LECTURE TRANSCRIPTS

MIT Professional Education

Tackling the Challenges of Big Data

Contents

1 Introduction : Samuel Madden

This class involves a number of different instructors who are going to dive into the details of big data.

We start with a little bit about what big data is and give some examples and then walk through a brief outline of the course. Unless you've been hiding under rocks for the last five years or so, you probably have heard of big data before, especially thanks to our friends at companies like IBM who have relentlessly marketed the term.

One of the things that I want to do in this course is to try and help you to differentiate the hype, the marketing from the reality of big data, trying to understand what the actual fundamental challenges are that come up when you're trying to build systems that manage data and trying to understand some of the opportunities that arise with building systems to manage data, as well as to learn about some of the new exciting algorithmic techniques that we can bring to bear with the big data problems.

So one of the things that excites me the most about big data is that it really represents a democratization of information. No longer is data solely the provenance of large corporations that house data about their business practices. Suddenly, we have data about all aspects of our lives

- about transportation,
- about medicine,
- about our social media and social interactions,
- about education through courses like this one that you're taking,
- about industrial monitoring, the engines in our cars and our planes,
- about government through open government initiatives,
- about our retail activities, attracting what we do and say as we move through a store on our smartphones,
- about science, or even about ourselves through little medical devices that we carry on ourselves like Fitbits.

So with that introduction to what excites me about big data, what I thought I would do is to give a couple of examples which will, I think, frame what some of the really key challenges that arise in the space of big data are.

Example : Massachuttes General Hospital

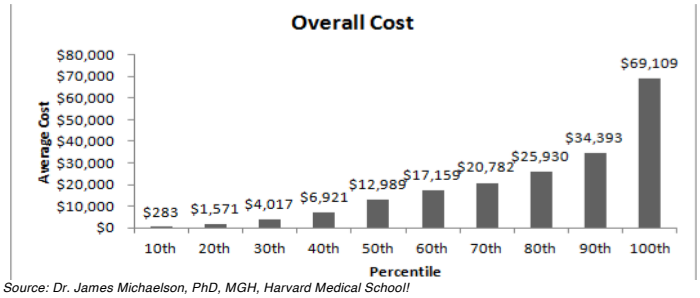
So the first example is due to my friend Jim Michaelson at the Massachusetts General Hospital. So Jim Michaelson has spent many years assembling what he claims is the world's largest cancer patient database.This database has about 173,000 patient records in it. And it includes not only information about the medical records from the Massachusetts General Hospital, the lab reports and the radiology exams and all of those kinds of things that you might expect. It's also been cross linked with a number of other databases, including something called the National Tumor Registry, which captures information about the sizes and types of tumors in these patients and relates them to other patients around the country who also have cancer.

It also includes links to the National Death Registry when these patients have died. And that turns out to be important because it allows us to understand where these people lived and some of their socioeconomic background. One of the things that Jim is trying to do with this database is to understand where patient costs come from at the Massachusetts General Hospital.

MGH Cancer Center "Super-Database"

Largest cancer database in the world (173,301 patients)!Based on national tumor registry!

Cross linked with death registry! Includes billing, reports, labs, imagery, genome SNPs!



Source: Dr. James Michaelson, PhD, MGH, Harvard Medical School!

So this graph that I've shown here represents the distribution of costs to treat lung cancer patients at MGH, the Mass General Hospital. And if you look at this graph, you see something pretty interesting. If you look at the median patient here in this 50th percentile bin, you can see that it costs about $13,000 to treat this patient. If you look at the top 10% of patients, you can see that it costs about $70,000 to treat them. So we're looking at a 5x difference in the cost to treat the median patient from the most expensive patient. You might wonder, what is it about these more expensive patients?

Is there some property of these patients that makes them cost more? And if you think about this for a minute, you can probably formulate some hypotheses of your own about what might be going on here.

For example, it might be that these patients are much sicker than the patients in the median range. They might come in with a more severe form of the disease, which costs more to treat. It turns out that according to the National Tumor Registry, if you look at the median patient versus the most expensive patient, there's really no different in the degree of sickness. You might also think that these patients are getting better with a higher probability. But it turns out that that also is not the case. These patients generally spend about the same amount of time in the hospital. And they generally die with about the same rate. Unfortunately, lung cancer is a terrible disease. And most of the patients don't make it. So the question we want to ask is, well, what are the factors that are driving costs in these lung cancer patients? And Jim's spent a lot of time correlating all the different variables against each other, trying to figure out what it is that really represents this cost differential.

And he eventually arrived at one dominant measure. If you have one of two doctors at the Massachusetts General Hospital-- call them Doctor Jones or Doctor Smith—it turns out that you have a much higher probability of being in this more expensive range, in this top 10% of patients. And if you drill down one level deeper, you find that Dr Jones and Doctor Smith are simply giving their patients more treatment. They get more x-rays. They get more chemotherapy. And yet, that additional treatment, in this particular case, is not necessarily being effective at making these patients get better with a higher probability. So Jim arrived at this conclusion through a very painful and laborious process of manual inspection of the data. One of the things that we've been trying to help them to do is to use software tools that help to automate the process of discovering these kinds of trends in his data. And that's why I think this is an exciting big data problem.

Example Flight Prediction

The second example big data problem I want to introduce you to is one that you probably are all familiar with. And this is the problem of delayed flights at the airport, predicting whether or not a given flight is going to be delayed and how delayed it is. So if you think about this problem a little bit, you can see that it's quite challenging, because flight delays depend on a number of different factors.

Weather : Your flight might be delayed because the **weather** is bad. The flight might be delayed just because it's always delayed, right? The flight might be delayed because the aircraft that you're supposed to fly out in hasn't arrived yet.

Other Concurrent Flights : The flight might be delayed because there's a lot of congestion in air traffic control and they simply can't get the flights in or out as quickly as they would like. So understanding why a flight is delayed is a really, really challenging problem.

Departure time : And there are a number of companies that have tried to solve this problem. You probably have gone on these websites and seen information about whether or not your flight is on time or not. And it's often wrong. And this problem is really hard. And I would argue that it's fundamentally a big data problem, because we have to integrate all these different data sets together that may give us conflicting view about whether a flight is likely to be delayed or not. And this is one of the really core problems, this integration of multiple data sets that arises in the big data space.

To give you one more example about why this flight problem is hard, I'm going to show you a slide from my friend, Luna Dong who works at Google now. But she did this research when she was at AT&T Labs. This is information about the same flight from three different websites. So if you go to these websites, you can see some information about whether or not the flights are delayed and when the flights are expected to take off and when they're expected to land. In this particular example, this is information about a flight that has already taken off and is expected to land in the next hour or two.

If you look at the schedule departure times, actually this flight has already taken off. But there's disagreement about when that flight actually took off from these three different data sets.

If you look at the scheduled arrival times, there's also disagreement. One of them tells us the flight will arrive at 9:40. One of them tells us it will arrive at 8:33 and one says it will arrive at 9:54.

So complicating this issue of we have many different data sets that we might get information from or many types of information that we might combine to solve this flight prediction problem, we have this additional problem, which is that different data sets actually disagree with each other. And we have to somehow this disambiguate this. So you can see that this problem of diverse data sets is one that really makes the big data space hard.

And I think it's one of the really core problems that when I talk to people in industry, they often mentioned as being something that they feel that they need help solving.

So those are my two examples, MGH and flight prediction. What I want to do now is step back a little bit and ask the general question of, what is big data? How do we know when we have a big data problem?

Too Many Bytes : Well, you might have a big data problem if you just have too much data. This is sort of the literal definition of big data, right? So this is sometimes called **Volume**.

Too high a rate : You might have a big data problem if you just have a lot of data coming at you really fast. So if you're trying to process every stock trade in the United States, that's a **Velocity** problem.

Too many sources : You've got data coming at you really fast. You might also have a problem if you have a whole lot of different data sets that need to be integrated together. Sometimes this is called the **Variety** problem.

So these three, volume, velocity, and variety, are the sort of standard definition that a lot of people use when they talk about big data.

Non-Scalable Anaysis : I would add a sort of other class of problems, that unfortunately doesn't start with a v, to this set. And that is this sort of class of problems to relate to non-scalable analysis. You have data that's just hard to process for some reason. It might be that it takes a lot of manual labor to find what you want. This is basically Jim Michaelson's problem at the Massachusetts General Hospital.

Or it might be that you have some algorithm that just takes a long time to run on the data, OK? And so you need better algorithms, or algorithms that can run on a smaller subset of the data, in order to extract the insight you want.

So part of the reason I picked the two example data sets that I picked when I introduced this was, I wanted to emphasize that big data isn't just about this sort of, literal definition I have, of I have too many bytes of data to process.

