type of searching may not be enough. Website visitors who have become accustomed to using Google or Amazon for searches expect much more and have come to rely increasingly on more sophisticated search technology.

In this chapter you'll see how MongoDB can provide some of the capabilities that more sophisticated text search engines provide—much more than the queries you've seen so far. These additional capabilities include indexing for fast word searches, matching exact phrases, excluding documents with certain words or phrases, supporting multiple languages, and scoring search result documents based on how well they match a search string. Although MongoDB text search isn't intended to replace dedicated search engines, it may provide enough capabilities that you won't need one.

Let's look at the various types of search capabilities dedicated search engines provide. In section 9.1.3 you'll see the subset of those capabilities provided by MongoDB.

If you've got it, why not use it?

On a LinkedIn MongoDB group discussion, someone asked what the benefit was to using MongoDB text search versus a dedicated search engine such as Elasticsearch. Here's the reply from Kelly Stirman, director of Products at MongoDB:

"In general Elasticsearch has a much richer set of features than MongoDB. This makes sense—it is a dedicated search engine. Where MongoDB text search makes sense is for very basic search requirements. If you're already storing your data in MongoDB, text indexes add some overhead to your deployment, but in general it is far simpler than deploying MongoDB and Elasticsearch side by side."

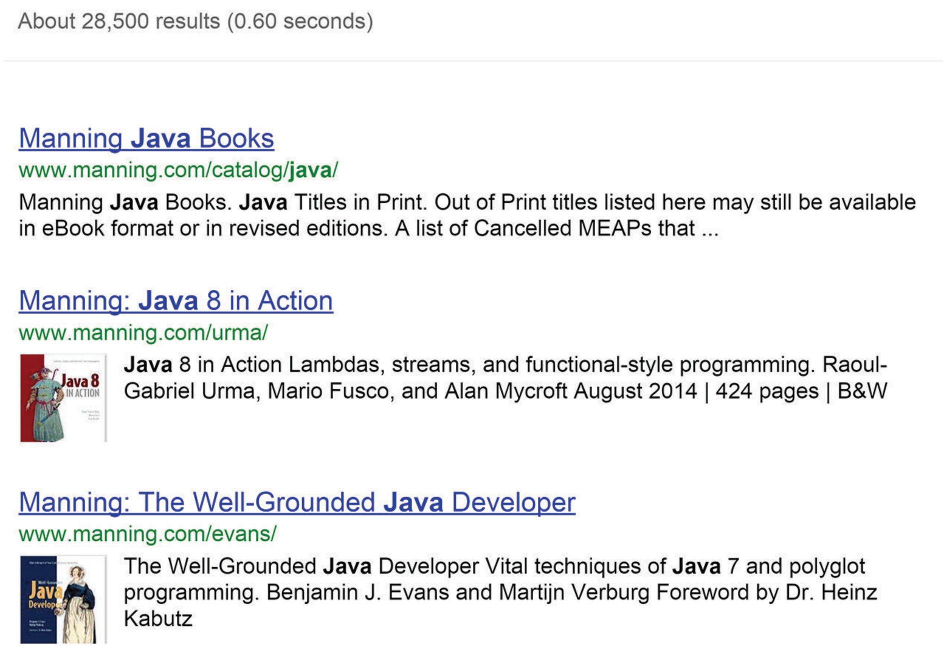
NOTE You can read more about Elasticsearch in *Elasticsearch in Action* by Radu Gheorghe et al (Manning Publications, 2015). You can also read a book on another popular dedicated search engine also built on top of Apache Lucene: *Solr in Action*, by Trey Grainger and Timothy Potter (Manning Publications, 2014).

9.1 Text searches—not just pattern matching

You probably perform some type of search on a daily basis, if not many times every day. As a programmer, you may search the internet for help dealing with particularly vexing programming bugs. You may then go home at night and search Amazon or another website for products; you may have even used the custom search on Manning.com, supported by Google, to find this book.

If you go toManning.com, you'll see a "Search manning.com" text search box in the upper-right corner of the site. Type a keyword, such as "java," into the text box and click the Search button; you'll see something like the display shown in figure 9.1.

Note that since the search is run against live data, your exact results may vary. Perhaps the book *Java 8 in Action*, newly published at the time this chapter was written, will be replaced with Java 9, 10, or even 11.


About 28,500 results (0.60 seconds)

Manning **Java** Books
www.manning.com/catalog/**java**/
Manning **Java** Books. **Java** Titles in Print. Out of Print titles listed here may still be available in eBook format or in revised editions. A list of Cancelled MEAPs that ...

Manning: **Java** 8 in Action
www.manning.com/urma/
Java 8 in Action Lambdas, streams, and functional-style programming. Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft August 2014 | 424 pages | B&W

Manning: The Well-Grounded **Java** Developer
www.manning.com/evans/
The Well-Grounded **Java** Developer Vital techniques of **Java** 7 and polyglot programming. Benjamin J. Evans and Martijn Verburg Foreword by Dr. Heinz Kabutz

Figure 9.1 Search results from search for term "java" at www.manning.com

The point of this search is to illustrate a couple of important features that text search engines provide that you may take for granted:

- The search has performed a *case-insensitive search*, meaning that no matter how you capitalize the letters in your search term, even using "jAVA" instead of "Java" or "java," you'll see results for "Java" or any uppercase, lowercase combination spelling of the word.
- You won't see any results for "JavaScript," even though books on JavaScript contain the text string "Java." This is because the search engine recognizes that there's a difference between the words "Java" and "JavaScript."

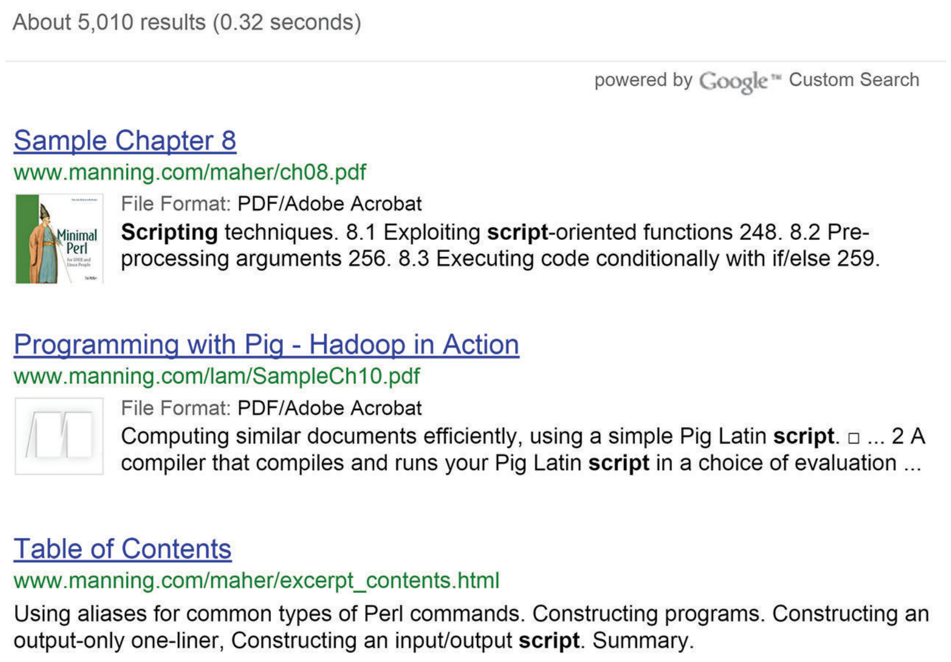
As you may know, you could perform this type of search in MongoDB using a regular expression, specifying whole word matches only and case-insensitive matches. But in MongoDB, such pattern-matching searches can be slow when used on large collections if they can't take advantage of indexes, something text search engines routinely do to sift through large amounts of data. Even those complex MongoDB searches won't provide the capabilities of a true text search.

Let's illustrate that using another example.

9.1.1 Text searches vs. pattern matching

Now try a second search on Manning.com; this time use the search term "script." You should see something similar to the results shown in figure 9.2.

Notice that in this case the results will include results for books that contain the word "scripting" as well as the word "script," but not the word "JavaScript." This is due


About 5,010 results (0.32 seconds)

powered by Google™ Custom Search

Sample Chapter 8
www.manning.com/maher/ch08.pdf
File Format: PDF/Adobe Acrobat
Scripting techniques. 8.1 Exploiting **script**-oriented functions 248. 8.2 Pre-processing arguments 256. 8.3 Executing code conditionally with if/else 259.

