- MongoDB, a powerful document-oriented database that is ideal for storing mail and other forms of social web data
- The Enron corpus, a public data set consisting of the contents of employee mailboxes from around the time of the Enron scandal
- Using MongoDB to query the Enron corpus in arbitrary ways
- Tools for accessing and exporting your own mailbox data for analysis

6.2. Obtaining and Processing a Mail Corpus

This section illustrates how to obtain a mail corpus, convert it into a standardized mbox, and then import the mbox into MongoDB, which will serve as a general-purpose API for storing and querying the data. We'll start out by analyzing a small fictitious mailbox and then proceed to processing the Enron corpus.

6.2.1. A Primer on Unix Mailboxes

An mbox is really just a large text file of concatenated mail messages that are easily accessible by text-based tools. Mail tools and protocols have long since evolved beyond mboxes, but it's usually the case that you can use this format as a lowest common denominator to easily process the data and feel confident that if you share or distribute the data it'll be just as easy for someone else to process it. In fact, most mail clients provide an "export" or "save as" option to export data to this format (even though the verbiage may vary), as illustrated in Figure 6-2 in the section Section 6.5 on page 268.

In terms of specification, the beginning of each message in an mbox is signaled by a special *From* line formatted to the pattern "`From` *`user@example.com asctime`*", where `asctime` is a standardized fixed-width representation of a timestamp in the form `Fri Dec 25 00:06:42 2009`. The boundary between messages is determined by a *From_* line preceded (except for the first occurrence) by exactly two new line characters. (Visually, as shown below, this appears as though there is a single blank line that precedes the *From_* line.) A small slice from a fictitious mbox containing two messages follows:

```
From santa@northpole.example.org Fri Dec 25 00:06:42 2009
Message-ID: <16159836.1075855377439@mail.northpole.example.org>
References: <88364590.8837464573838@mail.northpole.example.org>
In-Reply-To: <194756537.0293874783209@mail.northpole.example.org>
Date: Fri, 25 Dec 2001 00:06:42 -0000 (GMT)
From: St. Nick <santa@northpole.example.org>
To: rudolph@northpole.example.org
Subject: RE: FWD: Tonight
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
```

```
Sounds good. See you at the usual location.

Thanks,
-S

 -----Original Message-----
From:   Rudolph
Sent:   Friday, December 25, 2009 12:04 AM
To: Claus, Santa
Subject:    FWD:  Tonight

Santa -

Running a bit late. Will come grab you shortly. Standby.

Rudy

Begin forwarded message:

> Last batch of toys was just loaded onto sleigh.
>
> Please proceed per the norm.
>
> Regards,
> Buddy
>
> --
> Buddy the Elf
> Chief Elf
> Workshop Operations
> North Pole
> buddy.the.elf@northpole.example.org

From buddy.the.elf@northpole.example.org Fri Dec 25 00:03:34 2009
Message-ID: <88364590.8837464573838@mail.northpole.example.org>
Date: Fri, 25 Dec 2001 00:03:34 -0000 (GMT)
From: Buddy <buddy.the.elf@northpole.example.org>
To: workshop@northpole.example.org
Subject: Tonight
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit

Last batch of toys was just loaded onto sleigh.

Please proceed per the norm.

Regards,
Buddy

--
Buddy the Elf
```

```
Chief Elf
Workshop Operations
North Pole
buddy.the.elf@northpole.example.org
```

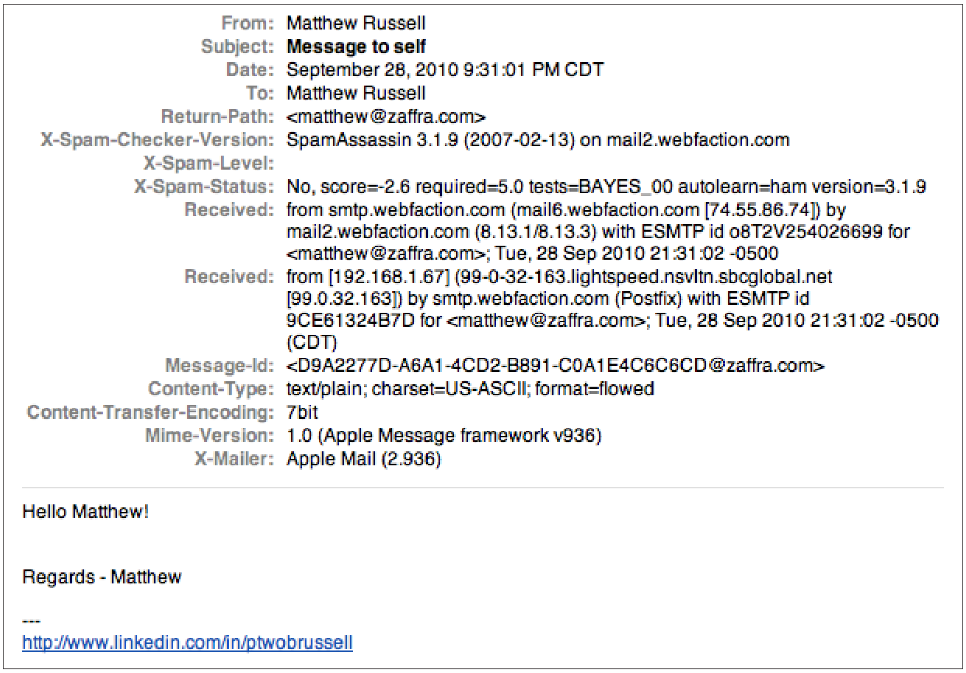
In the preceding sample mailbox we see two messages, although there is evidence of at least one other message that was replied to that might exist elsewhere in the mbox. Chronologically, the first message was authored by a fellow named Buddy and was sent out to *workshop@northpole.example.org* to announce that the toys had just been loaded. The other message in the mbox is a reply from Santa to Rudolph. Not shown in the sample mbox is an intermediate message in which Rudolph forwarded Buddy's message to Santa with the note saying that he was running late. Although we could infer these things by reading the text of the messages themselves as humans with contextualized knowledge, the Message-ID, References, and In-Reply-To headers also provide important clues that can be analyzed.

These headers are pretty intuitive and provide the basis for algorithms that display threaded discussions and things of that nature. We'll look at a well-known algorithm that uses these fields to thread messages a bit later, but the gist is that each message has a unique message ID, contains a reference to the exact message that is being replied to in the case of it being a reply, and can reference multiple other messages in the reply chain that are part of the larger discussion thread at hand.

Because we'll be employing some Python modules to do much of the tedious work for us, we won't need to digress into discussions concerning the nuances of email messages, such as multipart content, MIME types, and 7-bit content transfer encoding.

These headers are vitally important. Even with this simple example, you can already see how things can get quite messy when you're parsing the actual body of a message: Rudolph's client quoted forwarded content with > characters, while the mail client Santa used to reply apparently didn't quote anything, but instead included a human-readable message header.

Most mail clients have an option to display extended mail headers beyond the ones you normally see, if you're interested in a technique that's a little more accessible than digging into raw storage when you want to view this kind of information; Figure 6-1 shows sample headers as displayed by Apple Mail.



From: Matthew Russell
Subject: **Message to self**
Date: September 28, 2010 9:31:01 PM CDT
To: Matthew Russell
Return-Path: <matthew@zaffra.com>
X-Spam-Checker-Version: SpamAssassin 3.1.9 (2007-02-13) on mail2.webfaction.com
X-Spam-Level:
X-Spam-Status: No, score=-2.6 required=5.0 tests=BAYES_00 autolearn=ham version=3.1.9
Received: from smtp.webfaction.com (mail6.webfaction.com [74.55.86.74]) by mail2.webfaction.com (8.13.1/8.13.3) with ESMTP id o8T2V254026699 for <matthew@zaffra.com>; Tue, 28 Sep 2010 21:31:02 -0500
Received: from [192.168.1.67] (99-0-32-163.lightspeed.nsvltn.sbcglobal.net [99.0.32.163]) by smtp.webfaction.com (Postfix) with ESMTP id 9CE61324B7D for <matthew@zaffra.com>; Tue, 28 Sep 2010 21:31:02 -0500 (CDT)
Message-Id: <D9A2277D-A6A1-4CD2-B891-C0A1E4C6C6CD@zaffra.com>
Content-Type: text/plain; charset=US-ASCII; format=flowed
Content-Transfer-Encoding: 7bit
Mime-Version: 1.0 (Apple Message framework v936)
X-Mailer: Apple Mail (2.936)

Hello Matthew!

Regards - Matthew

---
http://www.linkedin.com/in/ptwobrussell

Figure 6-1. Most mail clients allow you to view the extended headers through an options menu

Luckily for us, there's a lot you can do without having to essentially reimplement a mail client. Besides, if all you wanted to do was browse the mailbox, you'd simply import it into a mail client and browse away, right?

It's worth taking a moment to explore whether your mail client has an option to import/export data in the mbox format so that you can use the tools in this chapter to manipulate it.

To get the ball rolling on some data processing, Example 6-1 illustrates a routine that makes numerous simplifying assumptions about an mbox to introduce the `mailbox` and `email` packages that are part of Python's standard library.

Example 6-1. Converting a toy mailbox to JSON

```
import mailbox
import email
import json

MBOX = 'resources/ch06-mailboxes/data/northpole.mbox'
```

```
# A routine that makes a ton of simplifying assumptions
# about converting an mbox message into a Python object
# given the nature of the northpole.mbox file in order
# to demonstrate the basic parsing of an mbox with mail
# utilities

def objectify_message(msg):

    # Map in fields from the message
    o_msg = dict([ (k, v) for (k,v) in msg.items() ])

    # Assume one part to the message and get its content
    # and its content type

    part = [p for p in msg.walk()][0]
    o_msg['contentType'] = part.get_content_type()
    o_msg['content'] = part.get_payload()

    return o_msg

# Create an mbox that can be iterated over and transform each of its
# messages to a convenient JSON representation

mbox = mailbox.UnixMailbox(open(MBOX, 'rb'), email.message_from_file)

messages = []

while 1:
    msg = mbox.next()

    if msg is None: break

    messages.append(objectify_message(msg))

print json.dumps(messages, indent=1)
```

Although this little script for processing an mbox file seems pretty clean and produces reasonable results, trying to parse arbitrary mail data or determine the exact flow of a conversation from mailbox data for the general case can be a tricky enterprise. Many factors contribute to this, such as the ambiguity involved and the variation that can occur in how humans embed replies and comments into reply chains, how different mail clients handle messages and replies, etc.