Big data is about a really big set of topics. And the hardest of these topics, actually, I would contend of these ones related to the variety of data that we have. And the fact is that often times the analysis we want to do on the data is very human-intensive, or we need to run the complex algorithms to extract the insight we want. And so we're going to spend a lot of time talking about these problems in the course.

So, where is data coming from? Where is the big data that we process coming from?

- Learning
- Science
- Transportation
- Retail
- Finance
- Entertainment
- Social
- Government
- Health/Medical

So I've already talked about medicine and flights a little bit, but I just want to give you guys a couple more examples of some of the data sets that we're going to be talking about in this course. In particular, I want to talk about two data sets. I want to talk about transportation, and I want to talk about entertainment.

But I want to drill into these in a little bit more detail. So let's start with transportation, One of the things that's happening in transportation, in particular in the automotive insurance space, is that there's a real shift towards people in the automotive insurance industry doing what's called usage-based insurance. So instead of ensuring you, based on the fact that maybe you drive a red car, or you live in a certain neighborhood, or that you're a 20-something male, your insurance company is going to start ensuring you based on the fact that you drive a certain number of miles a day.

Or that you drive in a certain way. And they're going to do that by using your cell phone, or a device that's in your car, to measure your driving habits. So this is both an exciting, and a little bit of a scary, shift. Because first of all, auto insurance is a multi-billion dollar a year industry. And this is going to be very disruptive to this business, this process of making this change.

Second of all, this disruption has the potential to really seriously violate your privacy. Because suddenly, the insurance companies are saying they want to monitor what you do. Where you drive on a daily basis. And so there's going to be a real change in the way the service is delivered, and it's going to potentially affect you in deep ways.

There are similar shifts that have happened in the entertainment space. So 10 years ago, when you bought a computer game, or when you rented a video, there wasn't a lot of information that was captured about you when you did that. Maybe, at the point of sale, some little piece of information was recorded. But now, when you watch a video online, or you play a video game, there's an incredible stream of information that's captured about you. And there are companies, like Zynga, for example, that have made many millions of dollars monitoring your everyday habits as you click on things on these games online. And again, it's been a real shift. It's made the entertainment has become much more tailored to you. It's allowed these companies to make the games better, to be more responsive to what you like. But it also has introduced some of these privacy challenges.

So I want to introduce these two topics and medicine clearly has similar issues involved in it-- because they represent shifts in industries, ways big data is changing things. And sometimes for the better, and also potentially, sometimes for the worse.

Now that we understand some of the sources of big data, what I want to do is just walk you through a little bit more detail in terms of what this course is going to cover.

So as you probably know, this course is going to be taught by a series of different lecturers, who are world's experts in big data, from MIT CSAIL.

Topics are going to include,

1. A series of case studies : Where we're going to look at big data as it applies to several different applications, including social media and transportation.
2. Ingesting and integrating data : That is, how do you get data into your big data processing system?
3. Storage and compute platforms : We're going to look at things like map reduce and relational Database systems, and how they have to change in order to support big data.
4. Presentation and visualization of information, and user interfaces for presenting big data.
5. Analytics : Set of algorithms and analytics that can be applied to efficiently process big data.
6. Security and privacy : How that influences all these different areas of big data.

So these are a few of the topics that you'll learn about in this course. We're very excited to teach you about this exciting and emerging field of big data, and we hope you enjoy the rest of the class.

2 Big Data Collection : Michael Stonebraker

2.1.1 Data Curation

This session is about data integration. And this is a really complicated topic. I will call it data curation, because what you really need to do is

- Ingest data from a data source usually not written by your team
- Validate it make sure it's correct.
- Transform the data
- Correct/Clean - Data is invariably dirty, and so you need to clean it-- i.e. correct it
- Consolidate - Need to consolidate it with any other data sources that you have on-site.
- Visualiize the information to be integrated – look at the data when it gets done.

So this whole process is called **Data curation.**

Where did this come from?

The roots of data integration, data curation go back to the data warehouse roots, which is the retail sector-- meaning people like Kmart, Walmart, those guys pioneered data warehouses in the early 1990s. And the whole idea was they wanted to get consolidated sales data into a data warehouse, and what you wanted to do was be able to let your business intelligence guys, namely your buyers make better buying decisions.

So the whole idea was to figure out that pet rocks are out, and Barbie dolls are in. So you want to send the pet rocks back to the factory, or move them up front and put them on sale, tie up the manufacturer Barbie dolls so that nobody else can get any. So that was buying decisions were the focus of early data warehouses.

The average system that got built in the '90s was 2x over budget and two times late, and it was all because of data integration issues. So these were big headaches in the '90s.

However, data warehouses were a huge success. The average data warehouse paid for itself within six months with smarter buying decisions. So then, of course, what happened was it's a small world, and essentially everybody else piled on and built data warehouses for their customer facing data, which is products, customers, sales, all that stuff.

This generated what's come to be called the **Extract, Transform, and Load** business. So ETL tools serviced the data warehouse market. So the traditional wisdom that dates from the 1990s is that if you want to construct one of these consolidated data warehouses, you send a very smart human off to think about things, and you define the schema, which is the way the data is going to look in the data warehouse. Then, you assign a programmer to go out and take a close look at each data source, and he has to understand it, figure out what the various fields are, what they mean, how it's formatted, and write the transformations from the local schema, whatever the local data looks like, to whatever you want the global data to look like in the warehouse. He has to write cleaning routines, and then you have to run this workflow ETL.

So there's a human with a few tools to help him out, and this scales to maybe 10 data sources or maybe 20, because it's very human intensive. So this is traditional ETL and where it came from. So the architecture is pretty straightforward. You have a collection of data sources.You have this ETL system, which is downstream from all your data sources and upstream from your data warehouse, and it's doing data curation, run by a human, and it's a bunch of tools.

So let's just take a little look at what some of the issues are.

Why is this stuff hard? Why was this 2x over budget?

Well, suppose you're somebody who sells stuff. So you sell widgets, and you have a European and a US subsidiary. So one of your sales records is your US subsidiary sold 100k of widgets to IBM Incorporated, and your European subsidiary sold 800k Euros of m-widgets to IBM SA. So of course the issues are you've got to translate currencies, because Euros aren't directly comparable to dollars. And then you have the thornier questions which is, is IBM SA the same thing as IBM Incorporated? Yes or no. And then, are m-widgets the same thing as widgets? So these semantic, thorny difficulties and transformations, and then, of course, this data isn't dirty, so if it was dirty, you'd have to clean it. So this stuff is just hard. And so what are the traditional ETL tools do to help you?

Well, first of all, they often give you a visual picture of the source schema and the schema you're aiming for, and they give you a line drawing tool so that you can say, here's the thing in the local schema. Here's an attribute in the local schema, map it to this other attribute in the global schema. So you basically get line drawing tools that allow you to line up the attributes, so that helps you a bunch. But then you have to do transformations, so all the ETL tools give you a scripting language. Think of it as Python, if you want. And what you do is that you write Python scripts that convert IBM SA to IBM Incorporated, if that's what you need to do. And as often as not, the ETL vendors give you a workflow so that you can define modules, line them up with boxes and arrows, and every box is some code written by a programmer.

Summary

Data Warehousing Roots

- Retail sector started integrating sales data into a data warehouse in the early 1990's
- Average system was 2X budget and 2X late
  Because of data integration headaches
- However, warehouse paid for itself within 6 months with smarter buying decisions!

Traditional Wisdom - ETL

- Human defines a global schema
- Assign a programmer to each data source to:
  - Understand it
  - Write local to global mapping (in a scripting language)
  - Write cleaning routine
  - Run the ETL
- Scales to (maybe) 25 data sources

2.1.2 Current Situation

The current situation, though, I think is very discouraging. It's very discouraging, because this technology works great for 10 or 20 or I'll even give you 30 data sources. Push me hard, twist my arm, I'll give you 50. Here's some of the problems.

Enterprises want to integrate more and more and more data sources.

The business intelligence guys have an insatiable appetite for more data sources. So let me give you an example.

2.1.2.1 Miller Beer Example :

A while ago, I got to make a sales call at Miller beer, the guys in Milwaukee. And they have a traditional data warehouse of sales of beer by brand, by zip code, by distributor, all that kind of stuff. So when I got to visit, it was in November. And the weather forecasters were predicting an El Nino event. So El Nino now we know, is a mid-equatorial Pacific upwelling of warm water. And it screws up the US weather whenever it appears. It makes it wetter than normal in the Pacific Coast and warmer than normal in New England. So this November, the weather forecasters were saying El Nino is coming. So I asked the Miller beer guys, do you expect to sell more or less beer if it's wetter? Because it's going to be wetter on the west coast. Do you expect to sell more or less beer if it's warmer? Because that's what's going to happen in New England? So is our beer sales correlated with precipitation or temperature? And the Miller guys said, boy, I'd really like to know the answer to that question. But of course, weather data wasn't in the data warehouse. So this is what causes people to want to get more and more and more data sources.