Programming with Pig - Hadoop in Action
www.manning.com/lam/SampleCh10.pdf
File Format: PDF/Adobe Acrobat
Computing similar documents efficiently, using a simple Pig Latin **script**. □ ... 2 A compiler that compiles and runs your Pig Latin **script** in a choice of evaluation ...

Table of Contents
www.manning.com/maher/excerpt_contents.html
Using aliases for common types of Perl commands. Constructing programs. Constructing an output-only one-liner, Constructing an input/output **script**. Summary.

Figure 9.2 Results from searching for term "script" on www.manning.com

to the ability of search engines to perform what's known as *stemming*, where words in both the text being searched, as well as the search terms you entered, are converted to the "stem" or root word from which "scripting" is derived—"script" in this case. This is where search engines have to understand the language in which they're storing and searching in order to understand that "script" could refer to "scripts," "scripted," or "scripting," but not "JavaScript."

Although web page searches use many of the same text search capabilities, they also provide additional searching capabilities. Let's see what those search capabilities are as well as how they might help or hinder your user.

9.1.2 Text searches vs. web page searches

Web page search engines contain many of the same search capabilities as a dedicated text search engine and usually much more. Web page searches are focused on searching a network of web pages. This can be an advantage when you're trying to search the World Wide Web, but it may be overkill or even a disadvantage when you're trying to search a product catalog. This ability to search based on relationships between documents isn't something you'll find in dedicated text search engines, nor will you find it in MongoDB, even with the new text search capabilities.

One of the original search algorithms used by Google was referred to as "Page Rank," a play on words, because not only was it intended to rank web pages, but it was developed by the co-founder of Google, Larry Page. Page Rank rates the importance, or weight, of a page based on the importance of pages that link to it. Figure 9.3, based

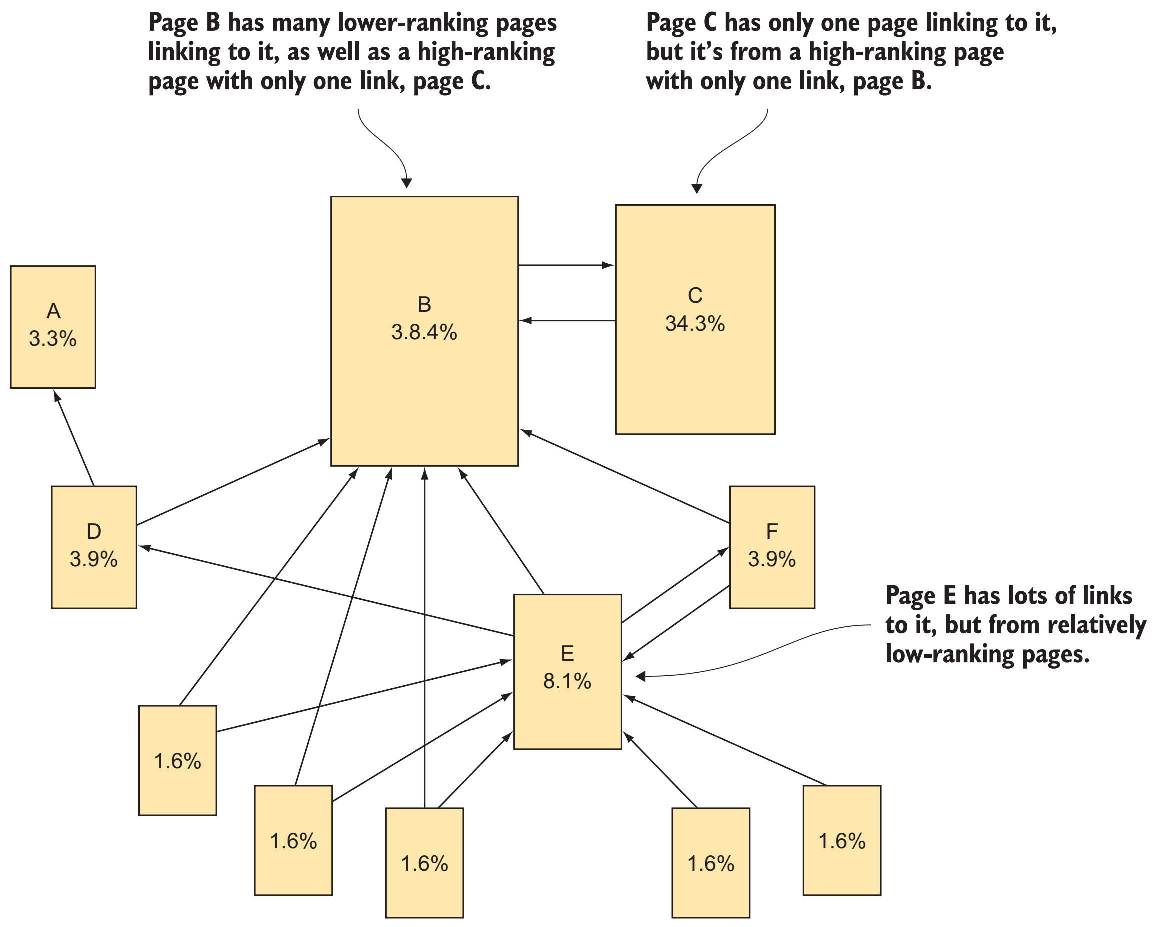

Page B has many lower-ranking pages linking to it, as well as a high-ranking page with only one link, page C.
Page C has only one page linking to it, but it's from a high-ranking page with only one link, page B.
A 3.3%
B 3.8.4%
C 34.3%
D 3.9%
F 3.9%
E 8.1%
Page E has lots of links to it, but from relatively low-ranking pages.
1.6%
1.6%
1.6%
1.6%
1.6%

Figure 9.3 Page ranking based on importance of pages linking to a page

on the Wikipedia entry for Page Rank, http://en.wikipedia.org/wiki/PageRank, illustrates this algorithm.

As you can see in figure 9.3, page C is almost as important as B because it has a very important page pointing to it: page B. The algorithm, which is still taught in university courses on data mining, also takes into account the number of outgoing links a page has. In this case, not only is B very important, but it also has only one outgoing link, making that one link even more critical. Note also that page E has lot of links to it, but they're all from relatively low-ranking pages, so page E doesn't have a high rating.

Google today uses many algorithms to weight pages, over 200 by some counts, making it a full-featured web search engine. But keep in mind that web page searching isn't the same as the type of search you might want to use when searching a catalog. Web page searches will access the web pages you generate from your database, but not the database itself. For example, look again at the page that searched for "java," shown in figure 9.4. You'll see that the first result isn't a product at all—it's the list of Manning books on Java.

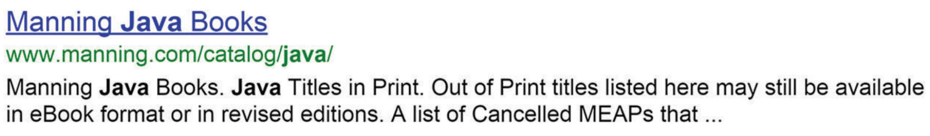
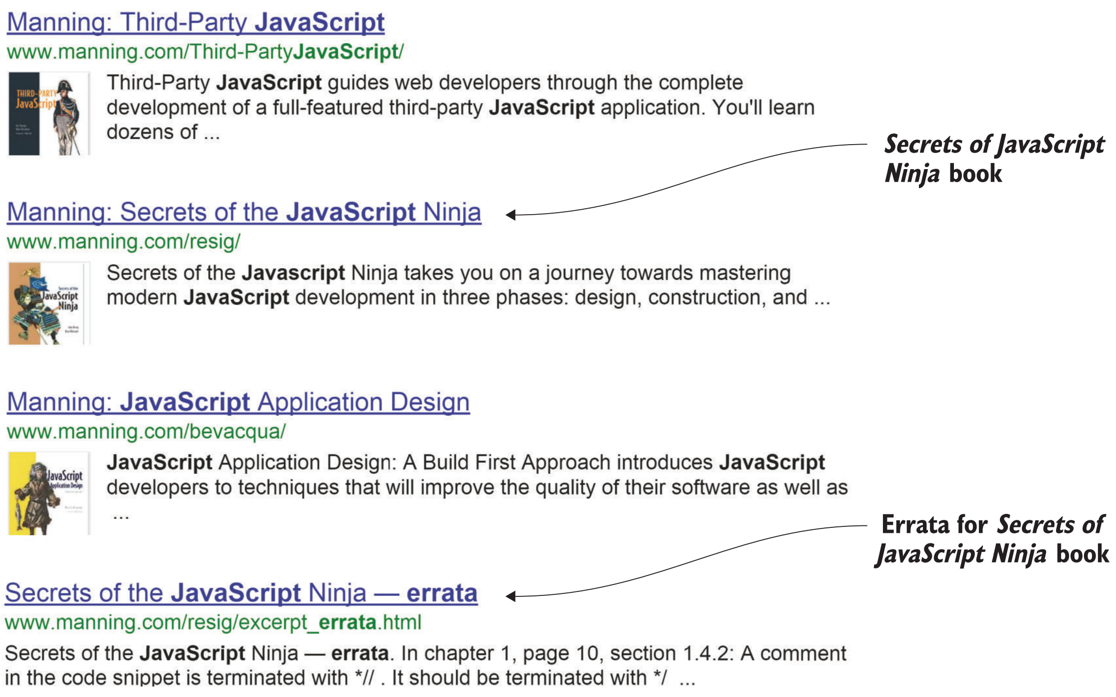