Table 6-1 illustrates the message flow and explicitly includes the third message that was referenced but not present in the *northpole.mbox* to highlight this point. Truncated sample output from the script follows:

```
[
 {
  "From": "St. Nick <santa@northpole.example.org>",
  "Content-Transfer-Encoding": "7bit",
  "content": "Sounds good. See you at the usual location.\n\nThanks,...",
  "To": "rudolph@northpole.example.org",
  "References": "<88364590.8837464573838@mail.northpole.example.org>",
  "Mime-Version": "1.0",
  "In-Reply-To": "<194756537.0293874783209@mail.northpole.example.org>",
  "Date": "Fri, 25 Dec 2001 00:06:42 -0000 (GMT)",
  "contentType": "text/plain",
  "Message-ID": "<16159836.1075855377439@mail.northpole.example.org>",
  "Content-Type": "text/plain; charset=us-ascii",
  "Subject": "RE: FWD: Tonight"
 },
 {
  "From": "Buddy <buddy.the.elf@northpole.example.org>",
  "Subject": "Tonight",
  "Content-Transfer-Encoding": "7bit",
  "content": "Last batch of toys was just loaded onto sleigh. \n\nPlease...",
  "To": "workshop@northpole.example.org",
  "Date": "Fri, 25 Dec 2001 00:03:34 -0000 (GMT)",
  "contentType": "text/plain",
  "Message-ID": "<88364590.8837464573838@mail.northpole.example.org>",
  "Content-Type": "text/plain; charset=us-ascii",
  "Mime-Version": "1.0"
 }
]
```

Table 6-1. Message flow from northpole.mbox

Date	Message activity
Fri, 25 Dec 2001 00:03:34 -0000 (GMT)	Buddy sends a message to the workshop
Friday, December 25, 2009 12:04 AM	Rudolph forwards Buddy's message to Santa with an additional note
Fri, 25 Dec 2001 00:06:42 -0000 (GMT)	Santa replies to Rudolph

With a basic appreciation for mailboxes in place, let's now shift our attention to converting the Enron corpus to an mbox so that we can leverage Python's standard library as much as possible.

6.2.2. Getting the Enron Data

A downloadable form of the full Enron data set in a raw form is available in multiple formats requiring various amounts of processing. We'll opt to start with the original raw form of the data set, which is essentially a set of folders that organizes a collection of mailboxes by person and folder. Data standardization and cleansing is a routine problem, and this section should give you some perspective and some appreciation for it.

If you are taking advantage of the virtual machine experience for this book, the IPython Notebook for this chapter provides a script that downloads the data to the proper working location for you to seamlessly follow along with these examples. The full Enron corpus is approximately 450 MB in the compressed form in which you would download it to follow along with these exercises. It may take upward of 10 minutes to download and decompress if you have a reasonable Internet connection speed and a relatively new computer.

Unfortunately, if you are using the virtual machine, the time that it takes for Vagrant to synchronize the thousands of files that are unarchived back to the host machine can be upward of two hours. If time is a significant factor and you can't let this script run at an opportune time, you could opt to skip the download and initial processing steps since the refined version of the data, as produced from Example 6-3, is checked in with the source code and available at *ipynb/resources/ch06-mailboxes/data/enron.mbox*.json.bz2. See the notes in the IPython Notebook for this chapter for more details.

The download and decompression of the file is relatively fast compared to the time that it takes for Vagrant to synchronize the high number of files that decompress with the host machine, and at the time of this writing, there isn't a known workaround that will speed this up for all platforms. It may take longer than a hour for Vagrant to synchronize the thousands of files that decompress.

The output from the following terminal session illustrates the basic structure of the corpus once you've downloaded and unarchived it. It's worthwhile to explore the data in a terminal session for a few minutes once you've downloaded it to familiarize yourself with what's there and learn how to navigate through it.

If you are working on a Windows system or are not comfortable working in a terminal, you can poke around in the*ipynb/resources/ch06-mailboxes/data* folder, which will be synchronized onto your host machine if you are taking advantage of the virtual machine experience for this book.

```
$ cd enron_mail_20110402/maildir # Go into the mail directory

maildir $ ls # Show folders/files in the current directory

allen-p         crandell-s      gay-r           horton-s
lokey-t         nemec-g         rogers-b        slinger-r
tycholiz-b      arnold-j        cuilla-m        geaccone-t
hyatt-k         love-p          panus-s         ruscitti-k
smith-m         ward-k          arora-h         dasovich-j
germany-c       hyvl-d          lucci-p         parks-j
sager-e         solberg-g       watson-k        badeer-r
corman-s        gang-l          holst-k         lokay-m
```

```
                ...directory listing truncated...

neal-s          rodrique-r      skilling-j      townsend-j

$ cd allen-p/ # Go into the allen-p folder

allen-p $ ls # Show files in the current directory

_sent_mail         contacts           discussion_threads notes_inbox
sent_items         all_documents      deleted_items      inbox
sent               straw

allen-p $ cd inbox/ # Go into the inbox for allen-p

inbox $ ls # Show the files in the inbox for allen-p

1.  11. 13. 15. 17. 19. 20. 22. 24. 26. 28. 3.  31. 33. 35. 37. 39. 40.
42. 44. 5.  62. 64. 66. 68. 7.  71. 73. 75. 79. 83. 85. 87. 10. 12. 14.
16. 18. 2.  21. 23. 25. 27. 29. 30. 32. 34. 36. 38. 4.  41. 43. 45. 6.
63. 65. 67. 69. 70. 72. 74. 78. 8.  84. 86. 9.

inbox $ head -20 1. # Show the first 20 lines of the file named "1."

Message-ID: <16159836.1075855377439.JavaMail.evans@thyme>
Date: Fri, 7 Dec 2001 10:06:42 -0800 (PST)
From: heather.dunton@enron.com
To: k..allen@enron.com
Subject: RE: West Position
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
X-From: Dunton, Heather </O=ENRON/OU=NA/CN=RECIPIENTS/CN=HDUNTON>
X-To: Allen, Phillip K. </O=ENRON/OU=NA/CN=RECIPIENTS/CN=Pallen>
X-cc:
X-bcc:
X-Folder: \Phillip_Allen_Jan2002_1\Allen, Phillip K.\Inbox
X-Origin: Allen-P
X-FileName: pallen (Non-Privileged).pst


Please let me know if you still need Curve Shift.

Thanks,
```

The final command in the terminal session shows that mail messages are organized into files and contain metadata in the form of headers that can be processed along with the content of the data itself. The data is in a fairly consistent format, but not necessarily a well-known format with great tools for processing it. So, let's do some preprocessing on the data and convert a portion of it to the well-known Unix mbox format in order to illustrate the general process of standardizing a mail corpus to a format that is widely known and well tooled.

6.2.3. Converting a Mail Corpus to a Unix Mailbox

Example 6-2 illustrates an approach that searches the directory structure of the Enron corpus for folders named "inbox" and adds messages contained in them to a single output file that's written out as *enron.mbox*. To run this script, you will need to download the Enron corpus and unarchive it to the path specified by `MAILDIR` in the script.

The script takes advantage of a package called `dateutil` to handle the parsing of dates into a standard format. We didn't do this earlier, and it's slightly trickier than it may sound given the room for variation in the general case. You can install this package with **`pip install python_dateutil`**. (In this particular instance, the package name that `pip` tries to install is slightly different than what you import in your code.) Otherwise, the script is just using some tools from Python's standard library to munge the data into an mbox. Although not analytically interesting, the script provides reminders of how to use regular expressions, uses the `email` package that we'll continue to see, and illustrates some other concepts that may be useful for general data processing. Be sure that you understand how the script works to broaden your overall working knowledge and data mining toolchain.

This script may take 10–15 minutes to run on the entire Enron corpus, depending on your hardware. IPython Notebook will indicate that it is still processing data by displaying a "Kernel Busy" message in the upper-right corner of the user interface.

Example 6-2. Converting the Enron corpus to a standardized mbox format

```
import re
import email
from time import asctime
import os
import sys
from dateutil.parser import parse # pip install python_dateutil

# XXX: Download the Enron corpus to resources/ch06-mailboxes/data
# and unarchive it there.

MAILDIR = 'resources/ch06-mailboxes/data/enron_mail_20110402/' + \
          'enron_data/maildir'

# Where to write the converted mbox
MBOX = 'resources/ch06-mailboxes/data/enron.mbox'

# Create a file handle that we'll be writing into...
mbox = open(MBOX, 'w')

# Walk the directories and process any folder named 'inbox'

for (root, dirs, file_names) in os.walk(MAILDIR):
    if root.split(os.sep)[-1].lower() != 'inbox':
        continue

    # Process each message in 'inbox'

    for file_name in file_names:
        file_path = os.path.join(root, file_name)
        message_text = open(file_path).read()

        # Compute fields for the From_ line in a traditional mbox message

        _from = re.search(r"From: ([^\r]+)", message_text).groups()[0]
        _date = re.search(r"Date: ([^\r]+)", message_text).groups()[0]

        # Convert _date to the asctime representation for the From_ line

        _date = asctime(parse(_date).timetuple())

        msg = email.message_from_string(message_text)
        msg.set_unixfrom('From %s %s' % (_from, _date))

        mbox.write(msg.as_string(unixfrom=True) + "\n\n")

mbox.close()
```

If you peek at the mbox file that we've just created, you'll see that it looks quite similar to the mail format we saw earlier, except that it now conforms to well-known specifications and is a single file.

Keep in mind that you could just as easily create separate mbox files for each individual person or a particular group of people if you preferred to analyze a more focused subset of the Enron corpus.

6.2.4. Converting Unix Mailboxes to JSON

Having an mbox file is especially convenient because of the variety of tools available to process it across computing platforms and programming languages. In this section we'll look at eliminating many of the simplifying assumptions from Example 6-1, to the point that we can robustly process the Enron mailbox and take into account several of the common issues that you'll likely encounter with mailbox data from the wild. Python's tooling for mboxes is included in its standard library, and the script in Example 6-3 introduces a means of converting mbox data to a line-delimited JSON format that can be imported into a document-oriented database such as MongoDB. We'll talk more about MongoDB and why it's such a great fit for storing content such as mail data in a moment, but for now, it's sufficient to know that it stores data in what's conceptually a

JSON-like format and provides some powerful capabilities for indexing and manipulating the data.