2.1.2.2 Novartis Example :

Also, there is emerging some very non-traditional, non-customer facing examples. So Novartis, which is down the street here in Cambridge, has 8,000 chemists and biologists doing wet experimentation in the lab and writing down their results in lab notebooks. So think of these as electronic lab notebooks, 8,000 of them. And every scientist gets to do his own thing. There is no global schema whatsoever. There's no ontology. There's no common vocabulary. There isn't even any common language, since some of the scientists are in Switzerland. And they write their results in German. So Novartis wants to integrate 8,000 spreadsheets. Why do they want to do this? Well, they've got various scientists who are doing experiments that either produce the same gook or start with the same gook and produce something. And they want to be able to tie these guys together so that they can collaborate. So they want to do

social networking at scale 8,000 by putting the data together. You cannot do traditional ETL at scale 8,000. It just has no chance in hell of ever working.

2.1.2.3 Goby Example :

Also, since the internet took off, there are a whole bunch of web aggregators who are scraping the web for content from a whole bunch of websites and organizing it into a common data warehouse. So we'll take a look at a company called Goby in a few minutes. They are integrating 80,000 URLs into a global schema.You cannot do ETL at scale 80,000. You can't do it at 8,000, let alone 80,000.

2.1.2.4 ETL - HeavyWeight

More over, FedEx has about 5,000 operational data stores. Probably 20 of them are in their data warehouse. What are you going to do with the other 4,980? So what are you going to do with a long tail of sources inside the enterprise? So enterprises are dying to integrate more and more data. And their ***traditional technology simply won't scale***. It is too ***human intensive***.

The second thing that's happening fairly recently is the rise of what are called data scientists. So every company I know of is trying to hire data scientists. There will be a bunch of discussion at various times in this course about data science. But they are invariably assigned to point projects. Suppose you're the brand manager. You're in charge of M&Ms for Mars candy. So you want to decide how to do your ad spend for M&Ms. So maybe you want to buy some keywords on Google. Well, what keywords do you want to buy? So you want to correlate candy eating with various key words. So suppose you have a database that does that. You then want to integrate that data source with a few others and decide how to do your marketing spend. This has nothing to do with a central data warehouse of customer and sales data. These are point projects, often in marketing. And you have a specific thing you want to do that you want to integrate three or four data sources and then do some sort of predictive model or correlation. So this has come to be called **data science**.

And to support data scientists, ETL is just way, way too heavy. You want something that's lightweight, that doesn't require a programmer. You don't want to assign a programmer to every single data scientist. So ETL, traditionally, won't scale. Or it's way too heavyweight. So this is creating a new problem space, a new sandbox in which humans need tools. So that's mostly what we're going to talk about.

The rest of this module, we're going to go over a curation example that is at scale 80,000. We'll take a look at Goby.

And then we'll look at low-end tools to support individual data scientists.

And then we'll look at high-end tools that try and support scalable data integration inside the enterprise.

So that's what's coming in the rest of this module.

2.1.3 Goby Example – The Problem

Note : This module requires navigating the website

What I want to do in this segment is tell you in a little more detail why data integration is so darn hard. So at the risk of failing completely, we're going to do a live demo of a system called goby.com, which is a web aggregator that I mentioned in the introduction that's aggregating a lot of websites. So we'll take a look at how dirty the data that they are stuck with actually is and how daunting the problem is to fix it up.

So let's dive in to goby.com, and take a look at a real world data integration problem. So we're going to take a look at goby.com, which got sold recently. So it's now been renamed Scout. So as I mentioned in the introduction, it aggregates about 80,000 URLs in the area of things to do. You know, downhill skiing, hot air ballooning, and events. Things like rock concerts, speeches by the mayor, and that kind of stuff. So in Goby you can say what are you interested in doing. So I'm a fanatic downhill skier. So I can go to things to do. And I get a category hierarchy. I go down here to outdoor recreation. And I go down here to skiing and snowboarding. And I'm interested in downhill skiing. So you're going to a category hierarchy to say what you want to do. No one wants to go downhill skiing in Boston. So we'll try it in Vermont. So what do I want to do, and where do I want to do it? So I search and I get back a aggregation from 80,000 websites. So I can look down here. So suicide six is a ski area. Killington is a ski area. So I got a whole bunch of ski areas. Everything is looking cool. And I want to focus you in on Mount Snow. So notice that here's Mount Snow. I can click on Mount Snow. And notice that it has an address of 39 Mount Snow Road in Dover, Vermont and a phone number of 464-3333. And it's called Mount Snow. And it's presumably a ski area. So that's aggregated from one of their sites. I continue down here. And whoops, I get Mount Snow again. So I click on this one just to see what's going on here. So here I get Route 100 West Dover, Vermont. So the address is completely different. Phone number is the same. And it's still called Mount Snow. But because the address is different, they don't figure out that these are the same thing. So the idea is Goby is trying to deduplicate this data. They are doing it based on the entity name and the entity address. And since the addresses are different, they

can't figure out that that's the same thing. So when you have the same ski area with two different addresses, it really is a problem. Now, presumably, one of those addresses is presumably wrong.

Which one, who knows? It's entirely possible that one of them is the maintenance department, and one of them is the snow line. Who knows?

So this is why it's really hard to deduplicate your data because it's really dirty. And it's really dirty in unpredictable ways. So Goby has a huge challenge trying to clean up their data. They have an ad hoc system.

So the whole idea is you'd like to be able to generate tools that would allow them to do a much better job. As we've seen in the Goby data looking at the Mount Snow ski area, data is really, really dirty. And sometimes it's not at all obvious how to clean it. So let me just give you one other quick example.

Suppose I have some data that shows that two restaurants are at the same address. Did one go out of business and get replaced by the other? Or is this a food court? So cleaning dirty data is really hard.

And sometimes it's not obvious how to clean it. Moreover, data transformations may well be a huge problem in addition, although the Goby data hasn't been able to illustrate that.

In Summary

- Data is dirty!!!!!
- Sometimes not clear how to clean it
  - 2 restaurants at the same address: food court or one went out of business??
- Transformations may be a big problem

2.1.4 Data Cleaning and Integration: New Ideas

In this session, we are going to look at a couple of new ideas on how to help out with data integration challenges. You can always use traditional ETL, but what we're going to do is look at ideas that you may want to check out that are not traditional ETL. So, the nice thing is that there's a whole bunch of start-ups in this space, probably a whole bunch more than I know about. There's a company called ***Paxata***. ***Trifacta*** is a commercial version of Data Wrangler, which we'll take a look soon.

Cambridge Semantics, Data Tamer is a commercial company we'll take a look at in the next segment, and there are a bunch of others. So there's a lot of activity in this space. And so if you are somebody who has to do data curation, check out some of the start-ups for ideas that may help you out.

So the focus of these start-ups, and a bunch of others, is at least to support two kinds of activities.

- Support for individual data scientists, who have point problems, as I mentioned in the introduction.
- And then support for scalable enterprise data integration.

So in the rest of this segment, we'll take a look at Data Wrangler, which is oriented toward individual data scientists, and in the next segment we'll take a look at Data Tamer, which is oriented toward enterprise support. So now we'll turn to checking out Wrangler.

Wrangler is an interactive system for data cleaning and transformation. By combining simple interactions, automatic transform suggestion, and visual transform previews, Wrangler helps analysts quickly sculpt the data set into a usable format. Imagine we are investigating property crime, and download data from the US Bureau of Justice statistics. The data is not in a format our analytic tools expect. You copy text from the CSV file containing the data. Paste the data into Wrangler, and click Wrangle to get started.

The interface now shows a table containing the data, an interactive transform history, and a transform editor. First, let's delete empty rows in the table. We select an empty row in the table. In response, Wrangler generated suggested transforms with natural language descriptions. When we mouse over the descriptions, Wrangler previews the transform's effect in the table. Red highlights indicate which rows will be deleted. We execute the suggested transform. Wrangler adds the transform to the transform history. We'll now use Wrangler's Text Selection to extract state names from

the year column. We select the text "Alaska" in row 6. Wrangler guesses we are extracting text between positions 18 and 24. It highlights matching text in each row, and previews the derived column. We can provide more examples that help Wrangler generalize our selection. We update the suggestions by selecting "Arizona" in row 12. We execute the highlighted suggestion, and Wrangler adds it to the history. We then rename the derived column "State".

The State column is sparsely populated. Missing values are indicated by the gray bar at the top of the column. When we click the gray bar, Wrangler suggests transforms for missing values. We can choose the next suggestion by pressing the Down key on the keyboard. We can execute the suggestion by pressing Enter. We now remove the rows containing the text "reported". We select the text "reported" in row 12. Wrangler suggests Extract, Cut, and Split transforms, but no Delete transforms. We can reorder Wrangler's rankings by clicking the Delete command from the Rows menu in the transform editor.