Manning **Java** Books
www.manning.com/catalog/**java**/
Manning **Java** Books. **Java** Titles in Print. Out of Print titles listed here may still be available in eBook format or in revised editions. A list of Cancelled MEAPs that ...

Figure 9.4 Searching results in more than just books.

Perhaps having a list of Java books as the first result might not be so bad, but because the Google search doesn't have the concept of a book, if you search for "javascript," you don't have to scroll down very far before you'll see a web page for errata for a book already in the list. This is illustrated in figure 9.5. This type of "noise" can be distracting if what you're looking for is a book on JavaScript. It can also require you to scroll down further than you might otherwise have to.


Manning: Third-Party **JavaScript**
www.manning.com/Third-Party**JavaScript**/
Third-Party **JavaScript** guides web developers through the complete development of a full-featured third-party **JavaScript** application. You'll learn dozens of ...

Manning: Secrets of the **JavaScript** Ninja
www.manning.com/resig/
Secrets of the **Javascript** Ninja takes you on a journey towards mastering modern **JavaScript** development in three phases: design, construction, and ...

Manning: **JavaScript** Application Design
www.manning.com/bevacqua/
JavaScript Application Design: A Build First Approach introduces **JavaScript** developers to techniques that will improve the quality of their software as well as ...

Secrets of the **JavaScript** Ninja — **errata**
www.manning.com/resig/excerpt_**errata**.html
Secrets of the **JavaScript** Ninja — **errata**. In chapter 1, page 10, section 1.4.2: A comment in the code snippet is terminated with *// . It should be terminated with */ ...

Secrets of JavaScript Ninja book
Errata for *Secrets of JavaScript Ninja* book

Figure 9.5 A search showing how a book can appear more than once

Although web page search engines are great at searching a large network of pages and ranking results based on how the pages are related, they aren't intended to solve the problem of searching a database such as a product database. To solve this type of problem, you can look to full-featured text search engines that can search a product database, such as the one you'd expect to find on Amazon.

9.1.3 MongoDB text search vs. dedicated text search engines

Dedicated text search engines can go beyond indexing web pages to indexing extremely large databases. Text search engines can provide capabilities such as spelling correction, suggestions as to what you're looking for, and relevancy measures—things many web search engines can do as well. But dedicated search engines can provide further improvements such as facets, custom synonym libraries, custom stemming algorithms, and custom stop word dictionaries.

Facets? Synonym libraries? Custom stemming? Stop word dictionaries?

If you've never looked into dedicated search engines, you might wonder what all these terms mean. In brief: *facets* allow you to group together products by a particular characteristic, such as the "Laptop Computer" category shown on the left side of the page in figure 9.6. *Synonym libraries* allow you to specify different words that have the same meaning. For example, if you search for "intelligent" you might also want to see results for "bright" and "smart." As previously covered in section 9.1.1, *stemming* allows you to find different forms of a word, such as "scripting" and "script." *Stop words* are common words that are filtered out prior to searching, such as "the," "a," and "and."

We won't cover these terms in great depth, but if you want to find out more about them you can read a book on dedicated search engines such as *Solr in Action* or *Elasticsearch in Action*.

Faceted search is something that you'll see almost any time you shop on a modern large e-commerce website, where results will be grouped by certain categories that allow the user to further explore. For example, if you go to the Amazon website and search using the term "apple" you'll see something like the page in figure 9.6.

On the left side of the web page, you'll see a list of different groupings you might find for Apple-related products and accessories. These are the results of a faceted search. Although we did provide similar capabilities in our e-commerce data model using categories and tags, facets make it easy and efficient to turn almost any field into a type of category. In addition, facets can go beyond groupings based on the different values in a field. For example, in figure 9.6 you see groupings based on weight ranges instead of exact weight. This approach allows you to narrow the search based on the weight range you want, something that's important if you're searching for a portable computer.

Facets allow the user to easily drill down into the results to help narrow their search results based on different criteria of interest to them. Facets in general are a tremendous aid to help you find what you're looking for, especially in a product database as large as Amazon, which sells more than 200 million products. This is where a faceted search becomes almost a necessity.

