Plumbing the Big Data Pipeline

Joseph Clark

Last updated: March 23, 2015

Contents

Preface

The goal of this book is to organize lecture notes under development for a course at ASU. The course is officially called "Business Data Warehouses and Dimensional Modeling" but I have come to believe that data warehousing is one part of a larger discipline often called *data engineering* that supports the more glamorous work of data science and analytics.

In my terminology, *data science* refers to the work of using statistics, algorithms, visualization and other techniques to develop models that describe, explain, or predict patterns in data. It is agile, iterative work, usually carried out on a personal computer using relatively small datasets. Once a model is built, however, it often needs to be run on a larger dataset, either in a one-time batch process or as a regular, automated system. This requires the expertise of *data engineers* who know how to harness data streams, automate data preparation "pipelines", and implement the data scientist's models with cluster-based computing in the cloud.

The term *analytics* in my usage refers to the creation of automated, self-service systems that provide data-crunching capabilities to end users, such as reports and dashboards. Another common term for this is *business intelligence*. The users of these systems may be called analysts or data scientists, but the builders of these systems are data engineers, using some mix of relational databases, ETL tools, and new big data infrastructures.

The big idea behind my approach to this course is that students should finish the semester having some comfort with data in its different forms, recognize it when they see it, and know how to transform it, apply structure to it, query it, and create applications around it, as necessary for the job. After my class, the students will take courses in data visualization and data mining—traditional data science stuff—and I would like them to be prepared so that, when one of those professors asks them to get a dataset and prepare it in a certain way, they won't break into a sweat.

A Data Engineering Pyramid

The traditional relational database paradigm, which other textbooks teach first, actually represents a complex data structure for complex queries, and to the modern data engineer it should represent something of an "end goal" rather than a starting point. I propose a sort of Maslow's hierarchy of data engineering, in which we need to understand the basics of finding, storing, and transforming data, before we can engineer automated business intelligence systems built on RDBMSs.

Data Engineering Pyramid

1. What it means to persist data on disk, and what problems a database solves. Data formats like CSV, XML, JSON. What data looks like. Students should be able to "munge" a small data set into a preferred format with a one-time script.

2. How data is accessed through the Internet. Applications. Services. HTTP and REST. Scaling up and out, distributed systems, "the cloud". Students should be able to host data in the cloud and serve it up via a web server in a preferred format.

3. Data models. E-R modeling and logical database design for aggregate-oriented, relational, and graph databases. Students should be able to design tables and run ad-hoc queries, as well as create a database backend for their web service.

4. Big data analytics. Physical constraints and optimizations that are driving database design for big data. Differences between transactional and analytical systems. Students should be able to look at a transactional data model, identify likely challenges for analytical queries, and design a solution such as a dimensional DW or a Hadoop/MapReduce solution.

5. Data pipelines. Automating data ingest with streams. Automating data transformations, analyses, and loads. Creating a live big data application using cloud services.

The sequence of these notes represents my pedagogical thinking and is subject to change.

TODO: structure of the book, how to use it, etc

Recommended Viewing

- TODO: Some nice warm-up video? Maybe move "The Secret Life of Big Data" here from ch. 1 if nothing better is found.

Recommended Reading

- Clark, J., & Xu. A. (2014). "Plumbing for Philosophers: The Operations of a Data Science Team". Proceedings of the pre-ICIS SIGDSA Workshop.

Chapter 1

Atoms, Bytes, and Databases

Preview

This chapter lays the groundwork for learning about data management by introducing data storage as a physical technology, then discussing how data is logically organized as files. An example shows some of the practical problems of a simple file-based approach to data management. Databases are defined and it is argued that they can provide solutions to these problems, particularly the problem of program-data dependence. The net effect of this chapter is to set up the practical challenges that a data engineering discipline must address, which we will study in the remainder of this course.

1.1 Atoms

Your computer is a physical machine, made of atoms and charged with electrons. It is easy to lose sight of the artifact and think of the computer only in symbolic terms: windows, menus, programs, and so on. One component of your computer is "memory" (or RAM) which holds—in the form of electrical impulses—the instructions and data that your computer is currently processing. This memory fades away when the machine is powered off. Consequently, for you to be able to do any non-ephemeral work, the computer needs some means of saving its state or its output to *persistent storage*. Usually this means a device like a hard disk drive (HDD), but there are other options for persistent storage such as solid-state drives (SSD), optical disks (CDs/DVDs), thumb drives, and tape drives. Any of these devices can store files in such a way that they can be "remembered" (loaded into active memory) the next time the computer is powered up.

Table 1.1: Measures of data.

# of bytes	which equals...	common term	abbreviation
2^{10}	1024	kilobyte	KB
2^{20}	1,048,576	megabyte	MB
2^{30}	1,073,741,824	gigabyte	GB
2^{40}	1,099,511,627,776	terabyte	TB
2^{50}	about a quadrillion	petabyte	PB
2^{60}	about a quintillion	exabyte	EB
2^{70}	about a sextillion	zettabyte	ZB
2^{80}	about an octillion	yottabyte	YB

In order to really understand "big data", we need to always keep in mind that these systems have material properties and are affected by laws of physics; they are more than just software. The physical constraints on hard drives, processors, and networks have driven the evolution of relational databases, analytical systems, and cluster-based computing for big data. We will consider these constraints more in Chapter 7, but take a look at the videos I've recommended (at the end of this chapter) by Andrew Blum and Genevieve Bell—both are compelling talks about the unexpected and often overlooked physical and human aspects of big data and the Internet.

1.2 Bytes

Logically (now we are back in the realm of symbols), any persistent storage device, such as a hard disk, can be thought of as a long, long list of "ones" and "zeroes" which encode meaning. Eight of these digits or *bits* taken together make up a *byte*. We refer to 2^{10} bytes as a kilobyte, 2^{20} bytes as a megabyte, and so on.[1] Table 1.1 provides the common terms for various measures of data.[2]

The bits and bytes on your disk would simply be electronic gibberish if your operating system didn't know how to read them. Fortunately, these bytes are structured according to a *file system* so your operating system (e.g. Windows, Mac OS, or Linux) can read them and interpret the data as *files*. A file is just a chunk of bytes on disk that is given a name and a

[1] It's not exactly the metric system. 2^{10} is 1024, which is close but not precisely 1000.

[2] There's a petition going around to officially name 2^{90}—about a nonillion bytes—a "hellabyte". I'm all for it. We're going to have to start calling it *something* soon.

place in the structure of folders or directories. In this sense, programs like Microsoft Excel are files, and so are the spreadsheet documents they allow you to create. The former are files that contain instructions for the computer, and the latter are data files that contain program output. More commonly, we use the word "files" to refer to the latter type only, the data files.

Files (in the second sense) are created by software programs, or by humans using those programs. You can write a simple program in any programming language that reads a file or writes to a file. Excel, for example, writes a `.xlsx` file and knows how to read a file of that type. You can invent your own way of encoding data in the files that your program reads and writes. So who sets the standards about data formats? Anyone can, but Microsoft's standards are a lot more likely to be taken seriously by the market! If they change the way they save a spreadsheet file, you can bet that other companies will quickly update their software to be able to read the new file format.

Files are very versatile. They can big or small, can encode text, images, sound, video, and other types of data. You can transfer them from one disk to another, e-mail them to colleagues, and share them on the Internet. But if this was the end of the story, we'd quickly run into some problems.

1.3 The Trouble with Files

Imagine that you work at a small mail-order business where salesmen take orders over the phone, write them down on paper, and hand them to a data entry secretary so that the orders will be fulfilled and sales commissions will be paid. The secretary enters the handwritten data into a digital file—let's just say it's a spreadsheet. At the end of every day, she e-mails the file containing the day's orders to the fulfillment department, which processes the customers' payments and ships the orders. At the end of every week, she sends the seven daily files to the accountant, who uses them to compute the commissions that must be paid to the salesmen.[3]

There are a number of problems that the business is going to experience in this scenario:

1. The secretary is quickly going to find that she has a *lot* of files to keep track of, if she creates a new one for each day. If she keeps it all in one file, it's going to get *big* very quickly, and the two departments who use the data are going to find it hard to navigate.

[3]This is a real-life example, more or less. At my first real IT job, we had an order entry process like this, except instead of e-mail, the data was transferred from one person to another by floppy disk!

2. The file contains all the information from the paper order form. This is a huge security risk, even without the obvious threats of hacking or stolen laptops—the secretary is *routinely* sharing customer credit card numbers with the accountant, who has no business reason to see them. Although a file may be encrypted with a password, there is no finer-grained means of control that would enable the secretary to grant one user access to part of the data, another user access to the other part. It's all or nothing.

3. An alternative is to have the secretary enter the information twice: one file for the accountant, and one file for fulfillment, each file containing just what its intended users are allowed to see. This is a lot of extra work, and doubles the number of typos and other errors that will creep into the files.

4. If the business grows, and a second data entry secretary is hired, the two must coordinate their work somehow. If they both open the data file at the beginning of the new day, and add new rows for new orders, and then save their work to the same place (i.e. a network drive or Dropbox folder that they share), there's a real risk that the second one who hits "save" will overwrite and delete all the work of the first. Some strategy needs to be devised so that they work independently, then consolidate their work.

5. What if the fulfillment department needs to add their own annotations to the file, for example, noting when a payment was declined or an order was returned by the customer? We quickly run into a *versioning* problem, where the fulfillment department is making changes to the file they received yesterday, then sending them to the secretary who has already added new orders for today. She has to find the changes and integrate them into her new file, which may have already had other modifications made. This gets even more complicated when the accountant starts requesting updates on the old orders, so commissions may be adjusted if the customer returns an order three weeks after it was taken. In addition to providing the updated file, the secretary also needs to highlight *how* it was changed.

6. If the content or structure of the information captured on the paper order form changes, it's going to create real headaches for data entry and processing. If, for example, salesmen are assigned to different regional territories, and need to record "region" on the order form,

the secretary must add a new column to the spreadsheet. But this information is not present in all the historical records, which affects the accountant—her process for calculating sales and commission totals must change. Over time, the spreadsheet's structure may change several times, forcing everyone to keep learning new processes.

7. If, in time, the business decides to re-use the order data for some new purpose, an analytical purpose, such as customer relationship management or marketing research, the analysts are going to find themselves dealing with monstrous spreadsheets, probably multiple versions of some of the data, and old versions having different structure and definitions than newer versions. The files may be all but indecipherable to outsiders.