One additional accommodation that we make for MongoDB is that we normalize the date of each message to a standard *epoch* format that's the number of milliseconds since January 1, 1970, and pass it in with a special hint so that MongoDB can interpret each date field in a standardized way. Although we could have done this after we loaded the data into MongoDB, this chore falls into the "data cleansing" category and enables us to run some queries that use the *Date* field of each mail message in a consistent way immediately after the data is loaded.

Finally, in order to actually get the data to import into MongoDB, we need to write out a file in which each line contains a single JSON object, per MongoDB's documentation. Once again, although not interesting from the standpoint of analysis, this script illustrates some additional realities in data cleansing and processing—namely, that mail data may not be in a particular encoding like UTF-8 and may contain HTML formatting that needs to be stripped out.

Example 6-3 includes the `decode('utf-8', 'ignore')` function in several places. When you're working with text-based data such as emails or web pages, it's not at all uncommon to run into the infamous `UnicodeDecodeError` because of unexpected character encodings, and it's not always immediately obvious what's going on or how to fix the problem. You can run the `decode` function on any string value and pass it a second argument that specifies what to do in the event of a `UnicodeDecodeError`. The default value is `'strict'`, which results in the exception being raised, but you can use `'ignore'` or `'replace'` instead, depending on your needs.

Example 6-3. Converting an mbox to a JSON structure suitable for import into MongoDB

```
import sys
import mailbox
import email
import quopri
import json
import time
from BeautifulSoup import BeautifulSoup
from dateutil.parser import parse

MBOX = 'resources/ch06-mailboxes/data/enron.mbox'
OUT_FILE = 'resources/ch06-mailboxes/data/enron.mbox.json'

def cleanContent(msg):

    # Decode message from "quoted printable" format
    msg = quopri.decodestring(msg)

    # Strip out HTML tags, if any are present.
    # Bail on unknown encodings if errors happen in BeautifulSoup.
    try:
        soup = BeautifulSoup(msg)
    except:
        return ''
    return ''.join(soup.findAll(text=True))

# There's a lot of data to process, and the Pythonic way to do it is with a
# generator. See http://wiki.python.org/moin/Generators.
# Using a generator requires a trivial encoder to be passed to json for object
# serialization.

class Encoder(json.JSONEncoder):
    def default(self, o): return  list(o)

# The generator itself...
def gen_json_msgs(mb):
    while 1:
        msg = mb.next()
        if msg is None:
            break
        yield jsonifyMessage(msg)

def jsonifyMessage(msg):
    json_msg = {'parts': []}
    for (k, v) in msg.items():
        json_msg[k] = v.decode('utf-8', 'ignore')

    # The To, Cc, and Bcc fields, if present, could have multiple items.
    # Note that not all of these fields are necessarily defined.

    for k in ['To', 'Cc', 'Bcc']:
        if not json_msg.get(k):
            continue
        json_msg[k] = json_msg[k].replace('\n', '').replace('\t', '').replace('\r', '')\
                                 .replace(' ', '').decode('utf-8', 'ignore').split(',')

    for part in msg.walk():
        json_part = {}
        if part.get_content_maintype() == 'multipart':
            continue

        json_part['contentType'] = part.get_content_type()
        content = part.get_payload(decode=False).decode('utf-8', 'ignore')
        json_part['content'] = cleanContent(content)

        json_msg['parts'].append(json_part)

    # Finally, convert date from asctime to milliseconds since epoch using the
    # $date descriptor so it imports "natively" as an ISODate object in MongoDB
    then = parse(json_msg['Date'])
    millis = int(time.mktime(then.timetuple())*1000 + then.microsecond/1000)
    json_msg['Date'] = {'$date' : millis}

    return json_msg

mbox = mailbox.UnixMailbox(open(MBOX, 'rb'), email.message_from_file)

# Write each message out as a JSON object on a separate line
# for easy import into MongoDB via mongoimport

f = open(OUT_FILE, 'w')
for msg in gen_json_msgs(mbox):
    if msg != None:
        f.write(json.dumps(msg, cls=Encoder) + '\n')
f.close()
```

There's always more data cleansing that we could do, but we've addressed some of the most common issues, including a primitive mechanism for decoding quoted-printable text and stripping out HTML tags. (The `quopri` package is used to handle the quoted-printable format, an encoding used to transfer 8-bit content over a 7-bit channel.[2]) Following is one line of pretty-printed sample output from running Example 6-3 on the Enron mbox file, to demonstrate the basic form of the output:

```
    {
        "Content-Transfer-Encoding": "7bit",
        "Content-Type": "text/plain; charset=us-ascii",
        "Date": {
            "$date": 988145040000
        },
        "From": "craig_estes@enron.com",
        "Message-ID": "<24537021.1075840152262.JavaMail.evans@thyme>",
        "Mime-Version": "1.0",
        "Subject": "Parent Child Mountain Adventure, July 21-25, 2001",
        "X-FileName": "jskillin.pst",
        "X-Folder": "\\jskillin\\Inbox",
        "X-From": "Craig_Estes",
        "X-Origin": "SKILLING-J",
        "X-To": "",
        "X-bcc": "",
        "X-cc": "",
        "parts": [
            {
                "content": "Please respond to Keith_Williams...",
                "contentType": "text/plain"
            }
```

2. See Wikipedia for an overview, or RFC 2045 if you are interested in the nuts and bolts of how this works.

```
        ]
    }
```

This short script does a pretty decent job of removing some of the noise, parsing out the most pertinent information from an email, and constructing a data file that we can now trivially import into MongoDB. This is where the real fun begins. With your newfound ability to cleanse and process mail data into an accessible format, the urge to start analyzing it is only natural. In the next section, we'll import the data into MongoDB and begin the data analysis.

> If you opted not to download the original Enron data and follow along with the preprocessing steps, you can still produce the output from Example 6-3 by following along with the notes in the IPython Notebook for this chapter and proceed from here per the standard discussion that continues.

6.2.5. Importing a JSONified Mail Corpus into MongoDB

Using the right tool for the job can significantly streamline the effort involved in analyzing data, and although Python is a language that would make it fairly simple to process JSON data, it still wouldn't be nearly as easy as storing the JSON data in a document-oriented database like MongoDB.

For all practical purposes, think of MongoDB as a database that makes storing and manipulating JSON just about as easy as it should be. You can organize it into collections, iterate over it and query it in efficient ways, full-text index it, and much more. In the current context of analyzing the Enron corpus, MongoDB provides a natural API into the data since it allows us to create indexes and query on arbitrary fields of the JSON documents, even performing a full-text search if desired.

For our exercises, you'll just be running an instance of MongoDB on your local machine, but you can also scale MongoDB across a cluster of machines as your data grows. It comes with great administration utilities, and it's backed by a professional services company should you need pro support. A full-blown discussion about MongoDB is outside the scope of this book, but it should be straightforward enough to follow along with this section even if you've never heard of MongoDB until reading this chapter. Its online documentation and tutorials are superb, so take a moment to bookmark them since they make such a handy reference.

Regardless of your operating system, should you choose to install MongoDB instead of using the virtual machine, you should be able to follow the instructions online easily enough; nice packaging for all major platforms is available. Just make sure that you are using version 2.4 or higher since some of the exercises in this chapter rely on full-text indexing, which is a new beta feature introduced in version 2.4. For reference, the MongoDB that is preinstalled with the virtual machine is installed and managed as a service with no particular customization aside from setting a parameter in its configuration file (located at */etc/mongodb.conf*) to enable full-text search indexing.

Verify that the Enron data is loaded, full-text indexed, and ready for analysis by executing Examples 6-4, 6-5, and 6-6. These examples take advantage of a lightweight wrapper around the `subprocess` package called Envoy, which allows you to easily execute terminal commands from a Python program and get the standard output and standard error. Per the standard protocol, you can install envoy with **`pip install envoy`** from a terminal.

Example 6-4. Getting the options for the mongoimport command from IPython Notebook

```
import envoy # pip install envoy

r = envoy.run('mongoimport')
print r.std_out
print r.std_err
```

Example 6-5. Using mongoimport to load data into MongoDB from IPython Notebook

```
import os
import sys
import envoy

data_file = os.path.join(os.getcwd(), 'resources/ch06-mailboxes/data/enron.mbox.json')

# Run a command just as you would in a terminal on the virtual machine to
# import the data file into MongoDB.
r = envoy.run('mongoimport --db enron --collection mbox ' + \
              '--file %s' % data_file)

# Print its standard output
print r.std_out
print sys.stderr.write(r.std_err)
```

Example 6-6. Simulating a MongoDB shell that you can run from within IPython Notebook

```
# We can even simulate a MongoDB shell using envoy to execute commands.
# For example, let's get some stats out of MongoDB just as though we were working
# in a shell by passing it the command and wrapping it in a printjson function to
# display it for us.

def mongo(db, cmd):
    r = envoy.run("mongo %s --eval 'printjson(%s)'" % (db, cmd,))
    print r.std_out
    if r.std_err: print r.std_err

mongo('enron', 'db.mbox.stats()')
```

Sample output from Example 6-6 follows and illustrates that it's exactly what you'd see if you were writing commands in the MongoDB shell. Neat!

```
MongoDB shell version: 2.4.3
connecting to: enron
{
  "ns" : "enron.mbox",
  "count" : 41299,
  "size" : 157744000,
  "avgObjSize" : 3819.5597956366983,
  "storageSize" : 185896960,
  "numExtents" : 10,
  "nindexes" : 1,
  "lastExtentSize" : 56438784,
  "paddingFactor" : 1,
  "systemFlags" : 1,
  "userFlags" : 0,
  "totalIndexSize" : 1349040,
  "indexSizes" : {
    "_id_" : 1349040
  },
  "ok" : 1
}
```

Loading the JSON data through a terminal session on the virtual machine can be accomplished through `mongoimport` in exactly the same fashion as illustrated in Example 6-5 with the following command:

```
mongoimport --db enron --collection mbox --file
/home/vagrant/share/ipynb/resources/ch06-mailboxes
/data/enron.mbox.json
```

Once MongoDB is installed, the final administrative task you'll need to perform is installing the Python client package `pymongo` via the usual **`pip install pymongo`** command, since we'll soon be using a Python client to connect to MongoDB and access the Enron data.