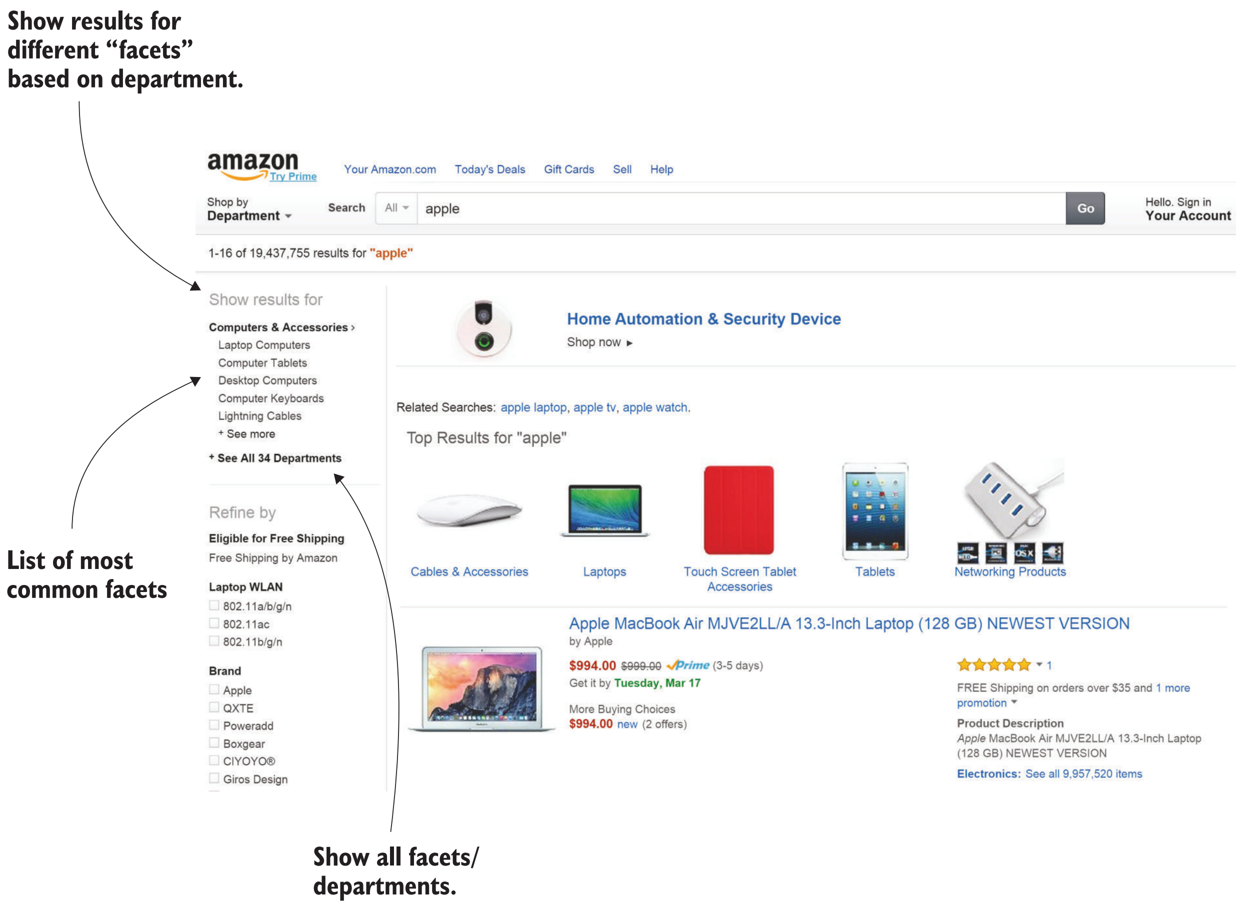

Show results for different "facets" based on department.
List of most common facets
Show all facets/ departments.
amazon Try Prime
Your Amazon.com Today's Deals Gift Cards Sell Help
Shop by Department
Search All apple Go
Hello. Sign in Your Account
1-16 of 19,437,755 results for "apple"
Show results for
Computers & Accessories ›
Laptop Computers
Computer Tablets
Desktop Computers
Computer Keyboards
Lightning Cables
+ See more
+ See All 34 Departments
Refine by
Eligible for Free Shipping
Free Shipping by Amazon
Laptop WLAN
802.11a/b/g/n
802.11ac
802.11b/g/n
Brand
Apple
QXTE
Poweradd
Boxgear
CIYOYO®
Giros Design
Home Automation & Security Device
Shop now ▸
Related Searches: apple laptop, apple tv, apple watch.
Top Results for "apple"
Cables & Accessories
Laptops
Touch Screen Tablet Accessories
Tablets
Networking Products
Apple MacBook Air MJVE2LL/A 13.3-Inch Laptop (128 GB) NEWEST VERSION
by Apple
$994.00 $999.00 Prime (3-5 days)
Get it by Tuesday, Mar 17
More Buying Choices
$994.00 new (2 offers)
1
FREE Shipping on orders over $35 and 1 more promotion
Product Description
Apple MacBook Air MJVE2LL/A 13.3-Inch Laptop (128 GB) NEWEST VERSION
Electronics: See all 9,957,520 items

Figure 9.6 Search on Amazon using the term "apple" and illustrating the use of faceted search

MongoDB's text search: costs vs. benefits

Unfortunately, many of the capabilities available in a full-blown text search engine are beyond the capabilities of MongoDB. But there's good news: MongoDB can still provide you with about 80% of what you might want in a catalog search, with less complexity and effort than is needed to establish a full-blown text search engine with faceted search and suggestive terms. What does MongoDB give you?

- Automatic real-time indexing with stemming
- Optional assignable weights by field name
- Multilanguage support
- Stop word removal
- Exact phrase or word matches
- The ability to exclude results with a given phrase or word

NOTE Unlike more full-featured text search engines, MongoDB doesn't allow you to edit the list of stop words. There's a request to add this: https://jira.mongodb.org/browse/SERVER-10062.

All these capabilities are available for the price of defining an index, which then gives you access to some decent word-search capabilities without having to copy your entire database to a dedicated search engine. This approach also avoids the additional administrative and management overhead that would go along with a dedicated search engine. Not a bad trade-off if MongoDB gives you enough of the capabilities you need.

Now let's see the details of how MongoDB provides this support. It's pretty simple:

- First, you define the indexes needed for text searching.
- Then, you'll use text search in both the basic queries as well as aggregation framework.

One more critical component you'll need is MongoDB 2.6 or later. MongoDB 2.4 introduced text search in an experimental stage, but it wasn't until MongoDB 2.6 that text search became available by default and text search–related functions became fully integrated with the `find()` and `aggregate()` functions.

What you'll need to know to use text searching in MongoDB

Although it will help to fully understand chapter 8 on indexing, the text search indexes are fairly easy to understand. If you want to use text search for basic queries or the aggregation framework, you'll have to be familiar with the related material in chapter 5, which covers how to perform basic queries, and chapter 6, which covers how to use the aggregation framework.

MONGODB TEXT SEARCH: A SIMPLE EXAMPLE

Before taking a detailed look at how MongoDB's text search works, let's explore an example using the e-commerce data. The first thing you'll need to do is define an index; you'll begin by specifying the fields that you want to index. We'll cover the details of using text indexes in section 9.3, but here's a simple example using the e-commerce `products` collection:

```
db.products.createIndex(
    {name: 'text',
     description: 'text',
     tags: 'text'}
);
```

Index name field

Index description field

Index tags field

This index specifies that the text from three fields in the `products` collection will be searched: `name`, `description`, and `tags`. Now let's see a search example that looks for `gardens` in the `products` collection:

```
> db.products
    .find({$text: {$search: 'gardens'}},
          {_id:0, name:1,description:1,tags:1})
    .pretty()
```

Search for text field gardens

```
{
    "name" : "Rubberized Work Glove, Black",
    "description" : "Black Rubberized Work Gloves...",
    "tags" : [
        "gardening"                      <-- gardening matches search
    ]
}
{
    "name" : "Extra Large Wheel Barrow",
    "description" : "Heavy duty wheel barrow...",
    "tags" : [
        "tools",
        "gardening",                     <-- gardening matches search
        "soil"
    ]
}
```

Even this simple query illustrates a few key aspects of MongoDB text search and how it differs from normal text search. In this example, the search for *gardens* has resulted in a search for the stemmed word *garden*. That in turn has found two products with the tag `gardening`, which has been stemmed and indexed under `garden`.

In the next few sections, you'll learn much more about how MongoDB text search works. But first let's download a larger set of data to use for the remaining examples in this chapter.

9.2 Manning book catalog data download

Our e-commerce data has been fine for the examples shown so far in the book. For this chapter, though, we're going to introduce a larger set of data with much more text in order to better illustrate the use of MongoDB text search and its strengths as well as limitations. This data set will contain a snapshot of the Manning book catalog created at the time this chapter was written. If you want to follow along and run examples yourself, you can download the data to your local MongoDB database by following these steps:

- In the source code included with this book, find the chapter9 folder, and copy the file catalog.books.json from that folder to a convenient location on your computer.
- Run the command shown here. You may have to change the command to prefix the filename, catalog.books.json, with the name of the directory where you saved the file.

```
mongoimport --db catalog --collection books --type json --drop
➥   --file catalog.books.json
```

You should see something similar to the results shown in the following listing. Please note that the `findOne()` function returns a randomly selected document.

Listing 9.1 Loading sample data in the `books` collections

```
> use catalog                                    ◁ Switch to catalog
switched to db catalog                             database
> db.books.findOne()                             ◁
{                                                  Show a randomly selected
        "_id" : 1,                                 book from catalog
nerat        "title" : "Unlocking Android",
        "isbn" : "1933988673",
        "pageCount" : 416,
        "publishedDate" : ISODate("2009-04-01T07:00:00Z"),
        "thumbnailUrl" : "https://s3.amazonaws.com/AKIAJC5RLADLUMVRPFDQ
.book-thumb-images/ableson.jpg",
        "shortDescription" : "Unlocking Android: A Developer's Guide
provides concise, hands-on instruction for the Android operating system and
development tools. This book teaches important architectural concepts in a
straightforward writing style and builds on this with practical and useful
examples throughout.",
        "longDescription" : "Android is an open source mobile phone
platform based on the Linux operating system and developed by the Open
Handset Alliance, a consortium of over 30 hardware, software and telecom
…
* Notification methods     * OpenGL, animation & multimedia     * Sample
     "status" : "PUBLISH",
        "authors" : [
                "W. Frank Ableson",
                "Charlie Collins",
                "Robi Sen"
        ],
        "categories" : [
                "Open Source",
                "Mobile"
        ]
}
```

The listing also shows the structure of a document. For each document you'll have the following:

- `title`—A text field with the book title
- `isbn`—International Standard Book Number (ISBN)
- `pageCount`—The number of pages in the book
- `publishedDate`—The date on which the book was published (only present if the `status` field is `PUBLISH`)
- `thumbnailUrl`—The URL of the thumbnail for the book cover
- `shortDescription`—A short description of the book
- `longDescription`—A long description of the book
- `status`—The status of the book, either `PUBLISH` or `MEAP`
- `authors`—The array of author names
- `categories`—The array of book categories

Now that you have the list of books loaded, let's create a text index for it.

9.3 Defining text search indexes

Text indexes are similar to the indexes you saw in section 7.2.2, which covered creating and deleting indexes. One important difference between the regular indexes you saw there and text indexes is that you can have only a single text index for a given collection. The following is a sample text index definition for the books collection:

```
db.books.createIndex(
    {title: 'text',
     shortDescription: 'text',
     longDescription: 'text',
     authors: 'text',
     categories: 'text'},

    {weights:
        {title: 10,
         shortDescription: 1,
         longDescription:1,
         authors: 1,
         categories: 5}
     }
);
```

Specify fields to be text-indexed.