In the early days of business computing, problems like these showed the limits of using files alone for persisting data to disk. In review, these problems are:

1. Challenges of scale
2. Security and privacy issues
3. Redundancy and inconsistency
4. Challenges of coordination
5. Version control problems
6. Program-data dependence
7. Files don't describe themselves well

And so, the people of the information age embarked upon a quest for a better way to manage data. The solutions they developed were *databases*.

1.4 Databases

A database is defined by Oxford as "a structured set of data held in a computer, especially one that is accessible in various ways".[4] This is a good enough definition. A database management system (DBMS) is the software that both structures the data, and makes it accessible. From here on out,

[4] http://www.oxforddictionaries.com/us/definition/american_english/database

when I use the word "database", I am usually referring to the whole package (database and DBMS), as this is the common usage.

What you get with any kind of database is a software system specifically designed to keep track of data *and its meaning and structure* somewhat independently of other programs, and prepared to receive new data from, or provide access to stored data to, multiple users at the same time with different needs. At a minimum, a database is a software system that stores data along with metadata and has some kind of API[5] by which users can create, read, update, and delete[6] data.

The term *metadata* is often defined as "data about data" although I think "information about data" is more accurate. What it means is that, instead of just storing the data, the database can also inform its users about what the data is. If the data is `4809650024`, the metadata may be `Professor Clark's phone number`. There are many types of databases and therefore many types of metadata—we will explore these more beginning in chapter 5—but the point is that users of a database can make some sense of the data without knowing the process or program that created it. Consider the analyst in my example, who wants to use the historical order data but was not familiar with the order entry process. If the data were stored in some kind of database, he could use the metadata as a guide to know which number was sales, which was returns, and so on.

In addition to metadata, databases also need to present an interface to humans (and generally also to other software programs) by which they can access the data. For the past few decades, the best-known such interface is the structured query language SQL.[7]. SQL has four main commands: `INSERT`, `UPDATE`, and `DELETE`, which are used to manipulate data in the database, and `SELECT`, which is used to *query* the database, in other words, to request data for some purpose. Dialects of SQL with minor differences are used by most of the long-established database brands on the market—Oracle, IBM DB2, Microsoft SQL Server, PostgreSQL, MySQL, etc—so it has the benefit of being an industry standard.

Because SQL is a public interface, it means that multiple users and multiple programs can access the data in the same database. Salespeople may enter their orders in a mobile app, customers may place orders online through a website, secretaries may enter the data using a Microsoft Access form, the shipping department may receive the data via e-mail reports, and

[5] application programming interface

[6] known as the "CRUD" operations

[7] The name of SQL may be pronounced "sequel" or "ess, queue, ell"

the accountant may view it in a spreadsheet, but, since all of these programs are speaking to the database via SQL, it doesn't matter that they were all purchased at different times, programmed in different languages, and used by different users. Databases therefore offer the benefit of *program-data independence*: since the data has metadata and a stable interface, we can change a program or process that interacts with it (such as the structure of a data entry form) without worrying about disrupting every *other* program or process that uses it.

Databases can be designed to overcome each the other problems with a file-based approach that I identified earlier. Performance can be tuned as databases scale, controlling the levels of redundancy and structure depending on practical needs. Databases can apply fine-grained security policies to manage what each user can view or alter, as well as ensuring reliability with replication and backups. Databases can coordinate the actions of multiple concurrent users and enforce atomicity and integrity of transactions so that versions of the data do not get mixed up. New types of databases have emerged in the past few years to deal with the demands of "big data" and we will discuss them in this course. But the two universals are that databases can describe themselves via metadata, and grant program-data independence via stable APIs such as—but not limited to—SQL.

1.5 Data Engineering

The primary limitation on the database approach is that it requires discipline and expertise. Someone must create and maintain the database—and the server upon which it resides—so that other users can access it. Since all of the data is kept "in one basket", so to speak, backups must be made, and measures must be taken to protect the database servers from hackers as well as natural disasters. Performance optimizations must be made based on the scale of the database, the types of queries it receives, and other context. Moreover, someone has to make decisions about the structure and definitions of the data—for example, when is revenue counted: when the order is placed, when payment is received, or later when it is known that the customer will not return the merchandise? These are some of the responsibilities of database administrators (DBAs).

In the age of big data, a new role is emerging which takes a larger view of the flow of business data, a role which I call Data Engineering. For many years (from perhaps 1985 to 2005), data management primarily meant choosing a relational database brand and hiring DBAs who knew that brand

(e.g., Oracle). But now, primarily due to explosions in volume, velocity, and variety of data, a single centralized database can no longer be the entire picture of how data is organized and accessed in a business. Data engineers must consider the *flow* of data at Internet speed and at scales beyond what can be stored on a single server. Cluster-based computing presents new trade-offs that must be made between data consistency and response time. And the different uses of data have such radically different operational needs (transaction processing vs. big data analytics, for example), that different data models need to be employed at the same time for different purposes.

A data engineer therefore needs to know about how data is accessed and shared between users and applications over networks and in computer clusters. He needs to know about the different types of data models and which tasks they are best suited for. He needs to know about the new challenges of "big data" and the tools and techniques that are being developed to work with it. And finally he needs to integrate a variety of data management solutions into a data "pipeline", automate it, and maintain it. The remainder of this book surveys each of these knowledge areas.

1.6 Tutorial: Reading and Writing Data with Python

In this tutorial, you will learn to use Python to write and read a data file. Scripting languages like Python, Perl, R, and Ruby, are the "glue" that tie together the different parts of a big data pipeline. You will be challenged to come up with a better way to structure the data with its metadata.

Setting Up Python

The Python programming language is available for Windows, Mac OS, and Linux systems, from `http://www.python.org`. At the time of this writing, two versions of Python are current: Python 2.7.9 and Python 3.4.2. The code in this book is compatible with this or later versions of Python 3, but should work with any version of Python 2 as well with very few exceptions. For most systems, Python is an easy download from the website and an automatic install. Just follow the instructions given by the installer.

To follow the tutorials in this book, you need to be able to use a command line interface to your operating system. On a Mac or Linux computer the command line is called "Terminal". On a Windows PC, there are multiple options: you can type "cmd" in the Start menu to access the classic Windows command prompt, or you can use a modern one called Powershell, or you can use a Unix-style terminal such as Cygwin.

In my teaching, I will use a Unix-style command line interface on a Windows computer. That way, all students using Mac, Windows, or Linux computers should be able to follow along. The one I use is called Git Bash, and it is a free accessory when you install the version control software Git from `http://git-scm.com`. I recommend you (Windows users) do the same.

When you run Git Bash, you will see some welcome and help messages, followed by a *prompt* such as:

```
yourname@YOURCOMPUTER ~
$ _
```

The prompt gives us some useful information. It tells us who is logged in, and what directory/folder you're currently working in (the `~` is an abbreviation for the user's home directory). The prompt is customizable, and will probably appear differently to users on Mac and Linux systems. From here on out, in my examples I will simply use `$` to represent the command prompt. To execute a command, type it at the prompt and press Enter. Some of the commands you can try are `ls` (list files in the directory) and `cd Documents` (change to the "My Documents" folder on Windows).

Now, let's see if your computer is set up to use Python. At the prompt (represented here by the `$`), type:

```
$ python
```

If you got an error, first: did you type the `$`? You weren't supposed to type that! The `$` is just my stand-in for the command prompt which will be different on every student's computer. You should have typed the word `python` only, and then hit Enter.

If that's not the problem, then you might have gotten an error saying `"python not found"` or something similar. To fix this, you need to update your operating system's `PATH` variable. This is a setting that tells the computer where, when someone enters a command, to look for the program the user indicated. For a Windows user, Python 3.4 will be installed in the directory `C:\Python34\` and some other necessary scripts will be in `C:\Python34\Scripts\`. You'll need to find your existing `PATH` and add those two directories at the end of it. Instructions for Windows users can be found here: `http://www.computerhope.com/issues/ch000549.htm`. On a Macintosh I'm not quite sure where Python will be installed by default, but some instructions for setting the `PATH` are found here: `http://www.tech-recipes.com/rx/2621/os_x_change_path_environment_variable/`. You'll want to close your terminal and open a new one after changing the `PATH`.

If everything goes well, the next time you enter the `python` command, the Python interpreter will open. It will tell you the version of python you are running, and give you a new prompt, probably `>>>`. At this prompt, you can enter lines of Python code, but you cannot enter operating system commands. Try entering the following at the prompt. (Don't enter the `>>>`!)

```
>>> print('hello world')
```

The interpreter should process this line of code—your first Python program!—and the words "hello world" will appear on the line below it.

At any time you can type `quit()` to exit the Python interpreter and return to the operating system prompt.

Writing a Python program

A computer program in its most basic form is simply a list of instructions for the computer to follow. The interactive Python interpreter we've just used is great for testing things out or doing one-time calculations, but it will not be the main tool we use for writing programs. The reason is simple: every time we want to repeat the instructions, we need to re-type them. If your program is simply `print('hello world')` then this is feasible, but if you need 20 or 100 lines of code, it would be better to write them once and be able to re-run them.

You will therefore need a *text editor*, a software program for working with plain text files.[8] Your computer probably has a built-in text editor such as Notepad on a Windows system, but I recommend you select one that is meant for programming. Notepad++ (on Windows), TextWrangler (on Mac OS), and Sublime (on either platform) are very popular. They assist programmers by, for example, indicating line numbers, and using colors to make program structure easier to read.

In your chosen text editor, create a file containing the code below:

```
1 print('Hello world')
2 print('2+2 is', 2+2)
3 input('Press ENTER to continue...')
```

[8]This does *not* include word processors like Microsoft Word. They save their data in proprietary formats that are encoded for fonts, layouts, etc, and do not simply store unadorned plain text.

Save the file as `hello.py`. The "`.py`" extension to the filename tells Python and other programs that this is a Python program. Be careful that your text editor doesn't automatically add a "`.txt`" extension instead. To run your new program, navigate to the folder/directory where you saved the file, and type the following:

```
$ python hello.py
```

The program will run, print some sentences to the terminal, and ask you to press Enter. Once you do, you'll be returned to the command prompt (`$`). You are now set up to write any Python programs you want, and run them. If this doesn't work, you may still need practice using the command prompt, or may need help to set your `PATH` correctly. Also, make sure you're at the operating system command line (`$`) and not inside the Python interpreter (`>>>`) when you enter `python hello.py`.

Writing some data to a file

In our next program, we're going to write some data to a file in the *CSV* (comma-separated values) format. This is one way of organizing some data to share it with others. Its limitations will be considered in the next chapter. Within the code, I will use *comments*. These lines, beginning with `#`, are ignored by Python, so we use them to provide explanations for other people who may have to maintain or use our code.