After executing the command, the data is in a relational format. We'll now create a cross tabulation of crime rates by state and year for subsequent graphing in Excel. We'll use an unfold operation to reshape the data. Unfold operations are similar to pivots in Excel. We select the year in Property Crime Rate columns. We find an Unfold transform in the suggestions. For operations that alter the layout, we preview the result with the ghosted overlay. We execute the suggested transform. We can export the transformed data to analysis tools such as Excel. Or, we can export the transformation itself. The output of Wrangler is a declarative data cleaning script. From this high level script, we can generate code for a

variety of runtime platforms. On the right, we show generated JavaScript code.

To find out more, visit vis.stanford.edu/wrangler.

So what you've seen from Data Wrangler is a visualization system that lets you look at your data, and some non-programmer tools that allow a non-programmer to massage your data, and transform it into a better representation. So the point project's support is mostly about visualization, and about non-programmer transformations. So expect a whole bunch more systems in this space.

Google Refine is another example of a system that has the same kind of flavor. Expect them at very low prices, because they are oriented toward supporting individual data scientists. And it's real clear to me that this market will be gated by the availability of qualified data scientists. So for example, I got to visit a large insurance company, who sells automobile insurance, and they are very interested in putting sensors in all their customer's cars. A market there was pioneered probably a decade ago by

Progressive. And so when you put a sensor in somebody's car, you get to record one-a-second data of exactly how they're driving, exactly where they're driving, and at what time of day they're driving.

Now presumably, if you drive in Dorchester at 4:00 in the morning, that's way more risky than driving in Lexington at 10:00 in the morning. So all of a sudden, you're going to be able to do risk analysis on way, way, way more variables. And this insurance company said, boy, we really need to move aggressively in this direction, but we are completely gated by the availability of qualified data scientists.

Meaning, people who have the necessary statistical and data management tools. So if you're an undergraduate and you want to make sure that you have a really good job when you graduate, train yourself as a data scientist.

2.1.5 Data Tamer Example

Now we'll move to the other end of the market. Supporting enterprises who want to integrate a lot of data sources. People like Novartis, who of 8,000, or Goby who has 80,000. So what I want to do is tell you about a research prototype called Data Tamer, which has been commercialized recently, and there's a start-up that is commercializing this software line. The whole idea is to do the long tail.

So when you do the long tail, all you've got to do is do it better, cheaper, faster, than whatever the company is currently doing. When you think about Novartis, you don't have to do everything. You just have to produce an ROI against whatever they're doing now. Now if you think about it for a minute, you

have no chance of doing traditional ETL. It is just too human intensive. The only way you stand chance is by applying machine learning and statistics, and then ask a human if the automatic stuff can't figure it out. So it inverts the ETL architecture. Instead of a human with some tools, it's an automatic system that asks a human if it can't figure out what to do. So we'll take a look at what Data Tamer looks like. You get a console to decide what you want to do next. So you can, if we start on the left, you can ingest the next data source. So take whatever the data source currently looks like, and stick it into a local database. And since I was responsible for this system, and I'm a database guy, it shouldn't surprise you that all state is in a database. So databases are good.

They store data.

They don't lose it.

So when you ingest a new data source, you ingest it into a database system.cIf you've ingested 20 data sources, you then may want to ingest the 21st and compare it against the 20 that you have already seen. And build up a global schema automatically. So there's a scheme integration module that does that. And if it can't figure out what to do, it asks a crowd sourcing system to get human help. If you've integrated the 21st data source, there may well be some duplicates. So you want to do entity consolidation. Like we saw in the Mount Snow example earlier in the segment. So there's a dedup module that tries to locate duplicate record to describe the same entity that need to be consolidated.

Again, if the dedup's module can't figure out what to do, it gets to ask the crowd. And then, you want a visualization transformation system along the lines of data wrangler so that a human can do the heavy lifting easily if you have to do transformations. So that's the architecture. Everything is in Postgres.

So databases, as I said, are good things to have. So what does ingest look like? So, what you do is, a data source is a collection of comma separated values. It's assumed to be in CSV format. Which is kind of universal interaction speak these days. And if it's not in CSV, then we require you to put it in CSV before we can deal with it. And we assume that your CSV is, in fact, a collection of attribute name attribute value pairs. So so-called semi-structured data. So if we had Doug, who's sitting here behind the camera, we might have Doug who lives in Cambridge. So we'd have Doug <name = Doug>, <city = Cambridge>. It's a whole bunch of stuff that looks like that. That's the only requirement. Otherwise, it's-- the data does whatever it does. And that's loaded into postgres. So you can ingest a data source. And now we have semi-structured data in a database system. So now you want to integrate that with what you've already seen. Now there are three cases that you have to distinguish. The first one is, I have no idea what the global schema is. And that's true for Novartis. They have no idea what the global schema is for 8,000 spreadsheets. On the other hand, there may be a complete global schema that somebody has already defined. Doesn't occur very often, but it does occur. And then sometimes, like in the case of Goby, there's a partial schema. Goby knows that all of their data has an entity name and has an entity address. And if their records don't have that, they figure out how to make that true. So you have to distinguish those three cases. And what Data Tamer simply does is it starts integrating data. So you give it the first two sources, tries to integrate the two of them. It won't be very smart. But as you integrate more and more data sources, it gets smarter and smarter. So the system gets better over time. And so you can say, there are some synonyms. So wages can be defined as the same thing as salary. You can have templates. Which is to say usually addresses are a city name, a state name, a zip code, a number, and the street name. And you can have authoritative tables. So for example, airport codes. Boston happens to be the airport code BOS. So if you have authoritative tables, we can take

advantage of that. So there'll be extra information. And we'll take advantage of that. The inner loop of schema integration, is I'm on the 21st data source. I get in some attribute names. Member Doug has a name, and Doug has an address. I get in names of attributes. If I've seen them before, then they might well match. So I need to check the incoming attributes against everything I've seen before. And what happens is, there's a collection of heuristic experts that try and do that. So if I have a data column in my global schema called human names, and in comes this new data source has a bunch of names in it. So I can say, well do the incoming data—does it look like what I've seen before? If the data is numeric I can run a t-test on the data. And that'll give me a score of how similar the due date is to what I have. I can treat the entire column that I know about as a document. Treat the incoming data column as a document. See how similar those two documents are if the data is text. So the inner loop is, I have a global schema that I'm building up, and in comes the iPlus First Data Source. And it's got a collection of attribute names and some values. So what I want to do is take every new name that I see and see if it matches something that I know about. So we have a collection of heuristics. And we can call them experts. The first thing you can do is, we can take the new attribute name, and do a cosine similarity on any attribute name we know about. If I have m widgets and I have widgets like we saw in the introduction they'll have a high cosine similarity. Because one is essentially one letter less than the other. And so we run cosine similarity on attribute names. In addition, we have all the data. So we have a column of data that we're building up in this global schema. And we have a new column of data. So if the data is numeric, we can write a t-test to see if the data is statistically looks like what it looks like in the global column. And again, that will give us another score for the heuristic likelihood that these are the same thing. And if the data is textual we can treat the entire column is a document. Now we have two documents. We can run cosine similarity on the two documents. You get a bunch of scores. You heuristically combine them. And the nice thing is that after modest training we've tried this out on Novartis' 8,000 spreadsheets, and we get 90% of the matching attributes correct without any human intervention. We also get more than 90% of the Goby attributes correct without any human intervention.

This cuts your human labor dramatically. Because we can get a lot of stuff done automatically. Without having to bother a human. What happens if the automatic algorithms aren't very short? Well, you've got to ask a Kraut. Now if you have biology, genomics, and chemistry names, the last thing you want to do is farm out questions to somebody like Mechanical Turk, because that means you'll be asking grandmothers in Cedar Rapids, Iowa. And they will be clueless about chemistry jargon. You are going to ask people inside the enterprise. And inside the enterprise, there are a hierarchy of experts. In fact, Novartis' uber expert is a guy named Wolfgang who is in Basel. He knows more than anybody else about genomics data. So you have a hierarchy of experts with specialization. So some people are good at genomics. Some people are good at chemistry. And so forth. So you start asking these people when you don't know the answer. And you ask every question to at least two people. And the way we do it is, in the hierarchy you ask the question to one person and at least one person who's allegedly smarter than that person. And so that means that if they get the same answer you raise the expertise of the lower guy. If they differ then you lower the expertise. So we have algorithms to adjust the expertise-ness of the experts. And of course, the problem is, is that if you want the answer to any of these crowd questions, you just ask every question to Wolfgang. And that gets you the best answer. But of course, Wolfgang has other things to do. Other than answer questions. So you need a marketplace to perform load balancing. And so the idea is questions have a cost. And you have a budget. And you try and either get the maximum certainty at a fixed budget, or you have a fixed certainty that you're aiming for.

