Plumbing the Big Data Pipeline

Joseph Clark

Last updated: February 2, 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.

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.

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

[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!

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*.

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

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

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 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

[5] application programming interface

[6] known as the "CRUD" operations

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

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.

1.4 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.

Tutorial: Data Munging 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.

Installing 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.

To follow the tutorials in this book, you need to be able to use the command line interface for your computer. On a Mac or Linux computer this is called "Terminal". On a Windows PC you can type "cmd" in the Start menu to access a command prompt.[8] At your command prompt (represented

[8] In class, I will tend to use a Unix-style command line interface on a Windows computer. I recommend Git Bash, which is a free accessory when you install the version control software Git from http://git-scm.com.

here by the `$`), type:

```
$ python
```

This should tell you what version of Python you have installed, and then begin the Python interpreter.[9] Here the prompt is different (probably `>>>`) and you can enter lines of Python code rather than operating system commands. Try the following:

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

TODO: The single quotes should not be "curly" quote marks. I will fix.

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.

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.[10] 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++, TextWrangler, and Sublime 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:

TODO: Display this as a code file with line numbers and no indentation.

[9] If it does not, then you will need to set your operating system's `PATH` variable, so that it knows where Python is stored. Search the Internet, or ask for help.

[10] This does *not* include word processors like Microsoft Word.

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

Save the file as `hello.py`. The ".py" extension to the filename tells Python and other programs that this is a Python program. To run the 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 need help to set your `PATH` or solve some other problems. In some cases, your system may be set up so that you can simply double-click on `hello.py` from the GUI instead of using the command line. Beware that without the final line of code above, this may run the program but close the window so quickly you cannot see it run.

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`:

TODO: Insert writefile.py code. Currently on GitHub

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()`.

TODO: Insert dataentry.py code. Currently on GitHub

Were you able to follow everything that was going on in the code? Python is a good language for learning to program, because it is closer than others to plain English. 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.

TODO: Insert readdata.py code. Currently on GitHub

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.

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

TODO: Discuss best practices for building a web service to make your data available; good URL structures and so on. Maybe talk also about how to document your API with a README file.

Tutorial: A RESTful Web Service

TODO: Introduce Flask as a minimal web server. Create a script that provides some static data from a pre-made JSON file. Set up a URL structure so that we have created a web service that makes sense.

TODO: Demo some software that can consume the data. Maybe Tableau or Excel can do this?

TODO: Re-program the back-end somehow, perhaps by including our previous data transform code (so code is stored as XML and transformed by our script into JSON). This shows that an API enables us to change the "back end" without disrupting the interface to users.

Recommended Viewing

- TODO: Some video on web services and REST architecture or SOA?
- TODO: Some video 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
- Grinberg, M. (2014). Flask Web Development. O'Reilly.

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.

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.

Tutorial: Taking our Web Service to the Cloud

TODO: Set up cloud accounts, perhaps AWS or Azure.

TODO: Create a computing node/instance/dyno/whatever to run our web service. Store the file (to be transformed) in bulk storage (S3). Show how to scale it out to multiple servers and then scale back to just one.

TODO: Probably a good time to introduce git/github for version control and automatic pushing to the cloud. Emphasize that it's best to use the same tools at scale that you use on the localhost.

TODO: Show how to use built-in, and third-party, analytics features to monitor our "operations".

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/Azure/Heroku or whatever we use for the cloud.

Chapter 5

A Multitude of Databases

Preview

Discuss the ways of structuring data in a database. Primarily focus on comparing RDBMSs (e.g. Postgres) with document stores (e.g. Mongo). Discuss metadata and queries. Graph databases and others (e.g. message queues) may be mentioned briefly. This chapter is primarily about logical data structure and doesn't talk about physical optimization.

5.1 How a Database Works

TODO: A little bit about storage and retrieval. Transactions. Physical arrangement of data on disk. How databases provide an abstraction–the logical/conceptual data model–so that users don't need to know the nitty-gritty of where and how data is stored on disk. Queries, indexes, query optimizers.

5.2 Data Models

TODO: What are data models? Reminding them of the difference between CSV and JSON/XML—the "flatness" of the former and "object orientation" of the latter—explain how the relational model got adopted. It allowed "flatness" while minimizing redundancy and maximizing integrity. Data stored in tables. This is modeled with an E-R diagram.

TODO: Impedance mismatch problem. Relational model allows for very flexible querying of data, but is very complex. JOINs are expensive. In the Internet era, for many jobs, it would be better to keep the structure simple

and the data aggregated together. Hence aggregate-oriented (KV and DS) databases.

TODO: A teaser that there are other models out there, like graph databases, for later study.

5.3 Databases in the Application

TODO: Databases usually run in a client-server mode. Generally accessible by interactive shell or by GUI. They have APIs so programs can access them. Generally they can process scripts. Scripts may also be a good way to do version control on them. Processing logic may be stored in the database (stored procedures, triggers) or in the application code. Pros and cons of the two approaches.

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

- 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.

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.

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.

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.

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.

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.