Call this file `writefile.py`:

```
1  # open a new file in "write" mode
2  f = open("datafile.csv", "w")
3
4  # write headings, then some data
5  # separated by the newline symbol \n
6  # which is like pressing Enter
7  f.write("NAME,AGE,SEX\n")
8  f.write("Robert,37,M\n")
9  f.write("Kelly,29,F\n")
10
11 # finish writing the file
12 f.close()
13
14 input('Press ENTER to continue...')
```

Run it once, and you will see that a new file called `datafile.csv` has appeared in the folder/directory where you are working. You can run the script any number of times and it will re-create the same file. The way we opened the output file, with the "w" code, deletes the existing file and creates a new one every time it is run.

A very simple data entry system

Now let's expand on this program to allow interactive data entry. In `dataentry.py`, we use a `while` loop to ask for data until such time as the user leaves the name field blank (by pressing Enter twice). The condition `True` is always true, so the loop repeats itself infinitely until the `break` is executed. Note that in Python, the block of code within the loop is indicated by indentation, so you must indent each line after `while` the same way, and stop indenting when you come to the line containing `f.close()`.

```
f = open("datafile.csv", "w")
f.write("NAME,AGE,SEX\n")

while True:
    # True is always true; loop will run until we 'break'
    name = input("Name (or ENTER to quit): ")
    if name == "": break # end the loop if no name given
    age = input("Age: ")
    sex = input("Sex: ")
    # write one line of data to the file "f"
    f.write( name + "," + str(age) + "," + sex + "\n" )

# finish writing the file
f.close()
```

Were you able to follow everything that was going on in the code? Python is pretty close to English, so sometimes you can just guess. If you are struggling, you may want to search the Internet for help with the keywords `while` and `if`. Don't be shy about asking for help!

A system to read the data

You have stored your data in an organized way, using a commonly-known format (CSV). It even has a little bit of metadata—the first line contains

headers that tell us the names of the fields in each row: name, age, and sex. With `readdata.py`, we design a system to read the data and describe what the metadata says about it. It will print this to the screen.

Note that we use the same command, `open`, to open the CSV file, but this time with the code "`r`" for "read". Each line of the file is `strip`ped of its newline character and then `split` into a list of three items. In Python, as in most programming languages, the items in a list are accessed by number beginning with zero. So `headers[0]` is the first item in the list of headers, `headers[2]` is the third item, and so on. In this program, we use a `for` loop instead of a `while` loop; this is another very common structure.

```
print("Accessing data file...\n")

f = open("datafile.csv", "r")  # code "r" means "read"

# Ingest the whole first line.
# .strip() removes the "\n" character
# .split(",") turns the items into a list
headers = f.readline().strip().split(",")

# for each remaining line in the file, do this:
for line in f.readlines():
    line = line.strip().split(",")

    # headers and line are lists. we access their items
    # with numerical indexes, counting from zero

    print(headers[0],"is",line[0])
    print(headers[1],"is",line[1])
    print(headers[2],"is",line[2],"\n")

# finish reading the file
f.close()

input('Press ENTER to continue...')
```

This simple program cannot be said to truly "understand" the data, but it does make as much use of the metadata as possible, printing it to the screen and informing the user of how each piece of data is labeled. Congratulations! You've now developed software systems for gathering data, saving it to disk

in an organized way, and reading the data back from storage. The rest of the course will expand on this foundation.

Challenges

1. The program `readdata.py` can read any number of rows but only three columns of data (because that's how many we used in `dataentry.py`). Can you alter it to handle any arbitrary number of columns?

2. After doing the above, can you now extend `dataentry.py` to allow the user to input any number of columns? It should probably ask for the column labels first, and then begin regular data entry as before.

3. Our data entry program did no validation on the data that was entered, just accepted whatever the user typed. How could a user enter bad data that would crash our program. More interestingly... how might a clever user enter data that would trick the system without crashing it?

4. We read the CSV file line by line (i.e. row by row) and printed out the three variables in a row together. How could we read, or process, all the variables in a *column* together? For example, how might we compute a sum or average of all values in the "AGE" column?

5. CSV is a great format for tabular data in which each row has the same number of columns, but what if your data doesn't fit that criterion? For example, what if we wanted to add cars, or pets, or children's names to each row of data. There could be zero, one, or any number of these for each row. Design a data format that would allow us to store such data, and still incorporate metadata that tells us what it is. Can you write a script to produce your data file? Can you write a script to read it? What limitations does your new data file have?

Recommended Viewing

- "What is the Internet, really?", TED talk by Andrew Blum, author of "Tubes": http://youtu.be/XE_FPEFpHt4. (Or this longer version of the talk: http://youtu.be/28hzKbcLlWg.)

- "The Secret Life of Big Data", keynote talk at Supercomputing 2013 by Genevieve Bell of Intel. http://youtu.be/CNoi-XqwJnA

Recommended Reading

- Greenspun, P. (Accessed November 2014). SQL for Web Nerds. http://philip.greenspun.com/sql/.
- Hoffer, J. A., Topi, H., & Ramesh, V. (2014). Essentials of Database Management. Pearson.
- Lutz, M. (2009). Learning Python. O'Reilly.
- Manoochehri, M. (2014). Data Just Right: Introduction to Large-Scale Data & Analytics. Addison-Wesley.

Chapter 2

Structures for Data Interchange

Preview

Dear Students: please use chapter 2 of the textbook ("Data Just Right" by M. Manoochehri) until I have time to write this chapter!

TODO: A discussion of the structure of files. Binary vs text. ASCII vs Unicode. Peek into the structure of a Word or Excel file and compare it to some other formats. Contrast CSV, XML, JSON. Show alternative methods of serialization, such as Avro. Munge some data with Python: e.g., write a script to transform server logs into JSON. Also we could use Pig here to automate the transformations?

2.1 The Context of Data Sharing

TODO: Explain some of the constraints—networks are slow, so size matters, for example.

2.2 Text Formats for Data Files

TODO: Comparison of CSV, XML, and JSON formats for modeling/storing data.

2.3 New Approaches for Serialization

TODO: Talk about serialization techs like Avro.

2.4 Tutorial: Automating Data Transformation

TODO: Start with some data in a complex format like HTML (web scraping maybe?) and transform it to JSON or CSV using a Python script. Show them regular expressions and maybe unix tools like sed + grep. Finally, show how it's even easier in Pig. (Is it?)

Recommended Viewing

- TODO: Find some good videos–perhaps about data modeling in XML vs JSON?
- TODO: Good video on Pig? Maybe the Jurney video fits here?

Recommended Reading

- Jurney, R. (2014). Agile Data Science. O'Reilly.
- Manoochehri, M. (2014). Data Just Right: Introduction to Large-Scale Data & Analytics. Addison-Wesley.

Chapter 3

Opening Your Data to the World

Preview

This chapter introduces the HTTP protocol and its implications for communicating with data through the Internet. I discuss the challenges of statelessness and unreliability, safe and unsafe HTTP methods, and show how a RESTful architecture enables us to cope with them. This leads into a section on API design, and a tutorial in which we use Python and Flask to create a RESTful web service to share our data.

3.1 The language of the Internet

Many courses on databases or data management begin by assuming that you have certain software available on a certain type of computer, and proceed to tell you how to use it. On the contrary, we will assume that at some point you are going to have to communicate data over the Internet, whether you are providing data *to* others, collecting it *from* others, or both. You will want them to be able to build applications[1] that interact with your systems, but you cannot know exactly what software they will use, and they won't be able to directly access whatever database system you're using. The Internet itself determines some limitations on how you can communicate in data.

The language of the Internet which we are concerned with is the HyperText Transfer Protocol, *HTTP*. What it means to call it a "protocol"

[1] I don't like the word "apps" but you can use it if you like.

is that HTTP defines a structure for messages to be sent between clients and servers. It is not a programming language, or a specification for the underlying networking technologies.[2] Instead, HTTP gives us a formula for how to write a data message so that the person or computer on the other end of the line will know how to read it. HTTP was invented around 1990 by Tim Berners-Lee and his team along with HTML, the web server, and the first web browser—together creating the World Wide Web.

Two types of messages in HTTP are requests and responses. Typically, a client (such as your web browser) sends a request to a server (such as ASU's web server) and the server sends back a response (such as a web page, image, or PDF).

TODO: A diagram of the request/response process.

Every HTTP message has an initial line, which is different for requests and responses, optional "headers" (which are metadata about the message), and then the optional main body of the message, which may be any type of data. In the message body you can send text data like HTML (a web page), XML, or JSON, or you can send binary data like pictures in PNG format, documents in DOC or PDF format, and more.

TODO: Illustration of typical request message

The initial line for an HTTP request includes a method, a URL[3], and the HTTP version, for example:

```
GET /path/to/file/index.html HTTP/1.1
```

The most common method you use when browsing the web is `GET`, which simply says, "send me this resource", and doesn't change data or have other side effects on the server. You sometimes use `POST` for submitting forms, commenting on blogs and so forth—this is a method that adds data to the server. You might use `PUT` when uploading files such as photos, and there are several other methods that are used from time to time.

The initial line for a response from a server includes the HTTP version, a response code, and an English "reason phrase", for example:

```
HTTP/1.0 200 OK
```

or

```
HTTP/1.0 404 Not Found
```

Most communication on the internet occurs in this pattern of requests

[2]Networking is a huge topic, including aspects of hardware and software, beyond the scope of these notes. Assume that the infrastructure exists for sending messages between computers. What HTTP does is tell us how to structure those messages so the machine on the other end knows how to read them.

[3]URL stands for "Uniform Resource Locator" and is the address of the resource (such as a file) that you are requesting.

and responses. The requests often include metadata ("headers") which may identify the person who made the request and the browser version he is using. Response headers may include the date of the response, and if there is a message body such as a web page, will tell you the type of data to expect, and its length. Several other headers may be useful to us in building systems to communicate data over the Internet, and I will discuss them below.

3.2 Communicating in Data

One problem we face in sharing data over the Internet is that the networks are unreliable. Sometimes a message will not reach its intended recipient. In other cases, the response may be delayed and the requester may repeat his request, so the server receives the same message twice. These kinds of problems can have severe side effects—think of a credit card being charged twice for one purchase—and in designing for data communications over the Internet we need to consider them.

Another problem is that communication via HTTP is *stateless*. If I make two requests to a web server, the server doesn't automatically know that the two requests came from the same person, or whether they were part of the same "session". This makes it difficult to know if a person is "logged in" when they are trying to access sensitive data, and it complicates the implementation of something you see on a lot of web sites: a "next page" or "more results" button. If the server doesn't remember what I searched for, or which page of results I saw last, it won't know which data is "next".