Optionally specify weights for each field.

There are a few other important differences between the regular indexes covered in section 7.2.2 and text indexes:

- Instead of specifying a `1` or `-1` after the field being indexed, you use `text`.
- You can specify as many fields as you want to become part of the text index and all the fields will be searched together as if they were a single field.
- You can have only one text search index per collection, but it can index as many fields as you like.

Don't worry yet about weights assigned to the fields. The weights allow you to specify how important a field is to scoring the search results. We'll discuss that further and show how they're used when we explore text search scoring in section 9.4.2.

9.3.1 Text index size

An index entry is created for each unique, post-stemmed word in the document. As you might imagine, text search indexes tend to be large. To reduce the number of index entries, some words (called *stop words*) are ignored. As we discussed earlier when we talked about faceted searches, stop words are words that aren't generally searched for. In English this include words such as "the," "an," "a," and "and." Trying to perform a search for a stop word would be pretty useless because it would return almost every document in your collection.

The next listing shows the results of a `stats()` command on our `books` collection. The `stats()` command shows you the size of the `books` collection, along with the size of indexes on the collection.

Listing 9.2 `books` collection statistics showing space use and index name

```
> db.books.stats()
{
        "ns" : "catalog.books",
        "count" : 431,
        "size" : 772368,
        "avgObjSize" : 1792,
        "storageSize" : 2793472,
        "numExtents" : 5,
        "nindexes" : 2,
        "lastExtentSize" : 2097152,
        "paddingFactor" : 1,
        "systemFlags" : 0,
        "userFlags" : 1,
        "totalIndexSize" : 858480,
        "indexSizes" : {
                "_id_" : 24528,

"title_text_shortDescription_text_longDescription_text_authors_text
_categories_text" : 833952
         },
        "ok" : 1
}
```

Size of books collection

Name and size of text search index

Notice that the size of the `books` collection (`size` in listing 9.2) is 772,368. Looking at the `indexSizes` field in the listing, you'll see the name and size of the text search index. Note that the size of the text search index is 833,952—larger than the `books` collection itself! This might startle or concern you at first, but remember the index must contain an index entry for each unique stemmed word being indexed for the document, as well as a pointer to the document being indexed. Even though you remove stop words, you'll still have to duplicate most of the text being indexed as well as add a pointer to the original document for each word.

Another important point to take note of is the length of the index name:

```
"title_text_shortDescription_text_longDescription_text_authors_text
_categories_text."
```

MongoDB namespaces have a maximum length of 123 bytes. If you index a few more text fields, you can see how you might easily exceed the 123-byte limit. Let's see how you can assign an index a user-defined name to avoid this problem. We'll also show you a simpler way to specify that you want to index all text fields in a collection.

9.3.2 Assigning an index name and indexing all text fields in a collection

In MongoDB a *namespace* is the name of an object concatenated with the name of the database and collection, with a dot between the three names. Namespaces can have a maximum length of 123 bytes. In the previous example, you're already up to 84 characters for the namespace for the index.

There are a couple of ways to avoid this problem. First, as with all MongoDB indexes, you have the option of specifying the name of the index, as shown here:

```
db.books.createIndex(
    {title: 'text',
     shortDescription: 'text',
     longDescription: 'text',
     authors: 'text',
     categories: 'text'},

    {weights:                                    <-- Specify weights for
        {title: 10,                                  fields with weights
         categories: 5},                             other than 1.

     name : 'books_text_index'                   <-- User-defined
    }                                                index name
);
```

This example also specifies weights for `title` and `categories`, but all other fields will default to a weight of 1. You'll find out more about how weights affect the sorting of your search results in section 9.4.3 when we cover sorting by text search score.

Please note that if an index already exists, you won't be able to create it again even if you're using a different name (the error message will be `"all indexes already exist"`). In that case, you'll first need to drop it using `dropIndex()` and then recreate it with the desired name.

WILDCARD FIELD NAME

Text search indexes also have a special wildcard field name: `$**`. This name specifies that you want to index any field that contains a string. For text indexes with the wildcard specification, the default index name is `$**_text`, thus enabling you to avoid the namespace 123-byte limit problem:

```
db.books.createIndex(
    {'$**': 'text'},                    <-- Index all fields
                                            with strings.
    {weights:
        {title: 10,
         categories: 5},
);
```

You can also include other fields in the text index to create a compound index, but there are some restrictions on how you can search a compound text index. You can read more about this and other details of the text index at http://docs.mongodb.org/manual/core/index-text/.

Now that you have a text index, let's see how to use it for searching.

9.4 Basic text search

Let's start with an example of a simple MongoDB text search:

```
db.books.find({$text: {$search: 'actions'}},{title:1})
```

This query looks much like the queries covered in chapter 5 using the `find()` command. The `$text` operator defines the query as a text search. The `$search` parameter then defines the string you want to use for the search. This query would return these results or something similar as results are returned in a random order:

```
{ "_id" : 256, "title" : "Machine Learning in Action" }
{ "_id" : 146, "title" : "Distributed Agile in Action" }
{ "_id" : 233, "title" : "PostGIS in Action" }
{ "_id" : 17, "title" : "MongoDB in Action" }
...
```

Even for this simple query there's quite a bit going on under the covers:

- The word *actions* was stemmed to *action.*
- MongoDB then used an index to quickly find all documents with the stemmed word *action.*

Although not noticeable on our relatively small collection, you can see how using an index to find the documents instead of scanning all the text fields for all the documents in the collection can be much faster even for modest-sized collections.

Next, try a more complex search, one using a phrase with more than one word:

```
db.books.find({$text: {$search: 'MongoDB in Action'}},{title:1})
```

So far the results will appear the same as for the previous example:

```
{ "_id" : 256, "title" : "Machine Learning in Action" }
{ "_id" : 146, "title" : "Distributed Agile in Action" }
{ "_id" : 233, "title" : "PostGIS in Action" }
{ "_id" : 17, "title" : "MongoDB in Action" }
...
```

For this query, the search string is split into words, stop words are removed, the remaining words are stemmed, and MongoDB then uses the text index to perform a case-insensitive compare. This is illustrated in figure 9.7.

In the figure, there's only one stop word, *in*, and the stemmed versions of each word are the same as the original word. MongoDB will next use the results to perform a case-insensitive search using the text index twice: once to search for *mongodb*, and then again to search for *action.* The results will be any documents that contain either of the two words, the equivalent of an or search.

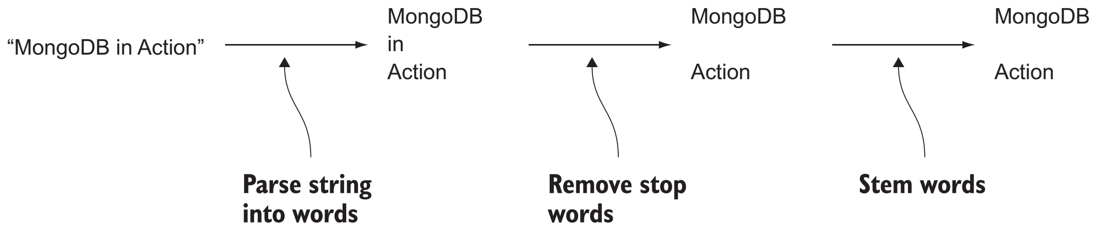


Figure 9.7 Text search string processing

Now that you've seen the basics of simple text searching, let's move on to more advanced searches.

9.4.1 More complex searches

In addition to searching for any of a number of words, the equivalent of an *or* search, MongoDB search allows you to do the following:

- Specify *and* word matches instead of *or* word matches.
- Perform exact phrase matches.
- Exclude documents with certain words.
- Exclude documents with certain phrases.

Let's start by seeing how to specify that a given word must be in the result document. You've already seen a search for *mongodb in action*, which returned not only books on MongoDB, but also any book with the word *action*. If you enclose a word in double quotes within the search string, it specifies that the word must always be in the result document. Here's an example:

```
db.books.
    find({$text: {$search: ' "mongodb" in action'}})
```

"mongodb" in double quotes means the word must be present.