Be advised that MongoDB supports only databases of up to 2 GB in size for 32-bit systems. Although this limitation is not likely to be an issue for the Enron data set that we're working with in this chapter, you may want to take note of it in case any of the machines you commonly work on are 32-bit systems.

6.2.5.1. The MongoDB shell

Although we are programmatically using Python for our exercises in this chapter, MongoDB has a shell that can be quite convenient if you are comfortable working in a terminal, and this brief section introduces you to it. If you are taking advantage of the virtual machine experience for this book, you will need to log into the virtual machine over a secure shell session in order to follow along. Typing **`vagrant ssh`** from inside the top-level checkout folder containing your *Vagrantfile* automatically logs you into the virtual machine.

If you run Mac OS X or Linux, an SSH client will already exist on your system and `vagrant ssh` will just work. If you are a Windows user and followed the instructions in Appendix A recommending the installation of Git for Windows, which provides an SSH client, `vagrant ssh` will also work so long as you explicitly opt to install the SSH client as part of the installation process. If you are a Windows user and prefer to use PuTTY, typing **`vagrant ssh`** provides some instructions on how to configure it:

```
$ vagrant ssh

Last login: Sat Jun  1 04:18:57 2013 from 10.0.2.2

vagrant@precise64:~$ mongo
MongoDB shell version: 2.4.3
connecting to: test

> show dbs
enron 0.953125GB
local 0.078125GB

> use enron
switched to db enron

> db.mbox.stats()
{
  "ns" : "enron.mbox",
  "count" : 41300,
  "size" : 157756112,
  "avgObjSize" : 3819.7605811138014,
  "storageSize" : 174727168,
  "numExtents" : 11,
  "nindexes" : 2,
  "lastExtentSize" : 50798592,
  "paddingFactor" : 1,
  "systemFlags" : 0,
  "userFlags" : 1,
  "totalIndexSize" : 221471488,
  "indexSizes" : {
    "_id_" : 1349040,
    "TextIndex" : 220122448
  },
  "ok" : 1
}

> db.mbox.findOne()
{
  "_id" : ObjectId("51968affaada66efc5694cb7"),
  "X-cc" : "",
  "From" : "heather.dunton@enron.com",
  "X-Folder" : "\\Phillip_Allen_Jan2002_1\\Allen, Phillip K.\\Inbox",
  "Content-Transfer-Encoding" : "7bit",
  "X-bcc" : "",
  "X-Origin" : "Allen-P",
  "To" : [
    "k..allen@enron.com"
  ],
  "parts" : [
    {
      "content" : " \nPlease let me know if you still need...",
          "contentType" : "text/plain"
    }
  ],
  "X-FileName" : "pallen (Non-Privileged).pst",
  "Mime-Version" : "1.0",
  "X-From" : "Dunton, Heather </O=ENRON/OU=NA/CN=RECIPIENTS/CN=HDUNTON>",
  "Date" : ISODate("2001-12-07T16:06:42Z"),
  "X-To" : "Allen, Phillip K. </O=ENRON/OU=NA/CN=RECIPIENTS/CN=Pallen>",
  "Message-ID" : "<16159836.1075855377439.JavaMail.evans@thyme>",
  "Content-Type" : "text/plain; charset=us-ascii",
  "Subject" : "RE: West Position"
}
```

The commands in this shell session showed the available databases, set the working database to `enron`, displayed the database statistics for `enron`, and fetched an arbitrary document for display. We won't spend more time in the MongoDB shell in this chapter, but you'll likely find it useful as you work with data, so it seemed appropriate to briefly introduce you to it. See "The Mongo Shell" in MongoDB's online documentation for details about the capabilities of the MongoDB shell.

6.2.6. Programmatically Accessing MongoDB with Python

With MongoDB successfully loaded with the Enron corpus (or any other data, for that matter), you'll want to access and manipulate it with a programming language. MongoDB is sure to please with a broad selection of libraries for many programming languages, including PyMongo, the recommended way to work with MongoDB from Python. A **`pip install pymongo`** should get PyMongo ready to use; Example 6-7 contains a simple script to show how it works. Queries are serviced by MongoDB's versatile `find` function, which you'll want to get acquainted with since it's the basis of most queries you'll perform with MongoDB.

Example 6-7. Using PyMongo to access MongoDB from Python

```
import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo
```

```
# Connects to the MongoDB server running on
# localhost:27017 by default

client = pymongo.MongoClient()

# Get a reference to the enron database

db = client.enron

# Reference the mbox collection in the Enron database

mbox = db.mbox

# The number of messages in the collection

print "Number of messages in mbox:"
print mbox.count()
print

# Pick a message to look at...

msg = mbox.find_one()

# Display the message as pretty-printed JSON. The use of
# the custom serializer supplied by PyMongo is necessary in order
# to handle the date field that is provided as a datetime.datetime
# tuple.

print "A message:"
print json.dumps(msg, indent=1, default=json_util.default)
```

Abbreviated sample output follows and demonstrates that using PyMongo is just like using the MongoDB shell, with the exception of a couple of special considerations with relation to object serialization:

```
Number of messages in mbox:
41299

A message:
{
 "X-cc": "",
 "From": "craig_estes@enron.com",
 "Content-Transfer-Encoding": "7bit",
 "X-bcc": "",
 "parts": [
  {
   "content": "Please respond to Keith_Williams\"A YPO International...",
   "contentType": "text/plain"
  }
 ],
 "X-Folder": "\\jskillin\\Inbox",
 "X-Origin": "SKILLING-J",
 "X-FileName": "jskillin.pst",
 "Mime-Version": "1.0",
 "Message-ID": "<24537021.1075840152262.JavaMail.evans@thyme>",
 "X-From": "Craig_Estes",
 "Date": {
  "$date": 988145040000
 },
 "X-To": "",
 "_id": {
  "$oid": "51a983dae391e8ff964bc4c4"
 },
 "Content-Type": "text/plain; charset=us-ascii",
 "Subject": "Parent Child Mountain Adventure, July 21-25, 2001"
}
```

It's been a bit of a journey, but by now you should have a good understanding of how to obtain some mail data, process it into a normalized Unix mailbox format, load the normalized data into MongoDB, and query it. The steps involved in analyzing any real-world data set will be largely similar to the steps that we've followed here (with their own unique twists and turns along the way), so if you've followed along carefully, you have some powerful new tools in your data science toolkit.

Map-Reduce in 30 Seconds

Map-reduce is a computing paradigm that consists of two primary functions: `map` and `reduce`. Mapping functions take a collection of documents and map out a new key/value pair for each document, while reduction functions take a collection of documents and reduce them in some way. For example, computing the arithmetic sum of squares, $f(x) = x_1^2 + x_2^2 + ... + x_n^2$, could be expressed as a mapping function that squares each value, producing a one-to-one correspondence for each input value, while the reducer simply sums the output of the mappers and reduces it to a single value. This programming pattern lends itself well to trivially parallelizable problems but is certainly not a good (performant) fit for every problem.

MongoDB and other document-oriented databases such as CouchDB and Riak support map-reduce both on a single machine and in distributed computing environments. If you are interested in working with "big data" or running highly customized queries on MongoDB, you'll want to learn more about map-reduce. At the present time, it is an essential skill in the big data computing paradigm.

6.3. Analyzing the Enron Corpus

Having invested a significant amount of energy and attention in the problem of getting the Enron data into a convenient format that we can query, let's now embark upon a quest to begin understanding the data. As you know from previous chapters, counting things is usually one of the first exploratory tasks you'll want to consider when faced with a new data set of any kind, because it can tell you so much with so little effort. This section investigates a couple of ways you can use MongoDB's versatile `find` operation to query a mailbox for various combinations of fields and criteria with minimal effort required as an extension of the working discussion.

Overview of the Enron Scandal

Although not entirely necessary, you will likely learn more in this chapter if you are notionally familiar with the Enron scandal, which is the subject of the mail data that we'll be analyzing. Following are a few key facts about Enron that will be helpful in understanding the context as we analyze the data for this chapter:

- Enron was a Texas-based energy company that grew to a multibillion-dollar company between its founding in 1985 and the scandal revealed in October 2001.
- Kenneth Lay was the CEO of Enron and the subject of many Enron-related discussions.
- The substance of the Enron scandal involved the use of financial instruments (referred to as *raptors*) to effectively hide accounting losses.
- Arthur Andersen, once a prestigious accounting firm, was responsible for performing the financial audits. It closed shortly after the Enron scandal.
- Soon after the scandal was revealed, Enron filed bankruptcy to the tune of over $60 billion dollars; this was the largest bankruptcy in U.S. history at the time.

The Wikipedia article on the Enron scandal provides an easy-to-read introduction to the background and key events, and it takes only a few minutes to read enough of it to get the gist of what happened. If you'd like to dig deeper, the documentary film *Enron: The Smartest Guys in the Room* provides all the background you'll ever need to know about Enron.

The website *http://www.enron-mail.com* hosts a version of the Enron mail data that you may find helpful as you initially get acquainted with the Enron corpus.

6.3.1. Querying by Date/Time Range

We've taken care to import data into MongoDB so that the `Date` field of each object is interpreted correctly by the database, making queries by date/time range rather trivial. In fact, Example 6-8 is just a minor extension of the working example from the previous section, in that it sets up a connection to the database and then issues the following `find` query with some parameters to constrain the query:

```
mbox.find({"Date" :
            {
             "$lt" : end_date,
             "$gt" : start_date
            }
          }).sort("date")
```

The query is saying, "Find me all of the messages that have a `Date` that's greater than `start_date` and less than `end_date`, and when you get those results, return them in sorted order." Field names that start with the dollar sign, such as `$lt` and `$gt`, are special operators in MongoDB and in this case refer to "less than" and "greater than," respectively. You can read about all of the other MongoDB operators in the excellent online documentation.

One other thing to keep in mind about this query is that sorting data usually takes additional time unless it's already indexed to be in the particular sorted order in which you are requesting it. We arrived at the particular date range for the query in Example 6-8 by arbitrarily picking a date based upon the general time range from our previous results from `findOne`, which showed us that there was data in the mailbox circa 2001.