Many solutions have been used to overcome these and other problems in the design of web applications. Some simple solutions are easy to imagine, but they have the feeling of rowing a boat upstream—they fight against the nature of the Internet rather than adapting to it. For example:

- session ID and page number in URL
- cookies to ID users and sessions
- timestamp to prevent duplicate POSTs
- TODO: more detail

3.3 REST to the Rescue

A more mature solution to the Internet's challenges is found in the Representational State Transfer or *REST* architecture, first proposed by Roy

Fielding in a dissertation at U.C. Irvine in 2000. Instead of fighting against the characteristics of HTTP and the Internet, this architecture is designed to take advantage of them.

According to a white paper by IBM's Alex Rodriguez, a modern implementation of REST architecture for a web service follows four basic design principles:

- Use directory-structure-like URLs mapped to resources.
- Map HTTP methods to CRUD functions.
- Design your applications to be stateless.
- Communicate in XML, JSON, or both.

The first of these principles requires that we think of our data as "resources" to be accessed, rather than programs to access them, and use URLs that seem to be addresses to those resources. The old, non-RESTful[4] way to implement comments on a blog (to give an example), might be to write a piece of code called `comment.php` and the URL to read the comments might be:

`http://www.myblog.com/comment.php?post=42\&action=read`

The RESTful way to address the comments for this article would be to use a URL like the following:

`http://www.myblog.com/articles/42/comments`

This implies that there is a folder or directory called "articles", and a folder for article #42 within it, and a folder of "comments" within that. On the server side, these folders need not actually exist, but the server should be programmed to receive a URL that looks like a folder structure and respond with the proper data. The simulated folder structure has a couple of benefits. First, it hides the details of how the site is programmed, and this means we could change the programming without changing the URL structure. Second, and more importantly, it allows our consumers to use the HTTP methods directly on the data (aka "resources") that they want, instead of indirectly on scripts.

The RESTful architecture maps the four main HTTP methods `POST`, `GET`, `PUT`, and `DELETE` to the four "CRUD" operations which are used in almost any application that operates on data. See Table 3.1. CRUD stands for Create, Read, Update, and Delete. It is almost universal that we will want to add new data to our system, read the data we have stored, change

[4] RESTless?

Table 3.1: Mapping of HTTP methods to CRUD operations

HTTP method	CRUD operation	Meaning
`GET`	Read	Fetch data without altering it.
`POST`	Create	Add this new data to the resource, creating the resource if it doesn't yet exist.
`PUT`	Update	Replace the existing resource with this (newer) data, creating it if it doesn't yet exist.
`DELETE`	Delete	Remove this resource.

existing data, and remove data from storage. Consider the contacts list on your phone—a simple database. You need to be able to add new contacts, browse them to find one you want to call, update them when friends get new phone numbers, and delete the old ones you never used.

Allowing our users (humans as well as software applications) to directly access data resources also helps us to achieve statelessness, and to mitigate some of the risks inherent in unreliable networks. Of the four main HTTP methods, one (`GET`) is "safe" meaning that by design it has no side effects on the server.[5] Two of the other methods (`PUT` and `DELETE`) are *idempotent*, meaning that they have the same effect—and no unintended side effects—whether they are executed once, twice, or a dozen times. There can still beside effects when using the `POST` method, such as adding the same blog comment twice, but we are at least removing the problems that occurred in the past when `GET` and `POST` were used for *everything* instead of only the operations proper to them.

Finally, web services must communicate in a standard data interchange format, usually JSON or XML, although the other formats discussed in the previous chapter may also be good candidates for the reasons stated there.

In sum, if we want people to be able to access our data over the Internet, we need to make it available via the HTTP methods, using sensible URLs that correspond to the data resources themselves. We need to receive requests in standard data formats, and send responses in standard data formats. If we are merely sharing our data, we can start by implementing the `GET`

[5]Except harmless ones like logs of web traffic, advertisements served, and so on.

method. The other HTTP methods are available if we want to create a truly interactive data application.

If we adhere to these principles, we gain a great deal of freedom! Since our URLs refer to resources rather than to pieces of code, we're free to use any technology we want to code our application, and to change it at any time without disrupting our users. Since our application is stateless and most of its methods are safe or idempotent, we can use cluster computing, caching, and other tricks on the back end to improve performance. (More on that in the next chapter.) By designing the RESTful interface or API *first* and then coding the implementation, we save a lot of time in programming, testing, and supporting customers.

3.4 Designing an API

Good API design begins with pencil and paper. An API can be thought of as a contract between you (the developer and owner of a data product) and your consumers (the people and products that will use your data). You are agreeing to make certain resources accessible in prescribed ways, and if you change the API unexpectedly, you may cause headaches for your consumers as their products and their businesses stop working. So let's first think about API design from the point of view of the consumer—what might he need from your data?

Imagine, for example, that your application is a Twitter-like microblog. The API should have a URL endpoint for each *entity* in the microblog's universe. These "entities" are its nouns; some are individual objects: users, messages, hashtags, etc., and others are *collections*, such as the set of all messages posted by a certain user. Your API should provide a way to access both objects and collections, and it should use the four main HTTP methods in the correct way, as its verbs.

An initial sketch of the API documentation might include the following:

`GET /users`: List all users of the system (usernames and real names)

`POST /users`: Create a new user

`GET /users/\{username\}`: Retrieve all data about the specified user

`GET /users/\{username\}/posts`: Retrieve all posts by this user

`DELETE /users/\{username\}`: Delete the specified user account

And similar methods would be documented for URL endpoints for messages, hashtags, and other data in the application. After the API endpoints, which correspond to resources, your API may provide a number of optional parameters to filter the response. For example, instead of all messages, your

user may only want the last 10, or only want messages posted during the last month. In a URL, these parameters follow a `?` and are separated by `&`. For example:

`GET /messages?limit=10`: Retrieve the last 10 messages.

`GET /messages?limit=10\&offset=10`: Retrieve 10 messages *after* the first 10 (in other words, retrieve "page 2" of messages).

`GET /messages?month=January\&year=2015`: Retrieve all messages from January 2015.

Some other good practices in setting up your API are to choose a good URL root. You could use something like `http://www.mysite.com/api/*` (the `*` is where your consumers put the specified URL endpoints), or you could use `http://api.mysite.com/*`. You may also want to include a version number, so that if you change the API, consumers can still use the old version. For example: `http://api.mysite.com/v1/*`.

Your API should also be well-documented so that your users don't have to figure it out for themselves. Where should you put this documentation? A good choice might be to put it in the form of a web page at the root URL, so that anyone with access to the API also has access to the documentation. Be sure to include sample URLs, as many people find it easier to modify the sample than to build their own from scratch.

For more on API design, see Recommended Reading below.

3.5 Tutorial: A RESTful Web Service

In this tutorial, we're going to set up an HTTP web service. A web service is like a website, except that it is intended to be used by machines—instead of stylish web pages with images, fonts, and colors, it will serve up data in JSON format at each URL we define. There are many ways to code a web service (or website) and there are web "frameworks" for almost every popular programming language.[6] In this exercise, I'll introduce a minimalistic web framework for Python called Flask.

Preparing a Python virtual environment

As we program this web service, we're going to install a number of "packages" or extensions to Python that are specific to this task, including the Flask framework. Over the course of the semester you will create a number of new

[6] Rails (for the Ruby language), ASP.NET (for C#), and Django (for Python) are some of the better known web frameworks.

Python programs, each depending on different packages. Rather than install them all in the same place (i.e., the folder where Python is installed), it is common practice to create a *virtual environment* for each new application you develop. The virtual environment will contain a *copy* of Python, plus any extensions you need for your app, and store them in the same folder as the code. Consequently, you can copy the folder to another computer or share it with someone else, and won't have to worry about whether they have the same version of Python or the same extensions – the ones you need will be there already.

To set up such an environment, you first need the `virtualenv` tool installed with Python. To install extensions to Python, we use the `pip` package manager. Try typing the following at the command line:

```
$ pip install virtualenv
```

Next, create a folder for your new application's code. This is going to be a LinkedIn-like web service that provides data on users' professional profiles and employment histories, so I'm calling it "flinkedin" and I'll put it in a folder with that name. You can call it whatever you like. These two commands in the shell will create a folder (`mkdir` is "make directory") and then navigate into it (`cd` is "change directory"):

```
$ mkdir flinkedin
$ cd flinkedin
```

Here's where we create the virtual environment:

```
$ virtualenv venv
```

This creates a new folder within the "flinkedin" folder, called "venv". If you explore in there, you'll see that it has installed a copy of Python and `pip`, among other things.

Now, to install Flask in the virtual environment's copy of Python, we need to run the virtual environment's copy of `pip`. It's in a subfolder called "bin" on Mac or Linux, and "Scripts" on Windows. Windows users, use this command:

```
$ venv/Scripts/pip install flask
```

Mac/Linux users instead use:

```
$ venv/bin/pip install flask
```

You should get a message stating that `pip` successfully installed `flask` and a number of other packages that it depends on. Now we have an environment, entirely contained within the "flinkedin" folder, that has the version of Python we want and the Flask package that we will use for this application.

Hello World, This is Flask!

The basic "Hello World" program for Flask is as follows. Save this code in the "flinkedin" folder and call it `app01.py`.

```
1  #!venv/bin/python
2  from flask import Flask
3
4  app = Flask(__name__)
5
6  @app.route("/")
7  def index():
8      return( "Hello World! This is Flask." )
9
10 if __name__ == '__main__':
11     app.run(debug=True)
```

Some of this is going to be repeated in every Flask application. Line 2, for example, tells Python that we're going to import something from the `flask` package that we installed. Lines 10-11 tell Python to start the web server when we run this piece of code by itself. The real interesting part is in lines 6-8, where we define a URL endpoint (the "/") and the code that is to run when someone accesses that URL. If our server name is `flinkedin.com`, the "/" designates the root URL, or `http://flinkedin.com/`.

Now, let's run the code. This should start a web server on your personal computer, accessible for now only by you. Note that we're using the copy of Python in the "venv" folder. Mac/Linux users, replace `Scripts` with `bin`.

```
$ venv/Scripts/python app01.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
```

Without closing the shell, open up a web browser and visit the website. You may type `http://127.0.0.1:5000` or `http://localhost:5000`. If all went well, the browser will show you the "Hello World" message.

Table 3.2: Flinkedin API Overview

Method	URL endpoint	Action
GET	/profiles	Retrieve list of profiles.
GET	/profile/<id>	Retrieve a specific profile.
POST	/profiles	Create a new profile.
PUT	/profile/<id>	Update an existing profile.
DELETE	/profile/<id>	Delete a profile.
GET	/companies	Retrieve list of companies.
GET	/companies/<id>/profiles	Retrieve list of profiles of people who have worked for a specific company.