And you want to get the cheapest cost of getting that certainty. So we're currently doing a large scale evaluation on Novartis' exact 8,000 spreadsheet database. And guess what. It works. If you're asking people randomly, they of course, random allocation of the crowd, will work in some domains. But if you know about people's expertise, you of course can do better than the crowd. So enterprise crowd sourcing is really a bunch of experts. And it works better if you take advantage of who's an expert.

So if you want to look at how the system really works, what ends up happening is all of these heuristics produce a certainty of how certain the automatic algorithms are. That the thing on the left is the same as the thing on the right. You can set your threshold to be whatever you want. If you set the threshold really high, then you'll ask a human a lot. But you'll get really, really good answers. If you set it low, it'll cost you a lot less money, but you'll get less certain answers. So you can adjust how much money you're willing to spend, versus how good the answer is. If you set the threshold to be the heuristic score of one, then what happens is everything above the line is automatically mapped. Everything below the line is sent to the crowd. So, in this case, five questions get sent to the crowd, and something like eight or nine of them get automatically mapped. So that gives you a sense for how schema integration works

automatically, as much as possible, with human intervention, when necessary. So now, if you add the 21st data source, it may have some duplicates with the 20 that you've already seen. As we saw in the couple segments ago, there were a whole bunch of mount snow records. So you want to be able to do consolidation when the entities are the same thing. So you have tables that are, the schema integration module constructs a collection of tables. So there is a global schema being built up, and is populated by

records. So what you want to do is see if you have duplicate records in any of these tables. Now Goby turns out to match only on attribute name and attribute address. But, that's of course, you can do much better by taking all values into account. So you can do weighted clustering on all the attributes that you know about. And the reason to weight things is, if you have an attribute called sex; well if two people are males, that gives you very little certainty that they're the same person. On the other hand, ski area almost always have vertical drop as one other attributes. If two ski areas have the identical vertical drop, chances are they the same ski area regardless of what they're called. So basically we solve the data clustering problem in n-space, meaning all n--all the attributes that we know about. It's an n squared calculation, where n is the number of records. If you get to scale, you don't like to do n squared calculations. That really takes a while. So as a first pass to try and knock down the complexity-- and, by the way, this works very well. It works wildly better than Goby. We also looked at some data from an insurance aggregate are called Verisk.

And they currently have a domain specific d duplication system. And we do just a bit better than their domain specific system, with a domain independent system. So automatic algorithms are getting pretty darn good, and they can be they're entirely horizontal. So what does this stuff look like? Well, we produce a collection of clusters. At the end of this n squared algorithm, we have a collection of clusters that we think are the same record. So the brown bubbles are individual records, the blue circles are clusters. And so, you can say, here's a particular entity that we think is the same thing, that we think has five records. It's a thing who's entity name is Craft Farm. They're a whole bunch of individual records from various sources. You can, in fact, select a specific record and blow it up. And you could also look at our edge to edge-- our record to record strength of clustering.

So we give you a little visualization system to check out our automatic system. And, again, if our automatic system thinks it's strong enough, and you set the threshold so that our strengths are greater, a human will never be asked. Otherwise, this will go to the crowd to be resolved. So where's Data Tamer going? Well, first of all, not all data is structured. So you want to be able to integrate text with structured data. Almost everybody wants to do that. And we're working on that.

Second thing is, if you integrate, say, an oracle database with another oracle database. Well, oracle knows about foreign key primary key relationships. So, right now, Data Tamer doesn't understand relationships, but we have to move our algorithms in that direction. Some people are really interested in hierarchical data. Which is purchase orders have line items in them. It's very hierarchical. We may extend Data Tamer in that direction. Input adapters are a big problem. Which is, right now you've got to get your data into CSV format. And we want to make that easier by having some way to

generate adapters quicker. And, of course, we're working hard on algorithms to make the automatic

algorithms better. And then, if you want to integrate Data Wrangler, or Google Refine, or some other tool; we want to be able to have user defined operations, not just the ones that we invented.

So are hard at work making Data Tamer better. And this gives you a sense for the direction you have to go if you want to use a long tail of data integration. So this is the end of the segment on enterprise data integration. In the next segment there will be some summary comments, and some

conclusions.

Data Tamer Summary

Data Tamer Goals

- Do the "long tail" – Better/cheaper/faster than the ad-hoc techniques being used currently
- By inverting the normal ETL architecture
    - Machine learning and statistics
    - Ask for human help only when automatic algorithms are unsure

Data Tamer Ingest

- Assumes (for now) a data source is a collection of records, each a collection of (attribute-name, value) pairs.
- Loaded into Postgres

Data Tamer Schema Integration

- Must be told whether there is a predefined partial or complete global schema or nothing
- Starts integrating data sources
    - Using synonyms, templates, and authoritative tables for help
    - 1st couple of sources require asking the crowd for answers
    - System gets better and better over time

- Inner loop is a collection of experts
  - T-test on the data
  - Cosine similarity on attribute names
  - Cosine similarity on the data
- Scores combined heuristically
- After modest training, get 90% of the matching attributes on Goby and Novartis automatically
- Cuts human cost dramatically

Data Tamer Crowd Sourcing

- Hierarchy of experts
- With specializations
- With algorithms to adjust the "expertness" of experts
- And a marketplace to perform load balancing • Currently doing a large scale evaluation at Novartis
  Late flash: it works!!!!

Data Tamer Entity Consolidation

- On tables defined by schema integration module
- Entity matching on all attributes, weighted by value presence and distribution
- Basically a data clustering problem
- With a first pass to try to identify "blocks" of records
  – Otherwise N ** 2 in the number of records
- Wildly better than Goby; a bit better than domain-specific Verisk module

Data Tamer Future

- Text
- Relationships
- Hierarchical data (maybe)
- Adaptors
- Better algorithms
- User-defined operations

2.1.6 Data Curation : Conclusion

So this module has talked about data curation. We went through the history of where it came from,

from the data warehouse market. We had some issues on why it was so hard. And we looked at some new ideas, both for individual data scientists, and for enterprise data integration. So I want to just give you a few concluding thoughts here.

Every enterprise that I talked to wants to integrate more and more data sources. And when you ask a CTO or a CIO what's his biggest problem, and he works for a big enterprise, chances are he's going to say data integration. This is the number one headache of most everybody.

The problem is ETL won't scale. It's too expensive to do it manually with a programmer. The people who are doing web aggregation are really into this in spades. So for instance, I was talking to the CTO of Groupon. And Groupon is trying to construct a global database of small businesses. They want to integrate 10,000 data sources. So scale thousands is getting to be routine. And it's really hard. It remains to be seen what fraction of this market can be helped by automatic tools. Data Tamer-style tools. The initial results are encouraging that that's the way you have to go. There's no chance of doing it with traditional ETL Cleaning your data is going to be a huge issue forever. Novartis's point of view is that they have a bad data cleaning problem. You've got to push it back toward your data sources. Get the originators of the data to get involved in cleaning. Because otherwise, once it gets way downstream,

it gets harder and harder and harder to do it. And the odds that you can get your data to be 100% clean is nearly zero. So you've got to be able to deal with data that is just too expensive to clean. One of the things, a long time ago, I got to visit the state of Montana. The department that issued checks for people on welfare and people with disabilities, and the administrator said, well we are up to the point where we're issuing-- where 50% of the checks we cut are correct. The other 50% are wrong. That's a little bit low for data quality. But you're going to have to live with-- there's some data that may be too expensive to clean.

Data transformations, as we saw in the data wrangler demo, are a big issue. You've got to figure out how to do them cheaper. Non-programmer tools is one way to go. Another way to go is, I can imagine a big database of transformations. Maybe it's on Google. Maybe it somewhere else in the web. But every time I hear a programmer write a transformation he always says, somebody has written this before.

Why do I have to redo it? It's just right now too hard to find. So it's and interesting search problem to search for a codes that you may be able to use. This whole issue is going to get gated by data scientists. The average application that I hear about, so you want to be able to correlate x to y. So you go out and you find the data. And the average data scientists spend 80% of his time getting, extracting, and cleaning his data. 20% of his time doing data science. So data scientists are going to have to get

very familiar with this kind of stuff. Because they are being asked to do stats, data management, and data integration. So data scientists, get familiar with this stuff. There's a lot of activity in this area. Lot of start-ups. It's a very active research area. And hold onto your seat belt. Without a doubt, better stuff will be coming in the future.

2.2 Hosted Data Platforms : Matei Zaharia

2.2.1 Cloud Computing

What is cloud computing?

Cloud computing has been in the press a lot lately, and many different definitions floating around, many services that claim to be clouds. But at a high level, what cloud computing means is computing resources that are available on demand. By computing resources, I mean a wide range of resources.