This query returns only the books with titles that include the word *mongodb*:

```
{ "title" : "MongoDB in Action"}
{ "title" : "MongoDB in Action, Second Edition" }
```

Exact match on phrases

Using double quotes also works for phrases, so if you specify the phrase *second edition*, only the second edition book is shown because multiple phrases make it an "and" search:

```
> db.books.
...     find({$text: {$search: ' "mongodb" "second edition" '}},
...     {_id:0, title:1})
{ "title" : "MongoDB in Action, Second Edition" }
```

Phrase "second edition" required as well as the word "mongodb"

Although the exact match logic will perform a case-insensitive compare, it won't remove stop words, nor will it stem the search terms. You can illustrate this by searching using the search string `'books'` with and without double quotes:

```
> db.books.
...     find({$text: {$search: ' books '}}).
...     count()
414
>
> db.books.
...     find({$text: {$search: ' "books" '}}).
...     count()
21
```

Stemmed version of word "books"—414

Exact word "books"—21

Here you can see that when you specified the word *books* without double quotes, MongoDB stemmed the word and could find 414 results. When you specify the exact match, using double quotes around the word *books*, MongoDB returned only the count of documents that contained the exact word *books*, 21 documents in all. The total number of results you will get may vary depending on the input data.

EXCLUDING DOCUMENTS WITH SPECIFIC WORDS OR PHRASES

To exclude all documents that contain a word, put a minus sign in front of the word. For example, if you wanted all books with the word *MongoDB* but not those with the word *second* you could use the following:

```
> db.books.
...     find({$text: {$search: ' mongodb -second '}},
...     {_id:0, title:1 })
{ "title" : "MongoDB in Action" }
```

Exclude documents with the word "second."

Note that the three dots on the second and the third lines are automatically added by the mongo shell to show that the input is longer than one line—in this case it's three lines long.

Similarly, you can exclude documents with a particular phrase by enclosing the phrase in double quotes and preceding it with a minus sign:

```
> db.books.
...     find({$text: {$search: ' mongodb -"second edition" '}},
...     {_id:0, title:1})
{ "title" : "MongoDB in Action" }
```

Exclude documents with the phrase "second edition."

MORE COMPLEX SEARCH SPECIFICATIONS

You can combine the text search with most other `find()` search criteria to further limit your search. For example, if you wanted to search for all Java books that still have a status of MEAP, you could use this:

```
> db.books.
...     find({$text: {$search: ' mongodb '}, status: 'MEAP' },
...     {_id:0, title:1, status:1})
{ "title" : "MongoDB in Action, Second Edition",
 "status" : "MEAP"}
```

status must be MEAP

Limits on combining text search criteria

There are a few limits as to what you can combine with a text search and how text indexes are limited. These limits are further defined at http://docs.mongodb.org/manual/core/index-text/ under restrictions. A few key limits include the following:

- Multikey compound indexes aren't allowed.
- Geospatial compound key indexes aren't allowed.
- `hint()` cannot be used if a query includes a `$text` query expression.
- Sort operations cannot obtain sort order from a text field index.

If you add more stop words, such as *the*, and change the search terms to use nonstemmed words, you'll see the same results. This doesn't prove that the stop words are in fact ignored or that the nonstemmed word is treated the same as the stemmed word.

To prove that, you'll have to look at the text search score to confirm that you're receiving the same score regardless of extra stop words or different words with the same stem. Let's see what the text search score is and how you can include it in your results.

9.4.2 Text search scores

The text search score provides a number that rates the relevancy of the document based on how many times the word appeared in the document. The scoring also uses any weights assigned to different fields when the index was created, as described in section 9.3.

To show the text search score, you use a projection field such as `score: { $meta: "textScore" }` in your `find()` command. Note that the name you assign the text score—`score` in this case—can be any name you want. The next listing shows an example of the same search shown earlier but with the text score displayed, followed by the search with a slightly different but equivalent search string. Please note that the output you're going to get may be different from the one presented here.

Listing 9.3 Displaying text search score

```
> db.books.
...     find({$text: {$search: 'mongodb in action'}}),
...     {_id:0, title:1, score: { $meta: "textScore" }}).
...     limit(4);
{ "title" : "Machine Learning in Action", "score" : 16.83933933933934 }
{ "title" : "Distributed Agile in Action", "score" : 19.371088861076345 }
{ "title" : "PostGIS in Action", "score" : 17.67825896762905 }
{ "title" : "MongoDB in Action", "score" : 49.48653394500073 }
>
>
> db.books.
...     find({$text: {$search: 'the mongodb and actions in it'}}),
...     {_id:0, title:1, score: { $meta: "textScore" }}).
...     limit(4);
{ "title" : "Machine Learning in Action", "score" : 16.83933933933934 }
{ "title" : "Distributed Agile in Action", "score" : 19.371088861076345 }
{ "title" : "PostGIS in Action", "score" : 17.67825896762905 }
{ "title" : "MongoDB in Action", "score" : 49.48653394500073 }
```

Search for "Mongodb in Action."

Include text search score in results.

Text search scores for first search string

Second string text scores—identical to first set of scores

Second text string with extra stop words and plural word "actions"

In this listing, the search string "MongoDB in Action" is changed to "the mongodb and actions in it." This new search string uses the plural form of *action*, and also adds a number of stop words. As you can see, the text scores are identical in both cases, illustrating that the stop words are in fact ignored and that the remaining words are stemmed.

WEIGHT FIELD TO INFLUENCE WORD IMPORTANCE

In the index created in section 9.3.2, you'll notice the definition of a field called weights. Weights influence the importance of words found in a particular field, compared to the same word found in other fields. The default weight for a field is 1, but as you can see, we've assigned a weight of 5 to the categories field and a weight of 10 to the title field. This means that a word found in categories carries five times the weight of the same word found in the short or long description fields. Similarly, a word found in the title field will carry 10 times the weight of the same word found in one of the description fields and twice the weight of the same word found in categories. This will affect the score assigned to a document:

```
db.books.createIndex(
    {'$**': 'text'},

    {weights:
        {title: 10,
         categories: 5}
    }
);
```

Specify weights for fields with weights other than 1.

This search is fine if you want to find all the books with the words *mongodb* or *action*. But for most searches, you also want to view the most relevant results first. Let's see how to do that.

9.4.3 Sorting results by text search score

To sort the results by relevancy, sort by the same text search score shown in the previous example. In fact, to sort by the text search score, you must also include the $meta function in your find() projection specification. Here's an example:

```
db.books.
    find({$text: {$search: 'mongodb in action'}},
         {title:1, score: { $meta: "textScore" }}).
    sort({ score: { $meta: "textScore" } })
```

Projection for text score

Sort by text score.

This example will result in a list sorted by the text score:

```
{ "_id" : 17, "title" : "MongoDB in Action", "score" : 49.48653394500073 }
{ "_id" : 186, "title" : "Hadoop in Action", "score" : 24.99910329985653 }
{ "_id" : 560, "title" : "HTML5 in Action", "score" : 23.02156177156177 }
```

As mentioned earlier, you can name the text search score anything you want. We've named it `score` in our examples, but you may choose something like `textSearchScore`. But keep in mind that the name specified in the `sort()` function must be the same as the name specified in the preceding `find()` function. In addition, you can't specify the order (ascending or descending) for the sort by text sort field. The sort is always from highest score to lowest score, which makes sense, because you normally want the most relevant results first. If for some reason you do need to sort with least relevant results first, you can use the aggregation framework text search (which is covered in the next section).

Now that you've seen how to use the text search with the `find()` command, let's see how you can also use it in the aggregation framework.

The projection field `$meta:"textScore"`

As you learned in chapter 5, section 5.1.2, you use a projection to limit the fields returned from the `find()` function. But if you specify any fields in the find projection, only those fields specified will be returned.

You can only sort by the text search score if you include the text search meta score results in your projection. Does this mean you must always specify all the fields you want returned if you sort by the text search score?

Luckily, no. If you specify only the meta text score in your find projection, all the other fields in your document will also be returned, along with the text search meta score.

9.5 Aggregation framework text search

As you learned in chapter 6, by using the aggregation framework, you can transform and combine data from multiple documents to generate new information not available in any single document. In this section you'll learn how to use the text search capabilities within the aggregation framework. As you'll see, the aggregation framework provides all the text search capabilities you saw for the `find()` command and a bit more.

In section 9.4.3, you saw a simple example in which you found books with the words *mongodb in action* and then sorted the results by the text score:

```
db.books.
    find({$text: {$search: 'mongodb in action'}},
         {title:1, score: { $meta: "textScore" }}).
    sort({ score: { $meta: "textScore" } })
```

Search for documents with the words mongodb or action.

