# Plumbing the Big Data Pipeline

Joseph Clark

Last updated: November 12, 2014

# Contents

**Preface** **v**

**1 Atoms, Bytes, and Databases** **1**

1.1 Atoms . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Bytes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Databases . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.4 Data Engineering . . . . . . . . . . . . . . . . . . . . . . 7
Tutorial: Data Munging with Python . . . . . . . . . . . . . . 8

**2 Structures for Data Interchange** **9**

Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . 9

**3 Opening Your Data to the World** **11**

Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . 11

**4 Data at Internet Scale** **13**

Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . 13

**5 Aggregate-Oriented Databases** **15**

Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . 15

**6 Relational Databases** **17**

Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . 17

**7 Analytical Databases** **19**

Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . 19

**8 Analytics Beyond Databases** **21**

Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . 21

**9 Data Streams** **23**
Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**10 Closing the Business Intelligence Loop** **25**
Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

**11 TBD: Beyond Hadoop** **27**
Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**12 TBD: Columnar Databases for Big Queries** **29**
Tutorial: Tutorial . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**Epilogue** **31**

# 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.

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.

## 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 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,

[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.

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 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

[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!

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.

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,

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

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 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

[5]application programming interface

[6]known as the "CRUD" operations

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

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.

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

# Chapter 2

# Structures for Data Interchange

## Preview

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?

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

-

# Chapter 3

# Opening Your Data to the World

## Preview

Talk about how to serve data up in the form of a RESTful web service API. This requires a discussion of HTTP and the basic concepts of a web server, as well as good API design and how to make it self-documenting.

Maybe use Flask as a framework to serve data up as JSON, XML, or CSV. Show how you can change the back-end without disrupting users of the API. Maybe we also want a program to consume the data, for example, Excel or Tableau.

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

-

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. Discuss the problems of networks and what they need to know about REST.

Set up blob storage on the cloud, perhaps with Amazon S3.

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

-

# Chapter 5

# Aggregate-Oriented 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.

Build a little Mongo database backend for the web service we created in the previous chapter. Enable some new types of queries based on Mongo's capabilities.

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

-

# 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.

Develop a tool that "shreds" some XML or JSON data into relational tables. Perhaps this is where Pig comes in?

Write some queries of increasing complexity, demonstrating SQL's features. Refer students to other texts for more on SQL.

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

•

# 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.

Create a kind of ETL process that pulls data from a 3NF database and sticks it into a dimensional model.

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

-

# 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.

Spin up a cluster on the cloud and do a simple MapReduce query (word count?) on a gigantic dataset.

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

-

# 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.

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

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

-

# 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.

Develop a dashboard using Tableau or some other software (Pentaho?). Take stock of how we got here from there.

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

-

# Chapter 11

# TBD: Beyond Hadoop

## Preview

Examine some variations on the Hadoop idea, such as Spark and Storm and Hive. How can we produce more sophisticated MapReduce jobs but keep them understandable and maintainable. For example, Python and `mrjobs`.

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

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

•

# Chapter 12

# TBD: Columnar Databases for Big Queries

## Preview

Discuss columnar databases as solutions for big, big data warehousing type systems. They have the power to crunch lots of rows but are structured for convenient self-service.

Use HBase or google BigQuery with some gigantic dataset and throw a dashboard (Tableau?) on top of it.

## Tutorial: Tutorial

Tutorial goes here.

## Recommended Reading

-

# Epilogue

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.