Example 6-8. Querying MongoDB by date/time range

```
import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo
from datetime import datetime as dt

client = pymongo.MongoClient()

db = client.enron

mbox = db.mbox

# Create a small date range here of one day

start_date = dt(2001, 4, 1) # Year, Month, Day
end_date = dt(2001, 4, 2) # Year, Month, Day

# Query the database with the highly versatile "find" command,
# just like in the MongoDB shell.

msgs = [ msg
         for msg in mbox.find({"Date" :
                                  {
                                   "$lt" : end_date,
                                   "$gt" : start_date
                                  }
                              }).sort("date")]
```

```
# Create a convenience function to make pretty-printing JSON a little
# less cumbersome

def pp(o, indent=1):
    print json.dumps(msgs, indent=indent, default=json_util.default)

print "Messages from a query by date range:"
pp(msgs)
```

The following sample output shows that there was only one message in the data set for this particular date range:

```
Messages from a query by date range:
[
 {
  "X-cc": "",
  "From": "spisano@sprintmail.com",
  "Subject": "House repair bid",
  "To": [
   "kevin.ruscitti@enron.com"
  ],
  "Content-Transfer-Encoding": "7bit",
  "X-bcc": "",
  "parts": [
   {
    "content": "\n \n - RUSCITTI BID.wps \n\n",
    "contentType": "text/plain"
   }
  ],
  "X-Folder": "\\Ruscitti, Kevin\\Ruscitti, Kevin\\Inbox",
  "X-Origin": "RUSCITTI-K",
  "X-FileName": "Ruscitti, Kevin.pst",
  "Message-ID": "<8472651.1075845282216.JavaMail.evans@thyme>",
  "X-From": "Steven Anthony Pisano  <spisano@sprintmail.com>",
  "Date": {
   "$date": 986163540000
  },
  "X-To": "KEVIN.RUSCITTI@ENRON.COM",
  "_id": {
   "$oid": "51a983dfe391e8ff964c5229"
  },
  "Content-Type": "text/plain; charset=us-ascii",
  "Mime-Version": "1.0"
 }
]
```

Since we've carefully imported the data into MongoDB prior to this query, that's basically all that you need to do in order to slice and dice the data by a date and/or time range. Although it may seem like a "freebie," this ease of querying is dependent upon your having thought about the kinds of queries you'll want to run during the munging and import process. Had you not imported the data in a way that took advantage of MongoDB's abilities to respect dates as particular kinds of specialized fields, you'd have had the chore of now doing that work before you could make this query.

The other thing worth noting here is that Python's `datetime` function—which was constructed with a year, month, and date—can be extended to include an hour, minute, second, and even microsecond, along with optional time zone information, as additional constraints in the tuple. Hours take values between 0 and 23. For example, a value of `(2013, 12, 25, 0, 23, 5)` would be December 25, 2013 at 12:23:05 AM. Although not necessarily the most convenient package to work with, `datetime` is definitely worth exploring since querying data sets by date/time range is among the most common things you might want to do on any given data analysis occasion.

6.3.2. Analyzing Patterns in Sender/Recipient Communications

Other metrics, such as how many messages a given person originally authored or how many direct communications occurred between any given group of people, are highly relevant statistics to consider as part of email analysis. However before you start analyzing who is communicating with whom, you may first want to simply enumerate all of the possible senders and receivers, optionally constraining the query by a criterion such as the domain from which the emails originated or to which they were delivered. As a starting point in this illustration, let's calculate the number of distinct email addresses that sent or received messages, as demonstrated in Example 6-9.

Example 6-9. Enumerating senders and receivers of messages

```
import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo

client = pymongo.MongoClient()
db = client.enron
mbox = db.mbox

senders = [ i for i in mbox.distinct("From") ]

receivers = [ i for i in mbox.distinct("To") ]

cc_receivers = [ i for i in mbox.distinct("Cc") ]

bcc_receivers = [ i for i in mbox.distinct("Bcc") ]

print "Num Senders:", len(senders)
print "Num Receivers:", len(receivers)
print "Num CC Receivers:", len(cc_receivers)
print "Num BCC Receivers:", len(bcc_receivers)
```

Sample output for the working data set follows:

```
Num Senders: 7665
Num Receivers: 22162
Num CC Receivers: 6561
Num BCC Receivers: 6561
```

Without any other information, these counts of senders and receivers are fairly interesting to consider. On average, each message was sent to three people, with a fairly substantial number of courtesy copies (CCs) and blind courtesy copies (BCCs) on the messages. The next step might be to winnow down the data and use basic set operations (as introduced back in Chapter 1) to determine what kind of overlap exists between various combinations of these criteria. To do that, we'll simply need to cast the lists that contain each unique value to `sets` so that we can make various kinds of set comparisons, including intersections, differences, and unions. Table 6-2 illustrates these basic operations over this small universe of senders and receivers to show you how this will work on the data:

```
Senders = {Abe, Bob}, Receivers = {Bob, Carol}
```

Table 6-2. Sample set operations

Operation	Operation name	Result	Comment
`Senders ∪ Receivers`	Union	`Abe, Bob, Carol`	All unique senders and receivers of messages
`Senders ∩ Receivers`	Intersection	`Bob`	Senders who were also receivers of messages
`Senders - Receivers`	Difference	`Abe`	Senders who did not receive messages
`Receivers - Senders`	Difference	`Carol`	Receivers who did not send messages

Example 6-10 shows how to employ set operations in Python to compute on data.

Example 6-10. Analyzing senders and receivers with set operations

```
senders = set(senders)
receivers = set(receivers)
cc_receivers = set(cc_receivers)
bcc_receivers = set(bcc_receivers)

# Find the number of senders who were also direct receivers

senders_intersect_receivers = senders.intersection(receivers)

# Find the senders that didn't receive any messages

senders_diff_receivers = senders.difference(receivers)

# Find the receivers that didn't send any messages

receivers_diff_senders = receivers.difference(senders)

# Find the senders who were any kind of receiver by
# first computing the union of all types of receivers
```

```
all_receivers = receivers.union(cc_receivers, bcc_receivers)
senders_all_receivers = senders.intersection(all_receivers)

print "Num senders in common with receivers:", len(senders_intersect_receivers)
print "Num senders who didn't receive:", len(senders_diff_receivers)
print "Num receivers who didn't send:", len(receivers_diff_senders)
print "Num senders in common with *all* receivers:", len(senders_all_receivers)
```

The following sample output from this script reveals some additional insight about the nature of the mailbox data:

```
Num senders in common with receivers: 3220
Num senders who didn't receive: 4445
Num receivers who didn't send: 18942
Num senders in common with all receivers: 3440
```

In this particular case, there were far more receivers than senders, and of the 7,665 senders, only about 3,220 (less than half) of them also received a message. For arbitrary mailbox data, it may at first seem slightly surprising that there were so many recipients of messages who didn't send messages, but keep in mind that we are only analyzing the mailbox data for a small group of individuals from a large corporation. It seems reasonable that lots of employees would receive "email blasts" from senior management or other corporate communications but be unlikely to respond to any of the original senders.

Furthermore, although we have a mailbox that shows us messages that were both outgoing and incoming among a population spanning not just Enron but the entire world, we still have just a small sample of the overall data, considering that we are looking at the mailboxes of only a small group of Enron employees and we don't have access to any of the senders from other domains, such as *bob@example1.com* or *jane@example2.com*.

The tension this latter insight delivers begs an interesting question that is a nice follow-up exercise in our quest to better understand the inbox: let's determine how many senders and recipients were Enron employees, based upon the assumption that an Enron employee would have an email address that ends with *@enron.com*. Example 6-11 shows one way to do it.

Example 6-11. Finding senders and receivers of messages who were Enron employees

```
# In a Mongo shell, you could try this query for the same effect:
# db.mbox.find({"To" : {"$regex" : /.*enron.com.*/i} },
#              {"To" : 1, "_id" : 0})

senders = [ i
            for i in mbox.distinct("From")
                if i.lower().find("@enron.com") > -1 ]

receivers = [ i
              for i in mbox.distinct("To")
```

```
                  if i.lower().find("@enron.com") > -1 ]

cc_receivers = [ i
                 for i in mbox.distinct("Cc")
                     if i.lower().find("@enron.com") > -1 ]

bcc_receivers = [ i
                  for i in mbox.distinct("Bcc")
                      if i.lower().find("@enron.com") > -1 ]

print "Num Senders:", len(senders)
print "Num Receivers:", len(receivers)
print "Num CC Receivers:", len(cc_receivers)
print "Num BCC Receivers:", len(bcc_receivers)
```

Sample output from the script follows:

```
Num Senders: 3137
Num Receivers: 16653
Num CC Receivers: 4890
Num BCC Receivers: 4890
```

The new data reveals that 3,137 of the original 7,665 senders were Enron employees, which implies that the remaining senders were from other domains. The data also reveals to us that these approximately 3,000 senders collectively reached out to nearly 17,000 employees. A *USA Today* analysis of Enron, "The Enron scandal by the numbers," reveals that there were approximately 20,600 employees at Enron at the time, so we have have upward of 80% of those employees here in our database.

At this point, a logical next step might be to take a particular email address and zero in on communications involving it. For example, how many messages in the data set originated with Enron's CEO, Kenneth Lay? From perusing some of the email address nomenclature in the enumerated outputs of our scripts so far, we could guess that his email address might simply have been *kenneth.lay@enron.com*. However, a closer inspection[3] reveals a few additional aliases that we'll also want to consider. Example 6-12 provides a starting template for further investigation and demonstrates how to use MongoDB's `$in` operator to search for values that exist within a list of possibilities.

