**October 27 Lecture Notes**

**Introducing the big data context**

In the first half of the course, you learned the canonical design principles for an analytical data model. Given a report the business wants, or a new analysis you're thinking about, you should have a good idea of how the data needs to be structured in order to support the required views of the data.

* Review solutions to the 3 dimensional modeling problems on the midterm

For the rest of the course, we will be taking a look at the context within which an analytical system is built. Data must come into the system from somewhere, and business users need to get data out of the system somehow. As "data engineers", you will be the ones who build and manage the big data pipelines at your work.

**Two pipelines**

Speaking with people in the industry has confirmed what my own research has suggested, that there are two types of data flows in organizations. The one I called “ad hoc analysis” is often called by others exploration or **discovery**. In that context, data scientists need to be able to develop analytical models by trial-and-error, using all kinds of data, and do not wish to be limited in their choices of data or techniques. To serve this need, vendors are offering expensive in-memory analytics “appliances” to enable the use of exploratory software (like Tableau) on huge data sets. These appliances (e.g. Teradata Aster, Oracle Exadata, etc.) generally would not be plugged into operational systems that run the business.

The second pipeline is the **production** or “productization” pipeline that describes how data flows through the organization as a matter of business. In this context, we need to deal with lots of incoming data, find ways to store and process and use it, and support all users (business, customers, and the public) who need access to it. Here is where we need to deal with the technical problems of big data and employ automation whenever possible.

**Scale up, or scale out?**

The Internet has significantly increased the amount and the velocity of data flowing into individual businesses. Some businesses, such as telecoms and banks, may have had a lot of data even before the 1990s, but they didn’t need to process it and provide answers to users in sub-second time frames the way that eBay or Google must do. While a small website like your personal blog may be able to run on a personal computer with a simple relational database and web server, major web businesses quickly outpace a single computer’s ability to deal with (1) incoming data and (2) fulfilling requests for output.

So how can you scale a data-intensive application to meet these demands?

The first approach is scaling up, aka **vertical scaling**. When I was a kid in the 1980s, you regularly heard on the news about new “supercomputers”, particularly from a company called Cray. There was a technological race to “scale up” to faster processors, more memory, more storage. This vertical scaling enabled businesses to process more data, faster. However, you rarely hear about supercomputers today. Why not?

Three facts are pertinent:

1. The latest and greatest computer technology is expensive.
2. There are diminishing returns to investment in bigger computers (i.e., there are some degrees of performance you can’t attain no matter what you spend).
3. Mid-range computer technology is a commodity and continually gives you more value per dollar.

Since the 1990s at least, there has been an emphasis to turn away from individual supercomputers and move toward **clusters of commodity computer hardware** for more storage and faster processing. In theory, a business can build as big of a cluster as it needs and, if demand increases, simply add more computers to the cluster. This is now known as **horizontal scaling**, or “scaling out”, and it has become the dominant paradigm in computing for business. Unlike vertical scaling, horizontal scaling does not suffer from diminishing returns and, theoretically, has no limit.

When my kids grow up, they will remember hearing in the news about famous computer clusters, like IBM’s Watson, instead of famous supercomputers.

**Distributed computing and databases**

So, we’re in the era of distributed computing. How does this change the game for databases?

First consider an application with lots of input and output. EBay is a good example – lots of bids are being placed, and lots of people are checking the bids. It has lots of INSERT/UPDATE queries hitting its database and lots of SELECT queries as well.

If the whole thing was running on a single server with a single database, there’s no problem. Relational databases like PostgreSQL comply with a set of standards called **ACID**. This includes:

* **A**tomicity
* **C**onsistency
* **I**solation
* **D**urability

Atomicity and consistency refer to the database’s ability to prevent half-baked transactions and make sure the data doesn’t become anomalous. Imagine a transaction where somebody makes a payment and then an order is placed for him. If the payment query succeeded and just then, a power surge caused the order placement query to fail, the customer would be pretty unhappy. By treating the two queries as one “atomic” unit, the DBMS would roll back the first part if the second part failed. Isolation means that the DBMS doesn’t allow transactions to overlap each other and cause conflicts, and Durability implies that the database doesn’t lose data if the power fails.

Now imagine that eBay has its database split up among several computers in a cluster. The database could be **partitioned** (split up into small pieces, one on each server) or **replicated** (the same data exists in multiple places) or both. A query comes in to server #1 place a bid on an item (INSERT). A query comes in to server #7 to check what’s the current bid on that item. What do you want server #7 to do?

* Respond immediately with the highest bid that it knows about?
* Check with all the other servers first to find out if they have more up to date information?

The second option would help achieve **consistency**, but would take much more time. The slowest thing in a computer cluster is the network connections between computers, and this solution would create lots of network traffic and delay each server while it was handling the request.

Many web businesses have figured out that absolute consistency is not always needed, especially not for reads (SELECTs). If you visit Twitter, for example, you probably want a result within a very short amount of time, maybe ¼ of a second. You prioritize **availability** (specifically read availability) over consistency. If someone posted something 30 seconds ago and you don’t see it yet, would you care? Would you ever know?

The trade-off that Twitter probably makes is to enable all of the servers in its cluster to answer SELECT queries, without checking with the others every single time. Writes (INSERTs) are propagated from one server to another by some mechanism, but it may have a delay. This is called **eventual consistency**.

An alternative to ACID is emerging, called BASE. It includes:

* Basically available
* Soft state
* Eventually consistent

In effect, it is a statement that the orthodoxy of ACID may be traded off. Some applications (e.g. ATMs) will prioritize consistency over availability. Others (e.g. Twitter) will prioritize availability. Still others (e.g. e-commerce) must make some tough decisions about how and when to prioritize one or another.

The trade-off is summarized in the **CAP theorem** proposed by Dr. David Brewer: *It is impossible for a distributed computer system to simultaneously guarantee: consistency, availability, and partition tolerance.*

**Principles for data engineering**

What I teach over the next few weeks will be guided by the following principles:

1. Build applications, not infrastructure
2. Build systems that can communicate data via the Internet
3. Build solutions that can scale horizontally

How can we build applications without building infrastructure? Didn’t I just tell you all about the need for computer clusters? The answer is that computing infrastructure is now available as a service, so you can “pay by the drink” instead of buying and setting up your own machines. **Infrastructure as a service (IaaS)** is what we often mean when we refer to **the cloud**. Another, closely related, type of cloud service is **platform as a service (PaaS),** in which you get something with a little more configuration such as an operating system and database management system set up.

The benefit of IaaS and PaaS is that you can order up a certain amount of storage, a certain number of processors, a certain quantity of RAM, and the cloud provider (such as Amazon or Microsoft) will do the preparation for you. You generally pay by the gigabyte per hour for storage, or number of processors per hour for computation. This means that you can “scale out” at peak times, and scale back when you don’t need as much infrastructure.

You need to know how to use these services to set up a database or data-driven application “in the cloud” without becoming a computer hardware expert. That will be the first focus in this part of the course.

Your applications need to communicate with other applications, as part of a larger project I refer to as building a data pipeline. Because our systems are not going to be sitting on one big supercomputer, they need to be able to communicate. Because we’re going to leverage tools that already exist, such as databases and business intelligence tools, they need to communicate via standard protocols, instead of ones we design ourselves. The web, and specifically the **HTTP protocol**, are the standards for communication between applications. The second focus of this part of the class is going to be on designing these data flows. We will learn how to transform data into optimal formats for different kinds of purposes, and share it via a “web services” model with our users.