They can be just storage or computing cycles. But they can also be higher level software built on top. So for example, several providers today offer databases as a hosted service. So the provider will set up and manage a database for you and do all the administration.

And as a user, you directly see and use a database. Now the "on demand" aspect means that resources are fast to set up and fast to give away or tear down. And it means that pricing is usually pay as you go. What I mean by that is that you pay at a small granularity. If you only use your hosted database for a day, you only pay for a day. And often with these services, you pay on granularity as little as an hour.

For big data workloads, in particular, clouds can be attractive for several reasons. So the first one is that clouds provide easy access to large scale infrastructure that would otherwise be very hard to set up and operate in-house.

For example, if you need to store 100 terabytes of data, or if you need to launch 100 servers to do a computation, this can be quite a bit of setup, quite a bit of administrative overhead to do in-house. But with a cloud, it's possible to pay with your credit card and have access to these resources right away and start trying out a computation and seeing if it would be useful.

The second reason that clouds are attractive for big data is that big data workloads are often bursty. And so they benefit quite a bit from the pay-as-you-go model. What I mean by that is, when you collect a large amount of data, you're not usually just continuously doing computations on it. Instead, maybe once in a while, you have a large computation that you want done. And with the cloud model, you can acquire a bunch of computing resources for just that computation, and then give them away and only pay for the time that you use them.

So because of these properties, cloud computing has seen a major growth in the past few years, in actually almost all software domains, and certainly in big data. Let's talk about some examples of cloud services.

Our cloud services actually exist at multiple different levels.

At the lowest level, you have just raw storage or computing cycles and a variety of services that offer that today. The most well-known are Amazon S3, or simple storage service, and EC2, elastic compute cloud, which lets you store bytes and launch virtual servers to do computations. But since Amazon began these services, quite a few other providers are offering similar ones as well, including Google's Compute Engine, Windows Azure from Microsoft, and Rackspace.

At the next level up, you have hosted services that provide a higher level piece of software and just host and manage it for you. And a great example of that is Amazon's relational database servers or RDS, which has hosted versions of just database software, including, for example, MySQL or Oracle.

Now normally, managing and hosting a database isn't entirely easy. You need to make sure the database is up and running, highly available. You need to make sure that you're setting up and taking backups. And you may even want to replicate the data archives to data centers for disaster recovery in case one of the data centers goes down. With the database hosted on Amazon, Amazon manages all these properties for you. And you can go ahead and use a database. Other examples of this include Google's BigQuery, which lets you on SQL queries on Google's distributed infrastructure, and Amazon Redshift, which is a hosted analytical database similar to the analytical databases you'll see at other parts of this course.

And finally, at the highest level, you have entire hosted applications which are directly accessed by end users. So one example of that is Salesforce, which provides a variety of enterprise software such as a customer relationship management software just hosted so that business users can directly use it. Splunk is a software for analyzing log files from servers. And there's a version of Splunk that runs in the cloud so that you just point logs from your servers to it, and then Splunk will analyze them and show you interesting graphs and interesting things that are going on. And finally, Tableau is a visualization software that lets you quickly explore and slice through data. And there's a hosted version of Tableau that is managed by the Tableau company itself.

2.2.2 Cloud Benefits

2.2.2.1 Benefits for users

And let's start talking about the benefits for users.

- ***Fast Deployment :*** First benefit for users is fast deployment. Using a cloud, computations can start in minutes without a long set-up period. And this can be a great way to quickly get started with a new type of computation. Normally, setting up hardware and then setting up software on it in-house can take months. And using a cloud, it's possible to just quickly get started, try something out, and see if it works.
- ***Oursourced Management :*** The second benefit is outsourcing management. So the cloud provider handles system administration, it handles reliability and disaster recovery, and it handles security of the hosted services. And these are all problems that usually require quite a bit of commitment and resources in-house. So for an organization that doesn't want to specialize in these, it may make sense to outsource them.
- ***Lower Costs :*** The third benefit can be lower costs. There are two reasons why costs might be lower from the cloud. First of all, users of the cloud will benefit from the economies of scale of the provider. This means that the provider, by being a large purchaser of resources, can often acquire lower prices, or it can share expertise for example, of administration or security staff among many customers, and be able to offer them cheaper than a customer would be able to do by themselves. That's one reason to get lower costs. The second reason is the pay-as-you-go model. So if you're not using the resource all the time, using the cloud, you can pay for it just in the times when you use it, whereas if you had to purchase it yourself, it's sitting there idle and you have, in some sense, wasted money.
- ***Elasticity :*** And finally, the fourth benefit to users is elasticity. Using a cloud service, it's easy to acquire large amounts of infrastructure for a short period of time and then give them away, whereas in-house, you'd have to acquire probably a smaller amount to hold onto, and then do all the computations on that.

2.2.2.2 Benefits for providers

So these are the benefits for users, but the cloud model also has some benefits for providers. And this is why more and more providers are going for this model.

- ***Economies of scale*** : The first benefit for providers is economies of scale. So because the provider is servicing multiple customers, they can share the expertise and the resources they purchase across these customers, and they're able to attain lower costs because of that. So for example, a provider might hire a system administrator only once, or a security expert only once, and have them administer or secure the resources of many different customers. And so that cost gets spread out across them. Then the second reason why economies of scale matter is that the provider, just by virtue of being bigger, might be able to acquire resources at a lower price. For example, a provider who needs to buy power might build a data center right next to the power plant and get a much lower rate on that than most organizations would be able to do.
- ***Fast Deployment*** : Second reason why providers are going for the cloud model is fast deployment. Traditional software sales cycles are very long. It takes months to set up an agreement, set up a long-term deal with a customer, and then wait for them to have the hardware in-house to actually run the software. And in contrast with the cloud, the provider can get customers right away, because they can quickly sign up for the service and get started, and can immediately get revenue or feedback on the software. In the same way, when a provider builds a new feature, they just need to launch it on the existing services and it immediately reaches users, without waiting for users to buy the next version of the software.
- ***Optimization across users*** : So the final reason is optimization across users. Because a provider sees the workloads of many users at the same time, they're able to take lessons from the workloads and apply optimizations that might be harder for a traditional software vendor to do, when the traditional vendor just gives the software to the user and never looks at it. For example, Amazon, with its hosted database service, might see how many different users use databases and come up with optimizations that will make these faster or more efficient because they're able to actually see the complete workload.

2.2.2.3 Benefits for Big Data Workloads

So finally, for big data workloads in particular, clouds have three benefits that are specific to the nature of the workload.

- ***Reliable Distributed Storage :*** First, cloud provides easy access to reliable distributed storage, which can often be quite difficult or expensive to set up in-house. Often, a challenge with collecting big data sets is making sure that you can actually preserve and archive the data over time and still be able to access it later. Usually, this requires replicating it to multiple geographic sites and making sure those sites are in sync and so on. So using a cloud service, the provider is already doing this for you, and you don't need to set up these different data centers and set up this infrastructure on your own.
- ***Elasticity :*** Second benefit is elasticity. A lot of big data computations are highly parallel, and using the elasticity of the cloud, you can easily acquire a few hundred servers at once, do the computation, and then an hour later, turn the servers off. And you've only paid for that one computer and been able to get back the answer quickly.
- ***Data Sharing across tenants :*** So the final reason is data sharing across tenants. Large data sets are actually expensive to move across the internet, and can also take quite a bit of time. So using the cloud, many tenants that are sitting in the same cluster in the same data center can very quickly share data. One example of this is the public data sets provided by Amazon--which, Amazon collects a number of interesting scientific and government data sets and lets different users just access them as soon as they launch machines in Amazon's cloud. And usually, bringing these data sets in-house would take quite a bit of time for an organization running in-house.

Challenges for Big Data workload using cloud

At the same time, clouds also have several challenges, many of them specific to the big data domain.

- ***Security and Privacy :*** The first one is security and privacy. So because you are outsourcing computation to a third party, it's harder to guarantee with a cloud that the data will remain secure and that no one else will get to look at it. And this is especially problematic in applications where legislation requires you to have strict controls over who accesses the data. So these can often be a challenge for the cloud.
- ***Data Import/Export :*** The second problem is data import and export. As I mentioned before, big data sets are expensive to move across the internet. But if you're going to start collecting your large data sets and processing them in a cloud, you'll need to move data into there, and this can often be time consuming or expensive.
- ***Lock-In :*** And finally, the third risk with a cloud model is lock-in. Because you've put all your data in a provider, it's expensive to move out. And because you might be relying on services and applications that only that provider hosts, it might be hard to migrate to a different provider if you need to in the future.

So this is certainly something to take into account when considering the cloud. So in this lecture, I'm going to start with a more detailed look at cloud economics, saying, where do the cost benefits of the cloud come from, and in what cases, what kind of workloads does it make sense to use the cloud?