Now, let's expand on the application to create a more complete URL structure. The API of our web service is going to support the URL patterns seen in Table 3.2.

For now, we'll simply create the URLs and some placeholder text at each. The code for `app02.py` follows.

```
1  #!venv/bin/python
2  from flask import Flask
3
4  app = Flask(__name__)
5
6  @app.route("/")
7  def index():
8      return( "Hello World! This is Flask." )
9
10 @app.route("/profiles", methods=['GET'])
11 def profiles():
12     return( "List of profiles" )
13
14 @app.route("/profile/<string:id>", methods=['GET'])
15 def get_profile(id):
16     return( "Profile " + id + " details" )
17
```

```
18 @app.route("/profiles", methods=['POST'])
19 def post_profile():
20     return( "Creating new profile" )
21
22 @app.route("/profile/<string:id>", methods=['PUT'])
23 def update_profile(id):
24     return( "Updating profile " + id )
25
26 @app.route("/profile/<string:id>", methods=['DELETE'])
27 def delete_profile(id):
28     return( "Deleting profile " + id )
29
30 @app.route("/companies")
31 def companies():
32     return( "List of companies" )
33
34 @app.route("/company/<string:id>/profiles")
35 def company_profiles(id):
36     return( "List users who worked for company " + id )
37
38 if __name__ == '__main__':
39     app.run(debug=True)
```

In `app02.py`, we've specified several new URL endpoints. In most cases we've indicated specific HTTP methods, so the same URL can be accessed with different methods and will trigger different parts of the code to run. Also, we sometimes use angle brackets in the URL to indicate a variable that may be captured by the code. For example in line 14, the URL is defined as `/profile/<string:id>`. If someone were to access `http://localhost:5000/profile/123`, the "123" would be captured as a string variable (i.e. text) called `id`. In our code, the function `get_profile(id)` would use the variable to construct this phrase: "Profile 123 details". Start the app and surf to that URL to see for yourself! Try some of the other URLs, too. (Of course, with your web browser you can only send `GET` requests.)

Now serving sample data

Now that we've got the URLs set up, let's create some actual data. We haven't covered databases yet, so we'll just hard-code some sample data into

the application.[7] Add the following code to your program and save it as `app03.py`.

```
# sample data
users = [
    {
        'id': '123',
        'name': 'John McLane',
        'jobs': [
            {'employer':'NYPD',
             'position':'Lieutenant',
             'start':'1988'}
        ]
    },
    {
        'id': '456',
        'name': 'James Edwards',
        'jobs': [
            {'employer':'NYPD',
             'position':'Officer',
             'start':'1990'},
            {'employer':'M.I.B.',
             'position':'Agent',
             'start':'1997'}
        ]
    }
]
```

You might have noticed that this data structure looks a lot like JSON! Just like Javascript, Python uses square brackets `[ ]` to delimit a list of similar things, and curly braces `{ }` to define a "dictionary" of named, dissimilar things. The items in a list can be accessed by number (e.g., `profiles[0]`) and the items in a dictionary can be accessed by name (e.g., `profiles[0]["id"]`). Commas are used to separate items in lists and dictionaries. Lists can be nested within dictionaries, and dictionaries can be nested within lists. To access Agent J's first employer's name, for example, you'd

[7] That means changes to the data will not be retained. Every time we re-start the program, it'll go back to the same initial data written into the code. In real applications, you'll want to use a database to persist the data to disk as it changes.

use `profiles[1]["jobs"][0]["employer"]`. It is very similar to Javascript in this regard. Other programming languages have similar notations.

Now, let's set up our first two `GET` methods to serve real data. We're going to need two additional functions that come with the Flask package: `jsonify` to print our data out as JSON, and `abort` to send 404 errors when necessary. Change the second line of the program like so:

```
2 from flask import Flask, jsonify, abort
```

The first generic method, to list all the profiles, uses the `jsonify` function to simply output the whole `users` data structure and give it the name "profiles":

```
35 @app.route("/profiles", methods=['GET'])
36 def profiles():
37     return( jsonify({'profiles':users}) )
```

The second method, to list a specific profile, has to use a little Python magic to find just the profile with the specified `id`. If there is no profile with that `id`, the data will have length of zero, and the program will return a 404 "Not Found" error which your web browser understands. If the profile exists, it will be returned as JSON.

```
39 @app.route("/profile/<string:id>", methods=['GET'])
40 def get_profile(id):
41     user = [user for user in users if user['id'] == id]
42     if len(user) == 0:
43         abort(404)
44     return( jsonify({'profile':user[0]}) )
```

Run the server (`venv/Scripts/python app03.py`) and point your web browser to a few URLs: `http://localhost:5000/profiles` to see all the users, `http://localhost:5000/profile/123` to see John McLane's profile, and `http://localhost:5000/profile/101` to make sure you get the expected 404 error.

Methods to modify the data

Next we'll implement some of the methods that modify the data. However, we won't be able to test these with a web browser, because it only sends

`GET` requests in normal circumstances. We'll use a utility called `curl` to send arbitrary HTTP requests without a browser. It should already be installed on a Mac or Linux machine, and on Windows if you have installed Git and are using Git Bash as your shell (as I recommended in Chapter 1). Type `curl --version` at your command line and, if it is installed, it'll print the version number and then exit.

Let's test it. Run the server (`venv/Scripts/python app03.py`) and, without stopping it, open a new terminal window. In the second shell, type:

```
$ curl -i http://localhost:5000/
```

What you'll see next is a message you should recognize as a successful HTTP response with an initial line, some headers, and the content of the message:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 27
Server: Werkzeug/0.10.1 Python/3.4.0
Date: Sun, 22 Mar 2015 23:19:31 GMT

Hello World! This is Flask.
```

Here's how you would issue a `POST` request with `curl`. It's pretty long for a one-liner, so I've used a backslash (\) to allow me to continue writing the command on a second line. (This is not needed on the second or third line because those are between quotes `''`.) In this command you see `-X` preceding the HTTP method we want to use, `-H` indicating a header we want on the message, and `-d` preceding the body of the message we are sending, which in this case is a JSON object that we want to add to our list of profiles. Finally, we type the URL.

```
$ curl -i -X POST -H "Content-Type: application/json" \
> -d '{ "id":"101", "name": "Samwise Gamgee", "jobs":
> [{"employer":"Bag End", "position":"Gardener",
> "start": "1954"}]}' http://localhost:5000/profiles
```

We haven't written the code yet to do anything with that `POST`, so you'll just get one of our placeholder text responses. Let's fix that now. Saving a copy of our applicaiton as `app04.py`, change the second line so that we're also importing `request`.

```
2  from flask import Flask, jsonify, abort, request
```

The following lines define a `post_profile()` function to accept the request and add the profile to our `users` data structure. If there is no JSON message attached to the request, it sends the user a 400 error, meaning "bad request". Otherwise it adds the received JSON to the list of profiles in memory. There's no database behind our application, so if the server is restarted it will lose any data you've added. (A database to persist the data to disk will be introduced in Chapter 5).

```
46 @app.route("/profiles", methods=['POST'])
47 def post_profile():
48     if not request.json:
49         abort(400) # 400 error = bad request
50     users.append(request.json)
51     return( jsonify({'profile':request.json}), 201 )
```

The response message should have HTTP code 201, meaning "created", and show you the new profile. You see, of course, that we never did any error checking to make sure that the profile had a properly formatted `id` number, or even a name. This is just a quick demo and a real-world web application would do lots of validation to preserve data integrity and security.

The `PUT` and `DELETE` methods combine aspects of what we've already developed. Like the `GET` method for an individual profile, they search the data for an `id` that matches the URL. Like the `POST` method, the `PUT` method checks for a JSON request message. Both of these methods delete the existing data, and the `PUT` method then appends the new data, essentially replacing the old data with the new. (Again, no real checking for data errors is done. This is for the classroom only!)

```
53 @app.route("/profile/<string:id>", methods=['PUT'])
54 def update_profile(id):
55     user = [user for user in users if user['id'] == id]
56     if len(user) == 0:
57         abort(404)
58     if not request.json:
59         abort(400)
60     users.remove(user[0])
```

```
61      users.append(request.json)
62      return( jsonify({'profile':request.json}) )
63
64  @app.route("/profile/<string:id>", methods=['DELETE'])
65  def delete_profile(id):
66      user = [user for user in users if user['id'] == id]
67      if len(user) == 0:
68          abort(404)
69      users.remove(user[0])
70      return( "Deleted profile " + id )
```

To test it, use the `PUT` method to change the M.I.B. agent's name to "J". This is a bit awkward because we have to re-write the whole profile in JSON, instead of just the part that we wanted:

```
$ curl -i -X PUT -H "Content-Type: application/json" \
> -d '{"id":"456","name":"J","jobs":[{"employer":"NYPD",
> "position":"Officer","start":"1990"},{"employer":
> "M.I.B","position":"Agent","start":"1997"}]}' \
> http://localhost:5000/profile/456
```

And since company policy, in fact, forbids keeping social media profiles, better just delete his profile entirely:

```
$ curl -i -X DELETE http://localhost:5000/profile/456
```

Visit `http://localhost:5000/profiles` to confirm it has been deleted.

Recap

In this tutorial we have used the Python web framework Flask to create an HTTP server that handles `GET` requests at a number of URL endpoints, and at least one type of `POST`, `PUT`, and `DELETE` request. We created a virtual Python environment (essentially, a private copy of Python and its extensions) in the same folder using `virtualenv`, and used `pip` to install the `flask` package we needed for this environment. Our data was hard-coded into our Python program (in a structure that looks rather similar to JSON) so that it is reset every time the server re-starts, with no persistence to disk. We used `curl` to send HTTP requests to the server to test it.

Challenges

1. Modify the `GET /profiles` method so that it doesn't give all of the details for each profile, instead, just list the `id` and name of each profile. Users of your API would then use `GET /profile/<id>` to drill down to the details.

2. Consider what happens when you have a large number of profiles. Will the response be too long? Create a new URL endpoint like `GET /profiles/pages/<page_num>`, which returns a "page" of results containing a limited number of profiles. For example, `/pages/1` might contain results 1-10 and `/pages/2` might contain results 11-20.

3. Create a new URL endpoint `POST /profile/<id>/jobs` so that someone can add a job to a specified profile without having to use `PUT` and re-type the entire profile.

4. Our data structure is essentially a list of profiles, with employers listed for each profile; this makes it easy to find a profile but more complicated to search for a company. Can you implement the `GET` method for the `/companies` URL, listing all the companies mentioned in our data?

5. As a follow-up to the previous challenge, try to implement the unfinished `/company/<id>/profiles` URL endpoint, listing all the profiles that include a given company.

6. In the tutorial for Chapter 1, you learned how to use Python to write and read files on the hard disk. Modify our application so that when the server is started, it loads the data from a file, and when `POST`, `PUT`, or `DELETE` methods are used, the data file is re-written. Now your application can be called a database: it has a way of organizing data on disk and an API for users to create, read, update, and delete data.

7. In addition to URL endpoints that return JSON, you might want to make a few human-readable web pages. Flask can serve up HTML pages with or without dynamic elements. If you know a little about web development, create a home page for your app. Save it as "index.html" in a folder called "templates" within the application folder. Add `render_template` to the `import` line, and change the `return()` line of the homepage route to: `return( render_template("index.html") )`. Now instead of a "Hello World" message your users will see an HTML page at the root URL.

Recommended Viewing

- TODO: Can I find some videos on good API design?

Recommended Reading

- Marshall, J. (2012). HTTP Made Really Easy: A Practical Guide to Writing Clients and Servers. http://www.jmarshall.com/easy/http/
- Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures. [Dissertation]. Chapter 5: Representational State Transfer (REST). http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- Rodriguez, A. (2008). RESTful Web Services: The basics. IBM developerWorks. http://www.ibm.com/developerworks/library/ws-restful/ws-restful-pdf.pdf
- Hunter, T. (2014). Consumer-Centric API Design. [e-book]. https://github.com/tlhunter/consumer-centric-api-design
- Grinberg, M. (2014). Designing a RESTful API with Python and Flask. http://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask

Chapter 4

Data at Internet Scale

Preview

Now we move up to the problems of scale in the internet era, which afflict single-node databases. Talk about distributed computing and massively-parallel processing. Explain how/why "the cloud" works. Discuss consistency and availability trade-offs with distributed data. For now,

4.1 Scaling Out

TODO: Sum up what we saw in the previous chapter as a "single node" web service with a nice diagram. Talk about the limitations of scaling "up" to larger and more expensive computers. Instead, the new paradigm is scaling "out" to large clusters of cheap commodity computers.

4.2 Trade-offs with Clusters

TODO: Talk about the CAP theorem; the trade-off between consistency and availability (or response time) is an instance of the more general trade-off between "safe" and "fast". Talk about architectural choices: sharding or replicating or both; master node or no; strong or eventual consistency; virtual machines or bare metal.

4.3 The Cloud

TODO: How the cloud evolved—mainly through Amazon's overcapacity due to seasonality (I think!)—and how it meshes so well with the distributed computing concept. Business benefits of using the cloud for storing and serving data. Maybe scare them a little by talking about what a chore "IT operations" can be if you do it yourself.

4.4 Tutorial: Taking our Web Service to the Cloud

In this tutorial, we'll take the Flask application that we developed in Chapter 3 and deploy it to the Amazon cloud, using a service called Elastic Beanstalk to manage the deployment. Once complete, the API you developed on your personal computer will be accessible to everyone through the Internet.

What you can get for your magic beans

Amazon Web Services (AWS) is one of the leading providers of IaaS and PaaS cloud services. [1] They offer services for storage, virtual machines for computing, and databases of various types. Since AWS's "menu" of services is continually growing, you may want to take a moment to go look at their web site (`http://aws.amazon.com/`) to see the latest. In particular, follow the link for "What is Cloud Computing?" and watch the three-minute video. AWS has a "free tier" for most of its services, so although you will need a credit card to sign up for an account, you will likely not be charged for any of the things we'll be doing in this course.[2]

With an account, you can provision a web server with some storage and perhaps a database service manually, but there is an easier option. Elastic Beanstalk is a service that manages applications in "containers". You tell it what type of container you need (e.g. Python), and Elastic Beanstalk will provision the servers, prepare the environment for your app, and deploy your code to production. After the initial setup, you only need to pay attention to the code, and let Amazon manage the rest. More information is available, including a helpful video, at `http://aws.amazon.com/elasticbeanstalk/`.[3]

[1] Its competitors include Microsoft Azure, Google Cloud Platform, and Rackspace.

[2] Also, AWS tends to charge based only on what you use, so if you need some paid services for a class assignment, you are likely to pay a few cents per hour instead of a few dollars per month. Check their website for the latest pricing.

[3] Some other services like Elastic Beanstalk include Heroku and IBM's BlueMix.

Preparing the app for deployment

When we deploy our web application through Elastic Beanstalk, it's going to need two things: our code, and a file called `requirements.txt` that tells it which Python packages are required. Do you remember that we set up a Python virtual environment for our app? That makes our task here extremely easy, because we intalled into the virtual environment only the Python packages we needed for this specific app. Other extensions we may have used in other apps are not included.

To create the `requirements.txt` file, change into the directory where our code is stored. In my case, that's a folder on my Desktop called "flinkedin":

```
$ cd Desktop/flinkedin
```

And now, using the copy of `pip` in our virtual environment (`venv`), we automatically create the `requirements.txt` file in this one-liner. Mac and Linux users, substitute "bin" for "Scripts":

```
$ venv/Scripts/pip freeze > requirements.txt
```

The contents of `requirements.txt`, if you look at them in a text editor, should look something like the following, although your version numbers may be different:

```
1 Flask==0.10.1
2 Jinja2==2.7.3
3 MarkupSafe==0.23
4 Werkzeug==0.10.1
5 itsdangerous==0.24
```

Finally, we will make some changes to our code to make Amazon happy. If you did the tutorial exactly as I did in Chapter 3, you have several versions of the program, the final one named `app04.py`. Make a new copy of this file called `application.py` because that's what AWS is looking for. (It is possible to configure AWS to accept a different program name, but easier to just go with the flow.) In `application.py`, you must change the variable `app` to `application`. Line 4 of the file becomes:

```
4 application = Flask(__name__)
```

Additionally, change `@app.route(...)` to `@application.route(...)` everywhere it appears.

Now, we need to make a zip file of our code. Enter the "flinkedin" folder with the file system, select the files `application.py` and `requirements.txt` but *not* the `venv` subfolder, and zip these files together. (On Windows, right-click the selection and Send to : Compressed (zipped) folder.) The zip file may have any name you prefer.

Journey to the Amazon

Now, you'll need to go through the process of creating an account with Amazon Web Services at `http://aws.amazon.com`. Once you've signed up, log in and visit the AWS Management Console, that is, the menu of all services offered by AWS. You should see Elastic Beanstalk in the list under the "Deployment & Management" subheading. Click on it.

1. In the navigation bar, click "Create New Application". Give your application a unique name and then click "Next".

2. If given a choice between a "Web Server Environment" and a "Worker Environment", choose the former. Click "Create web server".

3. On the next page, you'll be asked to choose the platform and type of environment. The platform you want is Python and the application type is "Load balancing, auto scaling".

4. On the next page you'll be asked for a source for your application version; this means your code. Choose "upload your own" and choose the zip file you created earlier, containing your code and `requirements.txt`. Click "Next".

5. You will be asked for an Environment name. Go with the defaults and click "Next".

6. On the "Additional Resources" page you'll be asked if you want a relational database (RDS DB) and some other options. Leave the boxes unchecked and click "Next".

7. On the "Configuration Details" page, leave the default choices and click "Next".

8. On the "Environment Tags" page, click "Next".

9. Finally, you will be allowed to review your selections. Unless something looks glaringly wrong, go ahead and click "Launch".

You will now have the pleasure of watching Elastic Beanstalk set up the environment and deploy your code. This may take several minutes. Good time to get some coffee. When the upload is complete, you should see a green checkbox indicating that the application is live. Near the top of the screen you'll see the name of the application and environment, with its URL in parentheses. Click the URL and you should see the "Hello World" message.

[figures/working_app_dashboard.png]

Managing your application

You have a number of tools at your disposal to manage your applicaiton from this dashboard. If you experienced an error and your application isn't working, for example, check out the "Logs" tab and request the last 100 log entries from your application. If you'd like to see how your servers are performing, click the "Monitoring" tab and see the information that's available to you. You also have the option of changing the details of your configuration. If you want to make some changes to the application and upload a new version, click the "Upload and Deploy" button.

TODO: This chapter could also use a nice discussion of Git for version control, in conjunction with the Elastic Beanstalk command line interface.

TODO: This chapter may need some updating as we discover the common errors that students tend to make.

Recommended Viewing

- TODO: Any good videos on CAP theorem which don't assume prior knowledge of relational databases? Otherwise use the Martin Fowler talk on NoSQL.

Recommended Reading

- Manoochehri, M. (2014). Data Just Right: Introduction to Large-Scale Data & Analytics. Addison-Wesley.
- Wilder, B. (2012). Cloud Architecture Patterns. O'Reilly.

- TODO: Some good tutorial on AWS and Elastic Beanstalk would be nice.

Chapter 5

A Multitude of Databases

Preview

Databases, that is, software systems that persist data for future retrieval, come in many forms because they face numerous challenges and trade-offs. They abstract away the physical storage and retrieval of data so that application developers can work with data models appropriate to their business rules and constraints. The relational data model has been dominant since the 1970s but now shares the field with a number of other types of data models. Databases introduce unique challenges for operations and maintenance.

5.1 Defining the Database

A database is a system that *persists* data for future retrieval. Since data is such a vital resource, databases are an important part of the infrastructure of a business. Other software applications depend on databases for access to the data they need. This means databases must deal with two particularly challenging constraints: (1) they cannot lose the data, and (2) they cannot be shut down. Designing and maintaining a database is sometimes likened to building and repairing a ship while in the middle of the ocean.

In the tutorial in Chapter 1, we created a few simple scripts to write a CSV file and to read and print the data to screen. The software we wrote, although crude, has one pretty interesting characteristic: we could restart the computer, and the data would still be there. It is persistent. In other kinds of software development, programmers can freely alter their code and test new features or bug fixes, iterating as many times as they like—but in

database development, the data persists. Since the data cannot be thrown out, database administrators and the developers of data-driven applications must work with the constraint that it must still use the existing data. We could re-write our scripts from the initial tutorial, but we're committed to the CSV format for the data storage unless we want to lose data. Therefore, choices made in the initial *data modeling* matter a great deal.

5.2 Data Models

What most real database management systems do is to provide a layer of abstraction on top of the physical storage of data. The abstraction is called a *data model* and there are several common ones. In the *relational data model*, data is stored in tables with rows and columns, with a few very simple rules:

- No two rows may be identical.
- No two columns may have the same name.
- No data field may have multiple values.
- There is no inherent order of rows or columns.

What it means to call this an *abstraction* is that application developers, or other users working with the data, will simply interact with tables, rows, and columns. The underlying implementation of data storage on disk will be handled by the database management system and will be invisible to users. One data table could be physically implemented on one disk or distributed across multiple disks, partitioned to keep rows together or to keep columns together, stored in the order data was added or in numerical or alphabetical order—but all of these details would be hidden from the user.

Databases are, however, what computer scientists call "leaky abstractions". In actual practice, there are choices you can make in physical implementation that will make a difference in terms of performance. Since these choices are fundamental to the emergence of analytical databases such as data warehouses, I will write more about them in Chapter 7.

To free the developers and users of data-driven applications from working directly with physical data storage on disk, databases offer APIs to enable database *queries*. These APIs may be RESTful HTTP interfaces, like the ones we designed in Chapter 3, and in fact there are some database-as-a-service products you access through the Internet.[1] However, the most

[1] Firebase (`https://www.firebase.com/`) and Orchestrate (`https://orchestrate.io/`) are two examples.

Table 5.1: cities

id#	city	state	population
1	Los Angeles	CA	3,880,000
2	Las Vegas	NV	603,488
3	Phoenix	AZ	1,510,000
4	Tucson	AZ	526,116
5	Santa Fe	NM	69,976
6	Salt Lake City	UT	191,180
7	San Antonio	TX	1,410,000

well-known API for accessing relational databases is the *SQL* query language.

Consider Table 5.1, a list of cities. Perhaps they are where our customers live. If we wanted to know the names of the cities in Arizona, in order from largest to smallest, we could write a query like:

```
SELECT city, population
FROM cities
WHERE state='AZ'
ORDER BY population DESCENDING
```

As the user of a relational database, you will have to learn about the relational modeling, and the basics of SQL, but you will not need to know how the database will retrieve the data from disk, or what algorithms it uses to sort and filter the data to produce the result you requested. In addition to `SELECT`, there are SQL queries for creating, updating, and deleting data. Along with "reading", these are called the *CRUD operations* and almost any database needs to support them. (More on SQL in Chapter 6.)

5.3 Databases in Applications

Typically we are not writing ad hoc SQL queries directly through the database, but rather using software *applications*. A data-driven application (or "app", if you really must), is a piece of software that retrieves or processes data from a database for some purpose of the user. These applications serve as "front ends" and depend on the database as a "back end" technology. Therefore, the database must run as a server—always on and waiting for requests to respond to.

There are a number of different architectures possible for data-driven applications. The simplest is when a database application is implemented on a single computer for a single user, a personal database such as an address book or contacts list. Although such a database can be useful to the individual, it is generally not a good idea that critical business data resources be stored in this way. Personal databases, just like personal spreadsheets, are challenging to scale and to coordinate.[2]

More sophisticated applications are built with *client-server architectures*. A *client* in this context is a piece of software or hardware that makes requests to a server over a network. A *server* is a centralized piece of software or hardware that waits for, and responds to, these requests. Client-server architectures can be categorized as "thin client" or "fat client", or "two-tier", "three-tier", and "n-tier". Enterprises these days are also bringing within their walls some of the data architecture concepts that emerged on the Internet, such as RESTfulness, as discussed in Chapter 3.

TODO: Diagram of fat-client, 2-tier application

Figure # depicts an example of a two-tier, fat client architecture. In the example, the database server supports what is probably a small workgroup of knowledgeable users. The users are running sophisticated software that accesses the database through the network. For example, they might be analysts using Excel, Tableau, or data mining software on their desktop computers to visualize or mine the data stored in the database. The database server is in charge of the *storage logic* of how data is persisted to disk, while most of the rest of the processing happens on client computers. An architecture like this allows expert users to select or create their own tools, but does not scale up well to thousands or millions of users. Only trusted users, who know enough not to accidentally corrupt the data, should be given this kind of direct access.

TODO: Diagram of thin-client, 3-tier application

Another very typical pattern for large scale applications is a thin-client, 3-tier architecture such as the one diagrammed in Figure #. This is a simple architecture that underlies most websites, mobile applications, and other software programs intended for a large user base. In a three-tier architecture, users access the data through an intermediary *application server* and it is the software in the middle that communicates with the database. The clients in this context tend to be simpler programs that handle only the *presentation logic* of the application: receiving input and displaying output. In what is undoubtedly the most common case, the application server is a web server

[2] See section 1.3 to review why.

Table 5.2: Types of logic in data-driven applications

Logic	Features	Location
Presentation Logic	Input and output; visual presentation of data	Client-side software
Processing Logic	Business rules; data processing and management	"Fat client" *or* application server such as a web server
Storage Logic	Data persistence, optimization, and retrieval	Database server

such as Apache, and the thin client is a web browser such as Chrome, Internet Explorer, or Safari.

A three-tier architecture allows for a division of labor, as the database administrators are in charge of data modeling, maintenance, and integrity, and application developers are in charge of the *processing logic* of how the application uses or changes the data. These developers access the data by embedding database API calls, such as SQL queries, into their own code. Because the data is independent of the application, other applications could be written that access the same database server—for example, a website and mobile app may exist side by side, written by different developers using different tools. With a little imagination, you can further generalize these concepts to four-tier or n-tier application architectures.

5.4 Choices in Logical Database Design

If this book were written anywhere between 1985 and 2005, give or take, there would really only be one starting point in learning data modeling: the relational model mentioned previously. Specialized analytical databases called data warehouses use *dimensional models* which are a variation on relational models and are generally implemented with the same technology—relational databases. Today, however, we are living in the midst of a "Campbrian explosion" of new types of databases and new types of data models. This necessitates some consideration of the benefits each type of data modeling offers.

Relational data modeling begins by identifying the *entity types* in the domain of the data. These are the nouns—people, organizations, things, events—that the data must describe or refer to. Employees and departments,

products and purchases, are some examples. For each entity type, a table is created, with one row for every *instance* of the entity type. Instances of data in different tables are related by the use of unique identifiers known as *keys*. For example, a row in the Orders table might indicate that the order was for product #42 (a reference to a row in the Products table) by customer #54321 (a reference to a row in the Customers table). In this way, details of the product and customer are stored only once in the database, despite participating in numerous orders.

At the time relational databases were introduced in the late 1970s, disk space was extremely expensive.[3] By eliminating unnecessary redundancy, the relational data model was an elegant solution that kept databases as small as possible. Also, by splitting the data up into its distinct entities and making explicit the relationships between them, relational databases enable us to make complex, ad hoc queries to retrieve data in interesting combinations.

Furthermore, relational data modeling helps businesses protect the integrity of their data by preventing *anomalies* or inconsistencies that can crop up in input. Because each piece of data is stored in only one place, for example, there is only one "source of truth". For example, a customer's name and address are stored in only one place, so if he changes it for one order, the updated information will be the only information kept in the database. Contrast this to something like a spreadsheet of orders, in which the same customer may appear with multiple entries of old and new information.

All in all, the relational model is a powerful and versatile way to model data. So why do we need any others?

Two broad categories of reasons are performance at scale, and ease of use. As we will see in Chapter 7, analytics applications such as business intelligence systems have unique needs. These systems are intended to allow people who are not experts (in database technology) to summarize, slice, and dice large amounts of data. Relational databases at the enterprise level can have dozens or even hundreds of tables; therefore, writing ad hoc SQL to analyze the data can be extremely complex, beyond the capabilities of a casual user. Also, the joining of numerous tables into a query can slow down the database's performance. In analytics applications, moreover, the data is typically accessed in a read-only manner: analysts are not making changes to the data, so there's little risk of the kind of anomalies that the relational model serves to prevent.

[3] According to Matt Komorowski, a gigabyte of disk space cost over $100,000 in 1980 and was less than 10 cents in 2010. http://www.mkomo.com/cost-per-gigabyte

For analytical data warehouses, therefore, the *dimensional modeling* popularized by Ralph Kimball is recommended. In this way of structuring data, each model features one large "fact" table and a number of smaller "dimension" tables. This makes it simple to create queries of the form *"FACT by DIMENSION"*: sales by quarter, shipments by zip code, returns by product type, etc. The goals of dimensional modeling are fast query performance, and simple query structure for casual database users. More on this in Chapter 7.

Another driver behind the move away from relational databases is to simplify application development for the Internet. One of the complexities of using a relational model is that intuitive "objects" or aggregates of data that we may work with in a software program, when we want to persist them to disk, must be split up into a number of rows in different tables. An *order*, for example, probably refers to rows of the customers table, products table, order lines table, and others. When we want to enable complex queries on the data, this may be worth it, but in many cases developers would find it simpler to just store the entire aggregate in one place. This is known as the *impedance mismatch* problem: the way data is modeled in computer memory when our software is running does not match the way it must be modeled when we store it to disk.

As a result, in recent years we have seen a rapid growth of *aggregate-oriented* databases such as MongoDB. In aggregate-oriented databases, an entire complex data structure is stored with a unique identifier (a key) and can be retrieved all at once with a call to that key. Some are *document stores* which understand the data as JSON-like or XML-like documents and can process complex queries on them; others are *key-value stores* that are indifferent to the data format and simply store and retrieve the data in any form the user wants.[4] In either case, the data is modeled in whatever way the user wants, without the formal restrictions of the relational model. A further advantage for application developers is that the model can change or expand at any time, so less up-front planning is needed.

It turns out that aggregate-oriented data models are also very helpful when we're building distributed systems. With a relational database, you run the risk that the rows of data needed for a particular query may be stored in separate disks all over your computer cluster. With a document store or key-value store, each aggregate is self-contained—the data that will

[4]Key-value stores and document stores have often been treated as distinct types, but Martin Fowler (see Recommended Viewing) argues that they are two extremes of one continuum, and the term "aggregate-oriented" is a good label for this category.

Table 5.3: Types of Data Models

Modeling Approach	**Brief Definition**
Relational Model	Numerous tables corresponding to atomic, non-redundant entity types, with relationships strictly defined by keys. Complex queries may join many tables.
Dimensional Modeling	Each model (schema) has one large "fact" and several small "dimension" tables. Queries allow summarizing the fact *by* the dimensions.
Aggregate-oriented	Arbitrary data structures are stored without breaking them up, addressed with unique keys. Data models may be document-oriented, understanding the data structures as JSON-like structures, or simple key-value stores.