Example 6-12. Counting sent/received messages for particular email addresses

```
import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo

client = pymongo.MongoClient()
```

3. In this particular case, a "closer inspection" was simply a search for "lay@enron" (`grep 'lay@enron'*` in a Unix or Linux terminal) on the *ipynb/resources/ch06-mailboxes/data/enron_mail_20110402/enron_data/maildir/lay-k/inbox* directory, which revealed some of the possible email aliases that might have existed.

```
db = client.enron
mbox = db.mbox

aliases = ["kenneth.lay@enron.com", "ken_lay@enron.com", "ken.lay@enron.com", 
           "kenneth_lay@enron.net", "klay@enron.com"] # More possibilities?

to_msgs = [ msg 
            for msg in mbox.find({"To" : { "$in" : aliases } })]

from_msgs = [ msg 
         for msg in mbox.find({"From" : { "$in" : aliases } })]

print "Number of message sent to:", len(to_msgs)
print "Number of messages sent from:", len(from_msgs)
```

Sample output from the script is a bit surprising. There are virtually no messages in the subset of the corpus that we loaded that were sent from one of the obvious variations of Kenneth Lay's email address:

```
Number of message sent to: 1326
Number of messages sent from: 7
```

It appears as though there is a substantial amount of data in the Enron corpus that was sent *to* the Enron CEO, but few messages that were sent *from* the CEO—or at least, not in the *inbox* folder that we're considering.[4] (Bear in mind that we opted to load only the portion of the Enron data that appeared in an *inbox* folder. Loading more data, such as the messages from *sent items*, is left as an exercise for the reader and an area for further investigation.) The following two considerations are left for readers who are interested in performing intensive analysis of Kenneth Lay's email data:

- Executives in large corporations tend to use assistants who facilitate a lot of communication. Kenneth Lay's assistant was Rosalee Fleming, who had the email address *rosalee.fleming@enron.com*. Try searching for communications that used his assistant as a proxy.
- It is possible that the nature of the court case may have resulted in considerable data redactions due to either relevancy or (attorney-client) privilege.

If you are reading along carefully, your mind may be racing with questions by this point, and you probably have the tools at your fingertips to answer many of them—especially if you apply principles from previous chapters. A few questions that might come to mind at this point include:

4. A search for "kenneth.lay@enron.com" (`grep -R "From: kenneth.lay@enron.com" *` on a Unix or Linux system), and other email alias variations of this command that may have appeared in mail headers in the *ipynb/resources/ch06-mailboxes/data/enron_mail_20110402/enron_data/maildir/lay-k* folder of the Enron corpus, turned up few results. This suggests that there simply is not a lot of outgoing mail data in the part of the Enron corpus that we are focused on.

- What are some of these messages about, based on what the content bodies say?
- What was the maximum number of recipients on a message? (And what was the message about?)
- Which two people exchanged the most messages? (And what were they talking about?)
- How many messages were person-to-person messages? (Single sender to single receiver or single sender to a few receivers would probably imply a more substantive dialogue than "email blasts" containing company announcements and such things.)
- How many messages were in the longest reply chain? (And what was it about?)

The Enron corpus has been and continues to be the subject of numerous academic publications that investigate these questions and many more. With a little creativity and legwork provided primarily by MongoDB's `find` operator, its data aggregations framework, its indexing capabilities, and some of the text mining techniques from previous chapters, you have the tools you need to begin answering many interesting questions. Of course, we'll only be able to do so much analysis here in the working discussion.

6.3.3. Writing Advanced Queries

MongoDB's powerful aggregation framework was introduced in version 2.2. It's called an *aggregation* framework because it is designed to allow you to compute powerful aggregates (as opposed to more primitive queries where you're basically filtering over the data) that involve pipelines of groupings, sorts, and more—all entirely within the MongoDB database instead of your having to dispatch and intermediate multiple queries from your Python script. Like any framework, it's not without its own limitations, but the general pattern for querying MongoDB is fairly intuitive once you have worked through an example or two. The aggregation framework lends itself to constructing queries one step at a time since an aggregate query is a sequence of steps, so let's take a look at an example that seeks to discover the recipients of a message sent by a sender. In a MongoDB shell, one possible interpretation of such a query could look like this:

```
> db.mbox.aggregate(
  {"$match" : {"From" : "kenneth.lay@enron.com"} },
  {"$project" : {"From" : 1, "To" : 1} },
  {"$group" : {"_id" : "$From", "recipients" : {"$addToSet" : "$To" } } }
  )
```

The query consists of a pipeline comprising three steps, where the first involves using the `$match` operator to find any message that is sent from a particular email address. Keep in mind that we could have used MongoDB's `$in` operator, as in Example 6-12, to provide an expanded list of options for the match. In general, using `$match` as early as possible in an aggregation pipeline is considered a best practice, because it narrows down the list of possibilities for each stage later in the pipeline, resulting in less overall computation.

The next step of the pipeline involves using `$project` to extract only the `From` and `To` fields of each message, since our result set requires knowing only the senders and recipients of messages. Also, as we'll observe in the next step, the `From` field is used as the basis of grouping with the `$group` operator to effectively collapse the results into a single list.

Finally, the `$group` operator, as just alluded to, specifies that the `From` field should be the basis of grouping and the value of the `To` field contained in the results of the grouping should be added to a set via the `$addToSet` operator. An abbreviated result set follows to illustrate the final form of the query. Notice that there is only one result object—a single document containing an `_id` and a `recipients` field, where the `recipients` field is a list of lists that describe each set of recipients with whom the sender (identified by the `_id`) has communicated:

```
{
  "result" : [
    {
      "_id" : "kenneth.lay@enron.com",
      "recipients" : [
        [
          "j..kean@enron.com",
          "john.brindle@enron.com"
        ],
        [
          "e..haedicke@enron.com"
        ],

               ...2 more very large lists...

        [
          "mark.koenig@enron.com",
          "j..kean@enron.com",
          "pr<.palmer@enron.com>",
          "james.derrick@enron.com",
          "elizabeth.tilney@enron.com",
          "greg.whalley@enron.com",
          "jeffrey.mcmahon@enron.com",
          "raymond.bowen@enron.com"
        ],
        [
          "tim.despain@enron.com"
        ]
      ]
    }
  ],
  "ok" : 1
}
```

The first time you see a query using the aggregation framework, it may feel a bit daunting, but rest assured that a few minutes fiddling around in the MongoDB shell will go a long way toward breeding familiarity with how it works. It is highly recommended that you take the time to try running each stage of the aggregated query to better understand how each step transforms the data from the previous step.

You could easily manipulate the data structure computed by MongoDB with Python, but for now let's consider one other variation for the same query, primarily to introduce an additional operator from the aggregation framework called `$unwind`:

```
> db.mbox.aggregate(
  {"$match" : {"From" : "kenneth.lay@enron.com"} },
  {"$project" : {"From" : 1, "To" : 1} },
  {"$unwind" : "$To"},
  {"$group" : {"_id" : "From", "recipients" : {"$addToSet" : "$To"}} }
  )
```

Sample results for this query follow:

```
{
  "result" : [
    {
      "_id" : "kenneth.lay@enron.com",
      "recipients" : [
        "john.brindle@enron.com",
        "colleen.sullivan@enron.com",
        "richard.shapiro@enron.com",

                ...many more results...

        "juan.canavati.@enron.com",
        "jody.crook@enron.com"
      ]
    }
  ],
  "ok" : 1
}
```

Whereas our initial query that didn't use `$unwind` produced groupings that corresponded to the particular recipients of each message, `$unwind` creates an intermediate stage of the following form, which is then passed into the `$group` operator:

```
{
  "result" : [
    {
      "_id" : ObjectId("51a983dae391e8ff964bc85e"),
      "From" : "kenneth.lay@enron.com",
      "To" : "tim.despain@enron.com"
    },
    {
      "_id" : ObjectId("51a983dae391e8ff964bdbc1"),
      "From" : "kenneth.lay@enron.com",
      "To" : "mark.koenig@enron.com"
    },

        ...many more results...

    {
      "_id" : ObjectId("51a983dde391e8ff964c241b"),
      "From" : "kenneth.lay@enron.com",
      "To" : "john.brindle@enron.com"
    }
  ],
  "ok" : 1
}
```

In effect, `$unwind` "unwinds" a list by taking each item and coupling it back with the other fields in the document. In this particular case the only other field of note was the `From` field, which we'd just projected out. After the unwinding, the `$group` operator is then able to group on the `From` field and effectively roll all of the recipients into a single list. The list has exactly the same size as the number of results that were passed into it, which means that there could be (and probably are) some duplicates in the final result. However, a subtle but important point is that the `$addToSet` modifier in the next `$group` operation eliminates duplicates since it treats the list it constructs as a set; therefore, the results of this query are ultimately a list of the unique recipients. Example 6-13 illustrates these queries with PyMongo as a wrap-up to this brief overview of MongoDB's aggregation framework and how it can help you to analyze data.

Example 6-13. Using MongoDB's data aggregation framework

```
import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo

# The basis of our query
FROM = "kenneth.lay@enron.com"

client = pymongo.MongoClient()
db = client.enron
mbox = db.mbox

# Get the recipient lists for each message

recipients_per_message = db.mbox.aggregate([
  {"$match" : {"From" : FROM} },
  {"$project" : {"From" : 1, "To" : 1} },
  {"$group" : {"_id" : "$From", "recipients" : {"$addToSet" : "$To" } } }
])['result'][0]['recipients']

# Collapse the lists of recipients into a single list

all_recipients = [recipient
                  for message in recipients_per_message
```

```
                      for recipient in message]

# Calculate the number of recipients per sent message and sort

recipients_per_message_totals = \
  sorted([len(recipients)
   for recipients in recipients_per_message])

# Demonstrate how to use $unwind followed by $group to collapse
# the recipient lists into a single list (with no duplicates
# per the $addToSet operator)

unique_recipients = db.mbox.aggregate([
  {"$match" : {"From" : FROM} },
  {"$project" : {"From" : 1, "To" : 1} },
  {"$unwind" : "$To"},
  {"$group" : {"_id" : "From", "recipients" : {"$addToSet" : "$To"}} }
])['result'][0]['recipients']

print "Num total recipients on all messages:", len(all_recipients)
print "Num recipients for each message:", recipients_per_message_totals
print "Num unique recipients", len(unique_recipients)
```

The sample output from this query is a bit surprising and reveals that just a few messages were sent by this individual (as we noted earlier), ranging from a couple of exchanges with a single individual to a rather large email blast to a huge email blast sent to nearly 1,000 people:

```
Num total recipients on all messages: 1043
Num recipients for each message: [1, 1, 2, 8, 85, 946]
Num unique recipients 916
```

Note the peculiarity that the number of total recipients in the largest email blast (946) is higher than the total number of unique recipients (916). Any guesses on what happened here? A closer inspection of the data reveals that the large email blast to 946 people contained 65 duplicate recipients. It's likely the case that the administrative work involved in managing large email lists is prone to error, so the appearance of duplicates isn't all that surprising. Another possibility that could explain the duplicates, however, is that some of the messages may have been sent to multiple mailing lists, and certain individuals may appear on multiple lists.