Next, I'm going to go into different types of cloud services, cover some of the services that are out there today, and also look at different dimensions that one might compare cloud services on to make a decision.

And finally, I'm going to go into more detail on the challenges of the cloud model, including security and lock-in, and steps one can take to address these challenges.

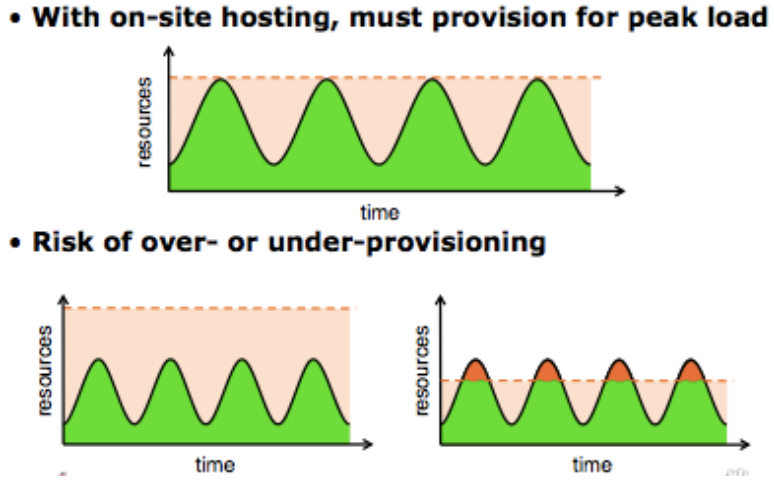
2.2.3 Cloud Economics

When does the cloud make economic sense? So I'm going to talk about three cases, or three reasons, compared to traditional on-site hosting when using a cloud may make sense.

These are ***variable utilization***, ***economies of scale***, and something I'm going to call ***cost associativity***, which is about acquiring more resources for a shorter period of time.

2.2.3.1 Variable utilization

The issue here is that with on-site hosting, you must provision resources for the peak loads that you anticipate them to have during the day even though in practice much of the time the load might be lower. For example, imagine you're purchasing a cluster of machines to do computation. You have to think about what will be the biggest computation it will have to do when the most users will be using the cluster. And that might happen, say, once a day. And during the rest of the day, parts of the cluster will be idle. In fact, most resources in practice follow a cyclical usage pattern when there's quite a bit of variation at different times in the day. Usually in the morning, there's no one in. Resources are at a low load. And then as the day goes on, they ramp up up to a peak. And then they ramp back down again as workers go away or go home.



So you end up with this cycle where you go up to a peak and you go down as naturally as users employ the resources. But with static provisioning, you have to choose a provisioning level in advance, as shown by the orange bar here, that's just at the peak you anticipate during the day. And as a result, there are these periods in between where you have the resources, but they're just sitting idle. And they are essentially wasted. You paid for them, but you're not actually using them. This variability during the day is one reason that you might have a problem.

But the second problem happens because the provisioning has to be done upfront. So there's a significant risk of over- or under-provisioning. By over-provisioning, I mean that the organization chose a resource level that's higher than what they actually need. For example, here, we chose this level. But actually the load is just down here. And as a result, there's quite a bit more wastage of resources compared to the one above. In this case, more than half of the resources are maybe

sitting idle on average.

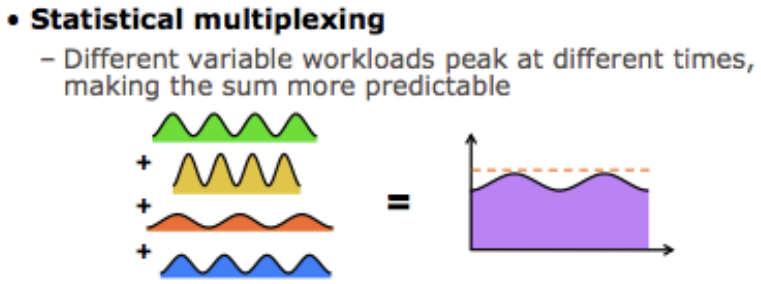
Under-provisioning can also be a problem. And under-provisioning just means that the level you chose is too low. So the issue here now is that peak times during the day when you don't have enough resources to sustain the computations. And this might mean that the computations slow down or that user applications have to wait in a queue to run. And so user experience is worse. Or if these are customer-facing applications, it might mean that the customers actually move away because they're not getting great, consistent performance. So in fact, most organizations are especially worried about this case of under-provisioning, so they'll err on the side of caution and over-provision, and then end up with this kind of wastage of resources shown on the left.

So with the cloud model, it's possible to ramp resources up and down at a much finer granularity because clouds usually charge for them at a much finer granularity. For example, the most common one today is granularity of one hour. So using our cloud, for example, imagine these servers here were servers from Amazon instead of servers that you had to purchase on-site. It's possible to ramp up the usage during the day, so add on more servers as the load goes up, and then ramp it back down when the load goes down, and overall have a much closer tracking of the resources to the actual load and much less wasted resources in these little orange spaces compared to the previous one. And with Amazon, in particular, the steps which you have up and down can be as small as one hour. So this means that even if the cloud provider offers higher hourly rates than it would take you to buy those resources in-house, it might actually be worth it because a lower percent of the resources you purchased are wasted.

So overall you might actually still be gaining something compared to buying these servers on-site. A question you might be thinking about seeing this load going up and down and the risks of over- and under-provisioning is how come the provider can do this better than individual users?

After all, Amazon, itself, or Google or Microsoft also has to buy servers. It also has to provision for a certain peak load. And it also has to deal with changes in utilization during the day. So there are two reasons why providers can actually do this better and actually pass on these benefits in this hourly pricing that I was talking about before.

Statistical Multiplexing : The first one is called statistical multiplexing. It's a fancy way of saying that if you have different workloads that are variable, and they're peaking at different times, the sum of them together might be less variable than the individual parts.



So in this little diagram here, we have four different cloud users. The top one has this green wavy pattern that we saw before. This one here maybe has a smaller time period. It's a bit spikier. This one here may be spread out over a longer time period. And maybe these are also spread out geographically, so they have peaks at different times during the day in different areas of the world. So when you add up the load from all these users, the sum of them ends up having a much lower variance.

And so the provider has a much more predictable usage pattern that they can deal with. And this means that a large provider like Amazon and Google can very easily know how many resources they'll need in the future, and even during the day, and be able to offer them efficiently. And finally, the second reason that some of these large providers can do it is because they already have large amounts of applications internally that need to use compute resources. And if cloud customers aren't using them, they can simply use them for these internal applications. So for example, both Amazon and Google needs to run large internet services. And they need to also do a bunch of lower priority batch computations on the back end, things like analyzing the data or training learning models or things like that. And so when someone isn't using the cloud resources, unlike an on-site provider who has bought them and would just have them sitting idle, these companies can actually use them for their internal workload. So they're not actually wasting them in the same sense that an on-site provider would. And this is why it's no accident that some of these companies were the first ones to start offering cloud, because they saw that their own usage pattern was bursty, and they wanted to do something with the gaps in their resource usage.

2.2.3.2 Economies of Scale

Second aspect I want to talk about when clouds can make economic sense is economies of scale. And this is essentially if the company purchasing the cloud, or the user purchasing the cloud, is much smaller than the cloud provider, then the cloud provider has some advantages in terms of the costs that they can offer. And these advantages can then be passed on to the customer.

Just as a concrete example, a small company that's trying to manage some servers might need to hire a system administrator. And say that they have a hundreds servers on-site. So if the syste administrator's overall cost of the company is $100,000 a year, this means that they're paying $1,000 per year for each server to have this person administrator. In contrast, a provider like Amazon might only hire one administrator for 10,000 servers. And they might have internal processes and just a larger number of servers that make it possible for one person to manage all of these at once. And as a result, Amazon's cost per server is only $10. On top of this issue here of static resources like the administrator is a fixed cost that you have to get at least once, there are also variable costs that can be smaller for the provider.

So because Amazon is a really large customer, they have more purchasing power. They can buy hardware like servers or a disk drives at lower cost. They can also buy electric power at a lower cost or security, for example hire one security guard for the entire data center, and so on. So these variable costs will also go down.

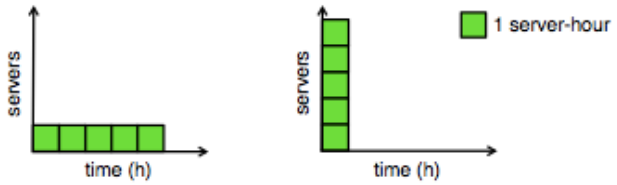
Finally though, there is a flip side. So unlike resources you purchase on-site, cloud providers also have to have a margin. So the costs that you see from them aren't going to be exactly the costs that Amazon or Google internally have. Instead, they're trying to make a profit. But the idea here is that if you're a small organization at least, Amazon's price, even with its margin, might still be lower than the price you have in-house.