Graph-oriented	Data modeled as a network with nodes and edges. No tables.

be retrieved together is all stored in the same place.

Of course, this does not apply when one wants to conduct a query that cuts across aggregates. To give an example: if you create a document store for a job search website containing numerous user profiles, it would be easy to query a user and retrieve all of the jobs listed on his CV. If you wanted to query a particular company and find all of the people who had worked there, however, such a query would have to scan all of the profiles (on all of the disks in the cluster!) costing a great deal of time. One of the strengths of the relational model is its ability to make these kinds of ad hoc queries.

In addition to relational, dimensional, and aggregate-oriented data models, a few others have gained traction recently. *Graph databases*, which model data as networks of nodes and edges (mathematical graphs), have the potential to simplify queries on social network type data. Message queues, columnar databases, and other models have found applications. In general, the rapid rise of new types of data models is referred to as the *NoSQL* movement. NoSQL itself stands for "not only SQL" and it is an umbrella term that doesn't have a strict definition. Martin Fowler's video on NoSQL, and the book *Seven Databases in Seven Weeks*, are both excellent references. See the end of this chapter.

Table 5.4: Strengths of Data Models

Modeling Approach	Strengths	Where to Use
Relational Model	Data integrity; complex ad hoc queries possible; minimal disk usage	Operational information systems where data is sensitive or critical
Dimensional Modeling	Intuitive for non-expert database users; fast read-only performance	Analytical databases for business intelligence
Aggregate-oriented	Data models in applications match data models on disk; data queried together is stored together; schemas can change	Web and mobile applications, particularly where natural aggregates (articles, profiles, shopping carts) are present
Graph-oriented	May be naturally well-suited to queries that span social networks	Experimental software for social networks and social network analytics

5.5 Tutorial: A MongoDB Backend

TODO: Demonstrate MongoDB on the local computer. It's a neat way to store, retrieve, and query data in JSON format.

TODO: Hook our Flask app (still local) to the Mongo database. Use the URL structure to generate queries. How do we get this into version control?

TODO: Add a new page to the app for "insert" queries. Now we have a database-driven web app. Maybe throw a little Bootstrap on top of it.

TODO: Move our project up to the cloud. Use a script to load a boatload of data into our cloud-scale Mongo database. Show how to shard it.

Recommended Viewing

- "Modern Databases" by Eric Redmond, co-author of "Seven Databases in Seven Weeks". http://youtu.be/G7-OGEYCMxQ
- The "Seven Databases Song". http://youtu.be/bSAc56YCOaE
- "Introduction to NoSQL" by Martin Fowler of Thoughtworks.

http://youtu.be/qI_g07C_Q5I

- "MongoDB Schema Design: How to Think Non-Relational" by Jared Rosoff of 10gen. http://youtu.be/PIWVFUtBV1Q

Recommended Reading

- Hoffer, J., Topi, H., & Ramesh, V. (2014). Essentials of Database Management. Pearson.
- Kim, Jeeyoung. (2014). "Why I Love Databases". `https://medium.com/@jeeyoungk/why-i-love-databases-1d4cc433685f`
- Redmond, E., & Wilson, J. (2012). Seven Databases in Seven Weeks. The Pragmatic Bookshelf.

Chapter 6

Relational Databases

Preview

Focus on simple versus complex queries, and what sort of stuff a relational database enables. Describe the types of queries that aren't easy with document stores. Talk about how to model a database with an ER diagram, create it with SQL, and query it with SQL.

6.1 A Well-formed Relation

TODO: Talk about E. F. Codd's paper, relational algebra, and the elegance of the relational data model. Talk about some of the tricky modeling cases: many-to-many, and so on. A bit about normalization. Relational databases enforce a schema, so up-front thought is important, and refactoring must be done carefully. Perhaps this is a drawback or perhaps it is an advantage.

6.2 SQL

TODO: All about the SQL language: DDL, DML, DCL. SQL describes what you want, it's descriptive rather than imperative. JOINS are the magic that allows very complex queries to be written very simply. Subqueries and other tools are also powerful.

6.3 ACIDity

TODO: How relational databases rule in terms of data quality, by normalization, and integrity and consistency, via transactions. Why you would want this for critical data like bank accounts. Why it seems best for analytics (b/c arbitrary queries are possible). Tease them that in the next chapter we'll explain about the need for a different model for data warehouses.

6.4 Tutorial: Powerful Queries with SQL

TODO: Demonstrate the limits of our MongoDB example by showing some queries that get difficult.

TODO: Design a relational database ER diagram that would enable the type of queries we want to do.

TODO: Do the "shredding" either of the original source, or straight from MongoDB, to put the data into relational tables. (Postgres?)

TODO: Do a number of queries in the interactive shell to show the power of the relational model for increasingly complex queries. Maybe also show some transactions.

TODO: Maybe: integrate the relational database into our web app, too.

Recommended Viewing

- "SQL vs NoSQL: Battle of the Backends" by Ken Ashcraft and Alfred Fuller of Google. http://youtu.be/rRoy6I4gKWU

Recommended Reading

- Ambler, S. & Sadalage, P. (2006). Refactoring Databases: Evolutionary Database Design. Addison-Wesley.
- Greenspun, P. (Accessed November 2014). SQL for Web Nerds. http://philip.greenspun.com/sql/.
- Hoffer, J. A., Topi, H., & Ramesh, V. (2014). Essentials of Database Management. Pearson.
- Kline, K., Hunt, B., & Kline, D. (2009). SQL in a Nutshell. O'Reilly.

Chapter 7

Analytical Databases

Preview

Again bring up the physical components of computing and data storage systems. Talk about performance issues with relational databases, why we normalize, and why we denormalize. Discuss indexes. Show how transactional systems are different from analytical systems at scale. This leads into the design of analytical databases—dimensional modeling—based on their use cases. Star schemas, three types of fact tables, grain, slowly changing dimensions.

7.1 OLTP and OLAP

TODO: Talk about how JOINs work internally—Cartesian products. Talk about table scans vs indexes and seeks. Day-to-day operations have very different workloads than analyses in terms of queries and joins, rows and columns. Remind them about transactions, and why it wouldn't be good to run huge queries that lock up your operational systems. Why do we need separate data warehouses.

7.2 Dimensional Modeling

TODO: Design considerations for databases and data marts—fast processing and ease of use. This means fewer joins, less use of codes, more redundancy because storage is cheap (and there's no updating so less risk of anomalies). Introduce the star schema. Grain. Three types of fact tables. Slowly changing dimensions.

7.3 Architecture for Data Warehouses

TODO: Kimball vs Inmon architectures—engineers vs business people? Jurney's approach: keep the source systems, and code to derive the data. Difficulty of doing ETL. Need for governance if you're going to do conformed dimensions. Enterprise data warehouse bus matrix approach. Agility.

7.4 Business Intelligence

TODO: Following on the Agile idea—what's the business value of all this? What do users want? Talk about fitting it to business processes and value chains. Stress that we are creating self-service data apps. Show the many ways they may consume data, including slice-and-dice, drill-down, and dashboards.

7.5 Tutorial: Designing a Data Mart

TODO: Show how to time a query. Run a big query, add an index, run it again and show how much better it went. Show how the query plan changed.

TODO: Design a star schema. First implement it with a VIEW, then show how to load it into another table. Show how much simpler the queries have become.

TODO: Now do a "periodic snapshot" star schema. It's little more complicated to do, and has a different grain, but look at the new queries it enables.

TODO: Use something like LOOKER to create dashboard components, then crystallize the outcomes as SQL. As a fallback, use Tableau or Excel.

Recommended Viewing

- TODO: Guthy-Renker case study
- TODO: Maybe some Kimball v Inmon debate? Or Imhoff, Devlin, somebody else big in that field.

Recommended Reading

- TODO: Definitely something about the enterprise bus matrix and conformed dimensions.

Chapter 8

Analytics Beyond Databases

Preview

What if our queries need to span billions of pieces of data? For example, clickstream data from web logs. Or the entire Wikipedia corpus. Or the entire Web. Can't fit it in one node, and the network would become the bottleneck if you tried to process it in one node. Discuss the Hadoop architecture and MapReduce. Show how well it works with the cloud (e.g. Amazon S3) and fits into the data processing pipeline.

TODO: Create the section headings and write the chapter.

8.1 Tutorial: Hadoop and Hive

TODO: Set up a Hadoop cluster and do a simple M/R job like word count.

TODO: Do something a little more complex and useful, maybe using For example, Python and mrjobs.

TODO: Show how Hive or one of its competitors can be used to transform SQL-like queries into MapReduce jobs.

Recommended Viewing

- "Learn MapReduce with Playing Cards" by Jesse Anderson. http://youtu.be/bcjSe0xCHbE
- TODO: More videos on Hadoop and M/R.

Recommended Reading

- Manoochehri, M. (2014). Data Just Right: Introduction to Large-Scale Data & Analytics. Addison-Wesley.
- TODO: more

Chapter 9

Data Streams

Preview

Setting up a data pipeline with data coming in via "streams" and being transformed, either in regular batches or continuous, to automate the preparation, analysis, and delivery of data. Talk about message queues, using Redis or Amazon Kinesis or the like, and asynchronicity.

TODO: Create the section headings and write the chapter.

9.1 Tutorial: Integrating Live Data Streams

TODO: Set up a flow of data from a stream (perhaps Twitter) into storage, transforming and loading into an analytical database, into a dashboard.

Recommended Viewing

- TODO: Perhaps a version of that AWS talk about kinesis?

Recommended Reading

- TODO: Perhaps that "Event Processing" book?

Chapter 10

Closing the Business Intelligence Loop

Preview

Talk about design considerations for self-service analytics systems, and what's the data engineer's role. Contrast periodic reporting with real-time systems. Here we might talk about architectures for business intelligence, and review a business case study. The focus is on the types of questions they might ask. Close the loop by connecting source systems with dimensional databases with BI applications. Also talk about integrating the "discovery" and "productization" functions. Goal is not "reports" but rather "applications" – products that provide answers in a self service way. Jurney's idea of "value the document over the relation"; how can we do document-oriented BI?

TODO: Create section headings and complete the chapter.

10.1 Tutorial: A Self-Service B.I. Portal

TODO: Complete our application by building a "portal" which should provide access to (a) the aggregate-oriented queries we enabled in Mongo, (b) data discovery tools in Looker or Tableau looking at our data mart, and (c) pre-made dashboards based on our streaming data.

Recommended Viewing

- "Agile Analytics Applications" by Russell Jurney, author of "Agile Data Science": http://youtu.be/woZdwluR3GM

Recommended Reading

-

Epilogue

TODO: Here review the options that were introduced, what was left out, and what decisions and trade-offs were seen. Perhaps there is some neat framework for decision making that can tie the earlier chapters together.

TODO: Add an index, maybe a glossary.