Projection for text score

Sort by text score.

Using the aggregation framework, you can produce the same results using the following code:

```
db.books.aggregate(
   [
     { $match: { $text: { $search: 'mongodb in action' } } },   <-- Search for documents with the words mongodb or action.
     { $sort: { score: { $meta: 'textScore' } } },              <-- Sort by text score.
     { $project: { title: 1, score: { $meta: 'textScore' } } }  <-- Projection for text score
   ]
)
```

As expected, this code will produce the same results you saw in the previous section:

```
{ "_id" : 17, "title" : "MongoDB in Action", "score" : 49.48653394500073 }
{ "_id" : 186, "title" : "Hadoop in Action", "score" : 24.99910329985653 }
{ "_id" : 560, "title" : "HTML5 in Action", "score" : 23.02156177156177 }
{ "_id" : 197, "title" : "Erlang and OTP in Action", "score" :
22.069632021922096 }
```

Notice that the two versions of the text search use many of the same constructs to specify the find/match criteria, the projection attributes, and the sort criteria. But as we promised, the aggregation framework can do even more. For example, you can take the previous example and by swapping the `$sort` and `$project` operators, simplify the `$sort` operator a bit:

```
db.books.aggregate(
    [
        { $match: { $text: { $search: 'mongodb in action' } } },
        { $project: { title: 1, score: { $meta: 'textScore' } } },
        { $sort: { score: -1 } }                                    <-- Sort by descending score.
    ]
)
```

One big difference in the second aggregation example is that, unlike with the `find()` function, you can now reference the `score` attribute you defined in the preceding `$project` operation. Notice, though, that you're sorting the scores in *descending* order, and therefore you're using `score: -1` instead of `score: 1`. But this does provide the option of showing lowest scoring books first if desired by using `score: 1`.

Using the `$text` search in the aggregation framework has some limitations:

- The `$match` operator using `$text` function search must be the first operation in the pipeline and must precede any other references to `$meta:'textScore'`.
- The `$text` function can appear only once in the pipeline.
- The `$text` function can't be used with `$or` or `$not`.

With the `$match` text search string, use the same format you would with the `find()` command:

- If a word or phrase is enclosed in double quotes, the document must contain an exact match of the word or phrase.
- A word or phrase preceded by a minus sign (–) excludes documents with that word or phrase.

In the next section, you'll learn how to use the ability to access the text score to further customize the search.

9.5.1 Where's MongoDB in Action, Second Edition?

If you look closely at the results from our previous text searches using the string `"MongoDB in Action"`, you may have wondered why the results didn't include the second edition of *MongoDB in Action* as well as the first edition. To find out why, use the same search string but enclose `monogdb` in double quotes so that you find only those documents that have the word *mongodb* in them:

```
> db.books.aggregate(
...     [
...         { $match: { $text: { $search: ' "mongodb" in action ' } } },
...         { $project: {_id:0, title: 1, score: { $meta: 'textScore' } } }
...     ]
... )
{ "title" : "MongoDB in Action", "score" : 49.48653394500073 }
{ "title" : "MongoDB in Action, Second Edition", "score" : 12.5 }
```

When you see the low text score for the second edition of *MonogDB in Action*, it becomes obvious why it hasn't shown up in the top scoring matches. But now the question is why the score is so low for the second edition. If you do a find only on the second edition, the answer becomes more obvious:

```
> db.books.findOne({"title" : "MongoDB in Action, Second Edition"})
{
        "_id" : 755,
        "title" : "MongoDB in Action, Second Edition",
        "isbn" : "1617291609",
        "pageCount" : 0,
        "thumbnailUrl" :
"https://s3.amazonaws.com/AKIAJC5RLADLUMVRPFDQ.book-thumb-
images/banker2.jpg",
        "status" : "MEAP",
        "authors" : [
                "Kyle Banker",
                "Peter Bakkum",
                "Tim Hawkins",
                "Shaun Verch",
                "Douglas Garrett"
        ],
        "categories" : [ ]
}
```

As you can see, because this data is from before the second edition was printed, the second edition didn't have the shortDescription or longDescription fields. This is true for many of the books that hadn't yet been published, and as a result those books will end up with a lower score.

You can use the flexibility of the aggregation framework to compensate for this somewhat. One way to do this is to multiply the text search score by a factor—say, 3—if a document doesn't have a longDescription field. The following listing shows an example of how you might do this.

Listing 9.4 Add text multiplier if `longDescription` isn't present

```
> db.books.aggregate(
    ...     [
    ...         { $match: { $text: { $search: 'mongodb in action' } } },
...
...         { $project: {
...             title: 1,
...             score: { $meta: 'textScore' },
...             multiplier: { $cond: [ '$longDescription',1.0,3.0] } }
...         },
...
...         { $project: {
...             _id:0, title: 1, score: 1, multiplier: 1,
...             adjScore: {$multiply: ['$score','$multiplier']}}
...         },
...
...         { $sort: {adjScore: -1}}
...     ]
... );
```

Calculate multiplier: 3.0 if longDescription doesn't exist

Calculate adjusted score: score * multiplier

Sort by descending adjusted score

```
{ "title" : "MongoDB in Action", "score" : 49.48653394500073,
  "multiplier" : 1, "adjScore" : 49.48653394500073 }
{ "title" : "MongoDB in Action, Second Edition", "score" : 12.5,
  "multiplier" : 3, "adjScore" : 37.5 }
{ "title" : "Spring Batch in Action", "score" : 11.666666666666666,
  "multiplier" : 3, "adjScore" : 35 }
{ "title" : "Hadoop in Action", "score" : 24.99910329985653,
  "multiplier" : 1, "adjScore" : 24.99910329985653 }
{ "title" : "HTML5 in Action", "score" : 23.02156177156177,
  "multiplier" : 1, "adjScore" : 23.02156177156177 }
```

Second edition now second on list

As you can see in the first $project operator in the pipeline, you're calculating a multiplier by testing whether longDescription exists. A condition is considered false if it's null or doesn't exist, so you can use the $cond function to set a multiplier of 1.0 if longDescription exists and a multiplier of 3.0 if longDescription doesn't exist.

You then have a second $project operator in the aggregation pipeline that calculates an adjusted score by multiplying the text search score by the multiplier 1.0 or 3.0. Finally, you sort by the adjusted score in descending order.

As you can see, the MongoDB text search does have its limitations. Missing text fields can cause you to miss some results. The MongoDB text search also provides some ways to improve your search by requiring certain words or phrases to be in the search results, or by excluding documents that contain certain words. The aggregation framework offers additional flexibility and functionality and can be useful in extending the value of your text search.

Now that you've seen the basics and a few advanced features of MongoDB text search, you're ready to tackle another complex issue: searching languages other than English.

9.6 Text search languages

Remember that much of MongoDB's text search power comes from being able to stem words. Searching for the word *action* will return the same results as searching for the word *actions*, because they have the same stem. But stemming is language-specific. MongoDB won't recognize the plural or other unstemmed version of a non-English word unless you tell MongoDB what language you're using.

There are three points at which you can tell MongoDB what language you're using:

- *In the index*—You can specify the default language for a particular collection.
- *When you insert a document*—You can override this default to tell MongoDB that a particular document or field within the document is a language other than the index-specified default.
- *When you perform the text search in a* `find()` *or* `aggregate()` *function*—You can tell MongoDB what language your search is using.

Stemming and stop words: Simple but limited

Currently MongoDB uses "simple language-specific suffix stemming" (see http://docs.mongodb.org/manual/core/index-text/). Various stemming algorithms, including suffix stripping, are further described at http://en.wikipedia.org/wiki/Stemming. If you require processing of a language not supported by the suffix stemming approach, such as Chinese, or wish to use a different or customized stemmer, your best bet is to go to a more full-featured text search engine.

Similarly, although MongoDB will use a different stop word dictionary based on the language, it doesn't allow you to customize the stop word dictionaries. Again, this is something that dedicated text search engines typically support.

Let's take a look at how you use each of these options.