6.3.4. Searching Emails by Keywords

MongoDB features a number of powerful indexing capabilities, and version 2.4 introduced a new full-text search feature that provides a simple yet powerful interface for keyword search on indexable fields of its documents. Although it's a new feature that's still in early development as of v2.4 and some caveats still apply, it's nonetheless worthwhile to introduce, because it allows you to search on the content of the mail messages as well as any other field. The virtual machine for this book comes with MongoDB already configured to enable its text search feature, so all that's required to build a text index is to use the ensureIndex method on a collection in your database.

In a MongoDB shell, you'd simply type the following two commands to build a single full-text index on all fields and then view the statistics on the database to see that it has indeed been created, and its size. The special $** field simply means "all fields," the type of the index is "text," and its name is "TextIndex." For the mere cost of 220 MB of disk space, we have a full-text index to run any number of queries that can return documents to help us zero in on particular threads of interest:

```
> db.mbox.ensureIndex({"$**" : "text"}, {name : "TextIndex"})
> db.mbox.stats()
{
  "ns" : "enron.mbox",
  "count" : 41299,
  "size" : 157744000,
  "avgObjSize" : 3819.5597956366983,
  "storageSize" : 185896960,
  "numExtents" : 10,
  "nindexes" : 2,
  "lastExtentSize" : 56438784,
  "paddingFactor" : 1,
  "systemFlags" : 0,
  "userFlags" : 1,
  "totalIndexSize" : 221463312,
  "indexSizes" : {
    "_id_" : 1349040,
    "TextIndex" : 220114272
  },
  "ok" : 1
}
```

Creation of the index and querying of the statistics works pretty much just as you'd expect with PyMongo, as shown in Example 6-14, except that the syntax is slightly different between the MongoDB shell's ensureIndex and the PyMongo driver's ensure_index. Additionally, the way that you'd query a database for statistics or run a query on the text index varies slightly and involves PyMongo's command method. In general, the command method takes a command as its first parameter, a collection name as its second parameter, and a list of relevant keyword parameters as additional parameters if you are trying to correlate its syntax back to the MongoDB shell (which provides some syntactic sugar).

Example 6-14. Creating a text index on MongoDB documents with PyMongo

```
import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo

client = pymongo.MongoClient()
db = client.enron
```

```
mbox = db.mbox

# Create an index if it doesn't already exist
mbox.ensure_index([("$**", "text")], name="TextIndex")

# Get the collection stats (collstats) on a collection
# named "mbox"
print json.dumps(db.command("collstats", "mbox"), indent=1)

# Use the db.command method to issue a "text" command
# on collection "mbox" with parameters, remembering that
# we need to use json_util to handle serialization of our JSON
print json.dumps(db.command("text", "mbox",
                            search="raptor",
                            limit=1),
                 indent=1, default=json_util.default)
```

MongoDB's full-text search capabilities are quite powerful, and you should review the text search documentation to appreciate what is possible. You can search for any term out of a list of terms, search for specific phrases, and prohibit the appearance of certain terms in search results. All fields are initially weighted the same, but it is also even possible to weight fields differently so as to tune the results that may come back from a search.

In our Enron corpus, for example, if you were searching for an email address, you might want to weight the To: and From: fields more heavily than the Cc: or Bcc: fields to improve the ranking of returned results. If you were searching for keywords, you might want to weight the appearance of terms in the subject of the message more heavily than their appearance in the content of the message.

In the context of Enron, *raptors* were financial devices that were used to hide hundreds of millions of dollars in debt, from an accounting standpoint. Following are truncated sample query results for the infamous word *raptor*, produced by running a text query in the MongoDB shell:

```
> db.mbox.runCommand("text", {"search" : "raptor"})
{
  "queryDebugString" : "raptor||||||",
    "language" : "english",
    "results" : [
        {
      "score" : 2.0938471502590676,
      "obj" : {
        "_id" : ObjectId("51a983dfe391e8ff964c63a7"),
        "Content-Transfer-Encoding" : "7bit",
        "From" : "joel.ephross@enron.com",
        "X-Folder" : "\\SSHACKL (Non-Privileged)\\Shackleton, Sara\\Inbox",
        "Cc" : [
          "mspradling@velaw.com"
        ],
        "X-bcc" : "",
        "X-Origin" : "Shackleton-S",
        "Bcc" : [
          "mspradling@velaw.com"
        ],
        "X-cc" : "'mspradling@velaw.com'",
        "To" : [
          "maricela.trevino@enron.com",
          "sara.shackleton@enron.com",
          "mary.cook@enron.com",
          "george.mckean@enron.com",
          "brent.vasconcellos@enron.com"
        ],
        "parts" : [
          {
            "content" : "Maricela, attached is a draft of one of the...",
            "contentType" : "text/plain"
          }
        ],
        "X-FileName" : "SSHACKL (Non-Privileged).pst",
        "Mime-Version" : "1.0",
        "X-From" : "Ephross, Joel </O=ENRON/OU=NA/CN=RECIPIENTS/CN=JEPHROS>",
        "Date" : ISODate("2001-09-21T12:25:21Z"),
        "X-To" : "Trevino, Maricela </O=ENRON/OU=NA/CN=RECIPIENTS/CN=Mtr...",
        "Message-ID" : "<28660745.1075858812819.JavaMail.evans@thyme>",
        "Content-Type" : "text/plain; charset=us-ascii",
        "Subject" : "Raptor"
        },

            ...73 more results...

            {
      "score" : 0.5000529829394935,
      "obj" : {
        "_id" : ObjectId("51a983dee391e8ff964c363b"),
        "X-cc" : "",
        "From" : "sarah.palmer@enron.com",
        "X-Folder" : "\\ExMerge - Martin, Thomas A.\\Inbox",
        "Content-Transfer-Encoding" : "7bit",
        "X-bcc" : "",
        "X-Origin" : "MARTIN-T",
        "To" : [
          "sarah.palmer@enron.com"
        ],
        "parts" : [
          {
            "content" : "\nMore than one Enron official warned company...",
            "contentType" : "text/plain"
          }
        ],
        "X-FileName" : "tom martin 6-25-02.PST",
        "Mime-Version" : "1.0",
                "X-From" : "Palmer, Sarah </O=ENRON/OU=NA/CN=RECIPIENTS/CN=SPALME2>",
                "Date" : ISODate("2002-01-18T14:32:00Z"),
                "X-To" : "Palmer, Sarah </O=ENRON/OU=NA/CN=RECIPIENTS/CN=Spalme2>",
                "Message-ID" : "<8664618.1075841171256.JavaMail.evans@thyme>",
                "Content-Type" : "text/plain; charset=ANSI_X3.4-1968",
                "Subject" : "Enron Mentions -- 01/18/02"
            }
        }
    ],
    "stats" : {
        "nscanned" : 75,
        "nscannedObjects" : 0,
        "n" : 75,
        "nfound" : 75,
        "timeMicros" : 230716
    },
    "ok" : 1
}
```

Now that you're familiar with the term *raptor* as it relates to the Enron story, you might find the first few lines of a highly ranked message in the search results helpful as context:

> The quarterly valuations for the assets hedged in the Raptor structure were valued through the normal quarterly revaluation process. The business units, RAC and Arthur Andersen all signed off on the initial valuations for the assets hedged in Raptor. All the investments in Raptor were on the MPR and were monitored by the business units, and we prepared the Raptor position report based upon this information....

If you felt like you were swimming in a collection of thousands of messages and weren't sure where to start looking, that simple keyword search certainly guided you to a good starting point. The subject of the message just quoted is "RE: Raptor Debris." Wouldn't it be fascinating to know who else was in on that discussion thread and other threads about Raptor? You have the tools and the know-how to find out.

B-Trees Are the Bee's Knees?

B-trees are the fundamental underlying data structure for MongoDB and most other database systems, because they exhibit very efficient performance characteristics for core operations (searches, inserts, updates, deletes) over the long haul, even when continually faced with worst-case situations. In computer science terminology, their performance of their core operations is characterized as "logarithmic" and is expressed as $O(\log n)$ in the "Big-*O*" notation that was introduced in Chapter 3. B-trees necessarily remain balanced and maintain data in sorted order.

These characteristics yield efficient lookups because the underlying implementations require minimal disk reads. Given that disk seeks for traditional platter-based hard drives are still fast (on the order of low-single-digit milliseconds), this means that huge volumes of data as stored by B-trees can be accessed just as quickly. MongoDB heavily utilizes B-trees not only for its secondary indexes, which you can build on particular fields or combinations of fields, but also to back its geospatial indexes and its text search index, which was just introduced in this section.

In case you're wondering, there's no complete consensus about the etymology of the name *B-tree*, but it's generally accepted that the *B* stands for Bayer, the man who is credited with inventing them. There is more information about B-trees and their common variants online than you'd probably care to read. If you decide to take the deep dive into MongoDB, it's worthwhile to learn something about them at the theoretical and practical implementation levels since they are so integral to MongoDB's design.

6.4. Discovering and Visualizing Time-Series Trends

There are numerous ways to visualize mail data, and that topic has been the subject of many publications and open source projects that you can seek out for inspiration. The visualizations we've used so far in this book would also be good candidates to recycle. As an initial starting point, let's implement a visualization that takes into account the kind of frequency analysis we did earlier in this chapter with MongoDB and use IPython Notebook to render it in a meaningful way. For example, we could count messages by date/time range and present the data as a table or chart to help identify trends, such as the days of the week or times of the day that the most mail transactions happen. Other possibilities might include creating a graphical representation of connections among senders and recipients and filtering by keywords in the content or subject line of the messages, or computing histograms that show deeper insights than the rudimentary counting we accomplished earlier.

Example 6-15 demonstrates an aggregated query that shows how to use MongoDB to count messages for you by date/time components. The query involves three pipelines.

The first pipeline deconstructs the date into a subdocument of its components; the second pipeline groups based upon which fields are assigned to its `_id` and sums the count by using the built-in `$sum` function, which is commonly used in conjunction with `$group`; and the third pipeline sorts by year and month.

Example 6-15. Aggregate querying for counts of messages by date/time range

```
import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo

client = pymongo.MongoClient()
db = client.enron
mbox = db.mbox

results = mbox.aggregate([
{
  # Create a subdocument called DateBucket with each date component projected
  # so that these fields can be grouped on in the next stage of the pipeline
  "$project" :
  {
     "_id" : 0,
     "DateBucket" :
     {
       "year" : {"$year" : "$Date"},
       "month" : {"$month" : "$Date"},
       "day" : {"$dayOfMonth" : "$Date"},
       "hour" : {"$hour" : "$Date"},
     }
  }
},
{
  "$group" :
  {
    # Group by year and date by using these fields for the key.
    "_id" : {"year" : "$DateBucket.year", "month" : "$DateBucket.month"},

    # Increment the sum for each group by 1 for every document that's in it
    "num_msgs" : {"$sum" : 1}
  }
},
{
  "$sort" : {"_id.year" : 1, "_id.month" : 1}
}
])

print results
```

Sample output for the query, sorted by month and year, follows:

```
    {u'ok': 1.0,
     u'result': [{u'_id': {u'month': 1, u'year': 1997}, u'num_msgs': 1},
      {u'_id': {u'month': 1, u'year': 1998}, u'num_msgs': 1},
      {u'_id': {u'month': 12, u'year': 2000}, u'num_msgs': 1},
      {u'_id': {u'month': 1, u'year': 2001}, u'num_msgs': 3},
      {u'_id': {u'month': 2, u'year': 2001}, u'num_msgs': 3},
      {u'_id': {u'month': 3, u'year': 2001}, u'num_msgs': 21},
      {u'_id': {u'month': 4, u'year': 2001}, u'num_msgs': 811},
      {u'_id': {u'month': 5, u'year': 2001}, u'num_msgs': 2118},
      {u'_id': {u'month': 6, u'year': 2001}, u'num_msgs': 1650},
      {u'_id': {u'month': 7, u'year': 2001}, u'num_msgs': 802},
      {u'_id': {u'month': 8, u'year': 2001}, u'num_msgs': 1538},
      {u'_id': {u'month': 9, u'year': 2001}, u'num_msgs': 3538},
      {u'_id': {u'month': 10, u'year': 2001}, u'num_msgs': 10630},
      {u'_id': {u'month': 11, u'year': 2001}, u'num_msgs': 9219},
      {u'_id': {u'month': 12, u'year': 2001}, u'num_msgs': 4541},
      {u'_id': {u'month': 1, u'year': 2002}, u'num_msgs': 3611},
      {u'_id': {u'month': 2, u'year': 2002}, u'num_msgs': 1919},
      {u'_id': {u'month': 3, u'year': 2002}, u'num_msgs': 514},
      {u'_id': {u'month': 4, u'year': 2002}, u'num_msgs': 97},
      {u'_id': {u'month': 5, u'year': 2002}, u'num_msgs': 85},
      {u'_id': {u'month': 6, u'year': 2002}, u'num_msgs': 166},
      {u'_id': {u'month': 10, u'year': 2002}, u'num_msgs': 1},
      {u'_id': {u'month': 12, u'year': 2002}, u'num_msgs': 1},
      {u'_id': {u'month': 2, u'year': 2004}, u'num_msgs': 26},
      {u'_id': {u'month': 12, u'year': 2020}, u'num_msgs': 2}]}
```

As written, this query counts the number of messages for each month and year, but you could easily adapt it in a variety of ways to discover communications patterns. For example, you could include only the `$DateBucket.day` or `$DateBucket.hour` to count which days of the week or which hours of the day show the most volume, respectively. You may find ranges of dates or times worth considering as well; you can do this via the `$gt` and `$lt` operators.

Another possibility is to use modulo arithmetic to partition numeric values, such as hours of the day, into ranges. For example, consider the following key and value, which could be part of a MongoDB query document as part of the initial projection:

```
"hour" : {"$subtract" : [
                          {"$hour" : "$Date"},
                          {"$mod" : [{"$hour" :"$Date"} , 2]}
                        ]
```

This query partitions hours into two-unit intervals by taking the hour component from the date and subtracting 1 from its value if it does not evenly divide by two. Spend a few minutes with the sample code introduced in this section and see what you discover in the data for yourself. Keep in mind that the beauty of this kind of aggregated query is that MongoDB is doing all of the work for you, as opposed to just returning you data to process yourself.

Perhaps the simplest display of this kind of information is a table. Example 6-16 shows how to use the `prettytable` package, introduced in earlier chapters, to render the data so that it's easy on the eyes.

Example 6-16. Rendering time series results as a nicely displayed table

```
from prettytable import PrettyTable


pt = PrettyTable(field_names=['Year', 'Month', 'Num Msgs'])
pt.align['Num Msgs'], pt.align['Month'] = 'r', 'r'
[ pt.add_row([ result['_id']['year'], result['_id']['month'], result['num_msgs'] ])
  for result in results['result'] ]

print pt
```

A table lends itself to examining the volume of the mail messages on a monthly basis,[5] and it highlights an important anomaly: the volume of mail data for October 2001 was approximately three times higher than for any preceding month! It was in October 2001 when the Enron scandal was revealed, which no doubt triggered an immense amount of communication that didn't begin to taper off until nearly two months later:

```
+------+-------+----------+
| Year | Month | Num Msgs |
+------+-------+----------+
| 1997 |     1 |        1 |
| 1998 |     1 |        1 |
| 2000 |    12 |        1 |
| 2001 |     1 |        3 |
| 2001 |     2 |        3 |
| 2001 |     3 |       21 |
| 2001 |     4 |      811 |
| 2001 |     5 |     2118 |
| 2001 |     6 |     1650 |
| 2001 |     7 |      802 |
| 2001 |     8 |     1538 |
| 2001 |     9 |     3538 |
| 2001 |    10 |    10630 |
| 2001 |    11 |     9219 |
| 2001 |    12 |     4541 |
| 2002 |     1 |     3611 |
| 2002 |     2 |     1919 |
| 2002 |     3 |      514 |
| 2002 |     4 |       97 |
| 2002 |     5 |       85 |
| 2002 |     6 |      166 |
| 2002 |    10 |        1 |
```

5. The two messages that appear to have been authored in the year 2020 are the result of bugs in the original export of the mail data that are beyond our control.

```
| 2002 |    12 |         1 |
| 2004 |     2 |        26 |
| 2020 |    12 |         2 |
+------+-------+----------+
```

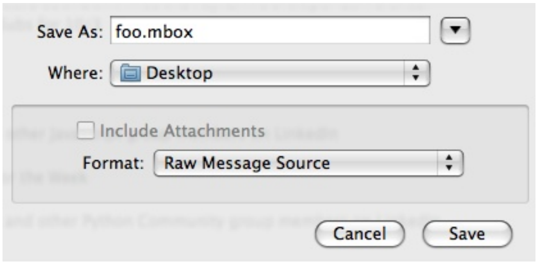
Applying techniques for analyzing the human language data (as introduced in previous chapters) for the months of October and November 2001 reveals some fundamentally different patterns in communication, both from the standpoint of senders and recipients of messages and from the standpoint of the words used in the language itself.

There are numerous other possibilities for visualizing mail data, as mentioned in the recommended exercises for this chapter. Using IPython Notebook's charting libraries might be among the next steps to consider.

6.5. Analyzing Your Own Mail Data

The Enron mail data makes for great illustrations in a chapter on mail analysis, but you'll probably also want to take a closer look at your own mail data. Fortunately, many popular mail clients provide an "export to mbox" option, which makes it pretty simple to get your mail data into a format that lends itself to analysis by the techniques described in this chapter.

For example, in Apple Mail, you can select some number of messages, pick "Save As…" from the File menu, and then choose "Raw Message Source" as the formatting option to export the messages as an mbox file (see Figure 6-2). A little bit of searching should turn up results for how to do this in most other major clients.



Save As: foo.mbox

Where: Desktop

Include Attachments [ ]

Format: Raw Message Source

Figure 6-2. Most mail clients provide an option for exporting your mail data to an mbox archive

If you exclusively use an online mail client, you could opt to pull your data down into a mail client and export it, but you might prefer to fully automate the creation of an mbox file by pulling the data directly from the server. Just about any online mail service will support POP3 (Post Office Protocol, version 3), most also support IMAP (Internet

Message Access Protocol), and it's not hard to whip up Python scripts for pulling down your mail.

One particularly robust command-line tool that you can use to pull mail data from just about anywhere is `getmail`, which turns out to be written in Python. Two modules included in Python's standard library, `poplib` and `imaplib`, provide a terrific foundation, so you're also likely to run across lots of useful scripts if you do a bit of searching online. `getmail` is particularly easy to get up and running. To retrieve your Gmail inbox data, for example, you just download and install it, then set up a *getmailrc* configuration file.

The following sample settings demonstrate some settings for a *nix environment. Windows users would need to change the `[destination] path` and `[options] message_log` values to valid paths, but keep in mind that you could opt to run the script on the virtual machine for this book if you needed a quick fix for a *nix environment:

```
[retriever]
type = SimpleIMAPSSLRetriever
server = imap.gmail.com
username = ptwobrussell
password = xxx

[destination]
type = Mboxrd
path = /tmp/gmail.mbox

[options]
verbose = 2
message_log = ~/.getmail/gmail.log
```

With a configuration in place, simply invoking `getmail` from a terminal does the rest. Once you have a local mbox on hand, you can analyze it using the techniques you've learned in this chapter. Here's what `getmail` looks like while it's in action slurping down your mail data:

```
$ getmail
getmail version 4.20.0
Copyright (C) 1998-2009 Charles Cazabon.  Licensed under the GNU GPL version 2.
SimpleIMAPSSLRetriever:ptwobrussell@imap.gmail.com:993:
  msg     1/10972 (4227 bytes) from ... delivered to Mboxrd /tmp/gmail.mbox
  msg     2/10972 (3219 bytes) from ... delivered to Mboxrd /tmp/gmail.mbox
  ...
```

6.5.1. Accessing Your Gmail with OAuth

In early 2010, Google announced OAuth access to IMAP and SMTP in Gmail. This was a significant announcement because it officially opened the door to "Gmail as a platform," enabling third-party developers to build apps that can access your Gmail data without you needing to give them your username and password. This section won't get