9.6.1 Specifying language in the index

Returning to the example index you created in section 9.3.2, you can modify the index definition to define a default language. Before changing the language for the `books` collection, run the following text search command. You should find no results because you're searching for a stop word: *in*. Remember, stop words aren't indexed:

```
> db.books.find({$text: {$search: 'in '}}).count()
0
```

Now delete the previous index and create the same index, but with the language `french`:

```
db.books.dropIndex('books_text_index');

db.books.createIndex(
    {'$**': 'text'},

    {weights:
        {title: 10,
         categories: 5},

     name : 'books_text_index',

     default_language: 'french'
    }
);
```

Drop existing text index on books

Add new index with language french

Now if you rerun the previous `find()`, you'll now find some books, because in French, the word *in* isn't a stop word:

```
> db.books.find({$text: {$search: 'in '}}).count()
334
```

If you check the indexes on the `books` collection, you'll see the language is now French:

```
> db.books.getIndexes()
[
        {
                "v" : 1,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "catalog.books"
        },
        {
                "v" : 1,
                "key" : {
                        "_fts" : "text",
                        "_ftsx" : 1
                },
                "name" : "books_text_index",
                "ns" : "catalog.books",
                "weights" : {
                        "$**" : 1,
                        "categories" : 5,
                        "title" : 10
                },
                "default_language" : "french",
                "language_override" : "language",
                "textIndexVersion" : 2
        }
]
```

Default text index language is French

9.6.2 Specifying the language in the document

Before you insert an example document that specifies the document language, change the index back to English by running the following commands:

```
db.books.dropIndex('books_text_index');

db.books.createIndex(
    {'$**': 'text'},

    {weights:
        {title: 10,
         categories: 5},

     name : 'books_text_index',

     default_language: 'english'
    }
);
```

Specify default language of English

Now insert a new document specifying the language as French:

```
db.books.insert({
    _id: 999,
    title: 'Le Petite Prince',
    pageCount: 85,
    publishedDate:  ISODate('1943-01-01T01:00:00Z'),
    shortDescription: "Le Petit Prince est une œuvre de langue française,
la plus connue d'Antoine de Saint-Exupéry. Publié en 1943 à New York
simultanément en anglais et en français. C'est un conte poétique et
philosophique sous l'apparence d'un conte pour enfants.",
    status: 'PUBLISH',
    authors: ['Antoine de Saint-Exupéry'],
    language: 'french'
})
```

Specify language as 'french'

MongoDB text search also allows you to change the name of the field used to specify the document language when you define the index, if you want to use something other than `language`. You can also specify different parts of the document to be in different languages. You can read more about these features at http://docs.mongodb.org/manual/tutorial/specify-language-for-text-index/.

Now that you've inserted a document in French, let's see how you can search for it in French as well.

9.6.3 Specifying the language in a search

What language your text search string represents can make a big difference in the results. Remember that the language affects how MongoDB interprets your string by defining the stop words as well as stemming. Let's see how the specified language affects both how the document was indexed and how MongoDB performs the search. Our first example, shown in the next listing, shows the effect of stemming on our document indexes as well as on our search terms.

Listing 9.5 Example of how language affects stemming

```
> db.books.find({$text: {$search:
     'simultanment',$language:'french'}},{title:1})
{ "_id" : 999, "title" : "Le Petit Prince" }
```

Language French; only finds "Le Petit Prince"

```
> db.books.find({$text: {$search: 'simultanment'}},{title:1})
{ "_id" : 186, "title" : "Hadoop in Action" }
{ "_id" : 293, "title" : "Making Sense of Java" }
{ "_id" : 999, "title" : "Le Petite Prince" }
```

Same search in English finds two different books

```
> db.books.find({$text: {$search: 'prince'}},{title:1})
{ "_id" : 145, "title" : "Azure in Action" }
{ "_id" : 999, "title" : "Le Petit Prince" }
```

Search for prince in English finds both French and English language books

When you search for `simultanment` and specify the language as French, you find only the French book *Le Petit Prince*. Yet when you do the same search without specifying the language—meaning you use the default language English—you return two completely different books.

How can this be? With just this example, you might assume MongoDB is ignoring any documents that aren't in the specified language. But if you look at the third `find()`, where you search for the word *prince* you see that MongoDB can indeed find books in either French or English.

What's up with this? The answer lies in stemming. When you specify a search string, MongoDB will search for the stemmed words in your search string, not the actual words from the string. A similar process is used for creating the index for the documents where the index will contain the stemmed versions of words in the document, not the words themselves. As a result, the stemmed word MongoDB comes up with for *simultanment* will be different for French and English.

For French, it's easy to see how MongoDB found the one document because the book description contained the word *simultanment*. For the English documents, though, the reason is less clear. The next listing helps clarify the situation a bit and also illustrates some of the limitations of stemming.

Listing 9.6 Results of stemming `simultaneous`

```
> db.books.find({$text: {$search: 'simultaneous'}},{title:1})
{ "_id" : 186, "title" : "Hadoop in Action" }
{ "_id" : 293, "title" : "Making Sense of Java" }
{ "_id" : 999, "title" : "Le Petite Prince" }
```

English search for simultaneous

```
 > db.books.find({$text: {$search: 'simultaneous',
     $language:'french'}},{title:1})
 >
```

French search for simultaneous; nothing found

In this listing you searched for the word *simultaneous* in both English and French. As you expected, when you searched in English, you found the two books previously found when you searched for *simultanment*.

But if you now search for *simultaneous* in French, you won't find the French book. Unfortunately, in this case what MongoDB calculates as the stem word in French isn't the same as the calculated stem word for *simultanment*.

This result can be confusing, and the process of calculating the stem for a word isn't an exact science. But in most cases you'll find what you'll expect.

Fortunately, the effect of language on stop words is much simpler. For stop words, MongoDB can use a dictionary to access a list of known stop words for a given language. As a result, the effect of language on the interpretation of stop words is much clearer. The next listing shows a simple example.

Listing 9.7 Example of how language affects stop words

```
> db.books.find({$text: {$search: 'de'}},{title:1})
{ "_id" : 36, "title" : "ASP.NET 4.0 in Practice" }
{ "_id" : 629, "title" : "Play for Java" }
{ "_id" : 199, "title" : "Doing IT Right" }
{ "_id" : 10, "title" : "OSGi in Depth" }
{ "_id" : 224, "title" : "Entity Framework 4 in Action" }
{ "_id" : 761, "title" : "jQuery in Action, Third Edition" }

> db.books.find({$text: {$search: 'de', $language: 'french'}}).count()
0
```

Search for "de" finds only English books

Search for "de" in French finds nothing

In this example, you search for the word *de* first in English and then in French. When the search is done in English, you find a number of books. In this case you're finding books with authors who have the word *de* in their name. You won't find the French book because in French *de* is a stop word and therefore isn't indexed.

If you perform this same search in French, you won't find any results because *de* is a stop word in French. As a result, the parsed search string won't contain any words to search for once the stop words are removed.

As you can see, language can have a big effect on the results of your text search as well as the text search indexes created for a document. That's why it's important to specify the language you'll be using in your index, document, and search string. If you're only worried about English, then your task is much simpler. But if not, read on to see which languages MongoDB supports. Hopefully you'll find the languages you need.

9.6.4 Available languages

MongoDB supports quite a few languages, and you can expect the list to grow over time. The following lists the languages supported by MongoDB as of release 2.6 (the same languages are also supported by MongoDB v3.0). The list also shows the two-letter abbreviation you can use instead of the full word:

- da—danish
- nl—dutch
- en—english

- fi—finnish
- fr—french
- de—german
- hu—hungarian
- it—italian
- no—norwegian
- pt—portuguese
- ro—romanian
- ru—russian
- es—spanish
- sv—swedish
- tr—turkish

In addition to this list, you can specify `none`. When you do, MongoDB skips any processing for stemming and stop words. For example, a document with the language of `none` will have an index created for each unique word in the document. Only the exact words will be indexed, without any stemming, and the index won't exclude any stop words. This approach can be useful if your documents contain words that MongoDB is having a difficult time processing. The downside is that you won't be able to take advantage of stemming finding "similar" words and your results will contain only exact word matches.

9.7 Summary

As you can see, MongoDB can provide a great deal of capabilities in using a basic query text search for your database. The aggregation framework provides even more complex search capabilities if needed. But MongoDB text search has its limits and isn't intended to completely replace dedicated text search engines such as Elasticsearch or Solr. If you can get by with MongoDB text search, though, you'll save yourself the effort and complexity of maintaining a duplicate copy of the data within a dedicated search engine.

Now that you know the full capabilities of MongoDB searches, updates, and indexing, let's move on to a topic that's new to MongoDB v3.0 and has to do with how MongoDB stores, updates, and reads data: the WiredTiger storage engine!