2.2.3.3 Cost Associativity

And finally, the third aspect I want to talk about is cost associativity. So let's talk first what is associativity? So associativity is a property of some mathematical operations, such as multiplication. In multiplication, it just means that a times b is the same thing as b times a.

When this comes the clouds, the reason this comes in is because pricing of most clouds is per resource hour. So this means for example, if you buy 100 servers for one hour, it's the same price as having one server 100 hours. And so it's possible, using the cloud model, to translate a lot of things that would have taken a long time on a single server, into using more servers for a short period of time, and still be able to do it at the same cost.

- **For the cloud: 100 servers for 1 hour cost the same as 1 server for 100 hours**



The picture here shows a really simple example. We have five server hours of work to do. And instead of doing them all sequentially on one server, we're going to do them all in parallel, and it's going to cost the same. Let's say each server hour costs $1. So what this means is that for parallel workloads, using the cloud we can get an answer faster, because we can ramp up to a large number of servers for just a short period of time. So even though this case here with the five servers has the same cost in CPU cycles for a $1, it gives you more productivity per dollar than buying a single server would.

So that's it. So to summarize, clouds can provide the most advantage when you're benefiting from one of these three reasons that I showed before. Either the user's usage on-site is valuable during the day, and you'd like to not pay for resources when they're idle.

Or the in-house organization is small, and just the economies of scale of offloading things like security and system administration to a cloud can be worth it, can lead to a lower cost per server.

Or finally, if you have highly parallel workloads, where using the cloud you can use more parallelism get this same workload done for a similar cost, but get more productivity.

2.2.4 Types of Cloud Services

So our next topic will be types of cloud services. I'm going to talk about the different dimensions one can compare cloud services on, and also, what are some examples of services available today, and how they fit along these dimensions.

So this is an area where there's actually a fair bit of vocabulary starting to come up-- things like infrastructure as a service, or public cloud and private cloud, and so on. And I just want to highlight the three main dimensions that you can compare cloud services on and give some examples and talk a little bit about the options among each one.

- Levels of Abstractions
- Multi-Tenancy
- Access Interfaces

2.2.4.1 Levels of Abstractions

The first dimension that people compare cloud services on is levels of abstractions. And there are three different levels of abstraction that people talk about today

- Software as a service
- Platform as a service
- Infrastructure as a service.

These are fairly established industry terms. And they're also defined more specifically in the NIST definition of cloud computing, which is a report from NIST about trying to clarify some of the terms around cloud computing and let users compare these different services.

Software as a Service

So software as a service is the highest level. And it means complete user-facing applications.

Examples of this application are Splunk Storm, which is a hosted version of Splunk's log aggregation product and is just called Storm. And it lets you visualize and collect data from logs.

Tableau Online, which is an online version of the Tableau visualization software. So these are end-user applications that we expect business users or analysts or just customers to use directly, the same way that they'd use software installed on a local machine. And you just get the complete application. And you don't know anything necessarily about how it's being hosted or managed on the back end by the provider.

Platform as a service

The next level is platform as a service. Platform means that these are developer-facing services—so things like, say, a web application host or a database. And their higher level than just raw machines.

You can't just go in there and start launching programs on the machine. But they're lower level than end software. Usually, you use these as components in a bigger application. So two examples of this are Amazon's hosted database service, RDS and Amazon's hosted MapReduce. So both RDS and MapReduce offer abstractions that you might use to build an application. RDS lets you host a relational database. MapReduce let's you run computations with and against a standard API. And Amazon offers these to you. You got to put in some code that will run, say, the database queries or the MapReduce jobs. But you don't get full control over the resources. Amazon is still assigning them and figuring out how they're going to share them across users.

Infrastructure as a service

And finally, the lowest level is infrastructure as a service. And here you do get just raw computing resources, nothing special built on top. And you can kind of do whatever you want with them. So for example, most cloud providers let you just launch virtual machines. You can run anything you want on your server. And there are also a lot of cloud storage services that provide virtual disks. So you just attach this disk to a virtual machine. You can run a database on it or a file system or whatever else you want.

So these are the three levels of abstraction. And essentially, the trade-off between these different levels is the lower levels like infrastructure give you more control. But the higher level give you more management and more complete applications. So you get to choose whether you want to write the application and manage it yourself, or go with a hosted one, or use a platform component that's somewhere in between.

2.2.4.2 Multitenancy

The second dimension to compare clouds on is levels of multitenancy. And the two common terms you'll see here are **public cloud** and ***private cloud***. So remember, I said at the beginning that a cloud means any computing resources available on demand. But for the provider of the cloud, there is still room in terms of how they'll share these resources among users and how much they'll benefit from putting users together versus assigning the resources to just one customer.

So ***public clouds*** are services where the underlying computer resources are shared by multiple tenants and are available for the general public to sign up on and start launching computations on. There are lots of examples of this. Google Compute Engine, Amazon, Windows Azure all offer public clouds.

Private clouds on the other hand means that resource are allocated to a single organization which will use them for their own internal workloads. So this makes a lot of sense if you have a large organization like, say, Bank of America or Walmart, which has a lot different internal workloads and still wants to share resources between them in a cloud fashion. And private clouds can actually be hosted either on premise at the organization or off premise at a provider that just sets up and manages servers isolated for this company. And there are providers out there like Rackspace that will do both. They can either help you install a cloud on premise or host one for you in an area of the data center that's just allocated to you.

2.2.4.3 Access Interfaces

Finally, the third dimension to compare clouds on is the access interfaces that they provide. And there can be two types of interfaces-- **open** or **proprietary**.

So many cloud interfaces that you see are actually standard. They're either interfaces that you'd already use in a private data center or they're just a standard across vendors that vendors happen too all commonly use.

So for example, the simplest one is virtual machines. Virtual machines all look like an x86 processor. And that means that any program you wrote for just pretty much any machine anywhere can run on these clouds without any trouble. The same thing happens with the interfaces for block devices for storage. Operating systems see them as a block device. They don't need to know that it's running in a special environment in the cloud or with hosting standard software, specially opened software with open interfaces.

So for example, MySQL database hosting just looks like a MySQL database and Hadoop MapReduce is a standard API for doing MapReduce. Now, there are other interfaces that are also quite powerful but are proprietary specific to a particular vendor. And usually, these are interfaces for doing a new types of computation or storage where maybe there's no existing open implementation that can do it. But the provider still wants to offer this for you. So for example, Amazon's DynamoDB is a key-value store. You'll see key-value store is covered in more detail in the rest of the course. And it provides high performance distributed storage for reading and writing small objects. And it's based on Dynamo, which is Amazon's internal technology for storing the web session data. But Amazon decided to just offer an interface to this that's specific to Dynamo DB. And it means if your application uses this interface, it can only really talk to Dynamo. Google BigQuery is another example. BigQuery is a service for launching SQL computations on Google's internal data infrastructure. And it has a specific API that doesn't look necessarily like the APIs of other databases. So in choosing between these interfaces, often there's a choice between, are you using something that's specific to a provider—maybe it provides special properties that you wouldn't get someone else-- or are using something open that makes it easier to migrate across providers but may also not do the same kinds of computation that you'd really like to have?

2.2.4.4 Examples of cloud services

So finally, let's talk about some example cloud services available today and categorize them along these three dimensions. I've just included five services here, but these dimensions are useful to think about when you're looking at other services too.

Amazon elastic compute cloud

So the first one is Amazon's Elastic Compute Cloud, or EC2. And Amazon's EC2 just lets you host virtual machines, they're just running x86 code, and you can just launch any programs you want on them. So if you look at the level of abstraction for this, this is ***Infrastructure as a Service***. This is just low-level computer infrastructure You can do anything you

want with it. The hosting for Amazon EC2 is ***public***. So multiple users, users you may never know, could be acquiring virtual machines in the same data center. And the interface to it is a standard in the sense that it's just the x86 interface for running code. Although an interesting aspect here is that the API for actually launching, spinning up these machines, and turning them off is actually specific to Amazon. So there are still aspects of it that are proprietary.

Rackspace private cloud

The next product is Rackspace's Private Cloud. So Rackspace Private Cloud also provides virtual machine hosting. And so it's also an ***Infrastructure as a Service***. You can own whatever you want in these VMs, but the hosting is private. Rackspace's Cloud can either be a private slice of a data center that Rackspace owns, but that's only dedicated to you, or actually, Rackspace can set up private clouds within an organization's own data centers as well. And the interface to this is still a standard in the sense that it's still just running x86 code.

Amazon RDS

Now, moving up to the next level of abstraction, we have things like Amazon's Relational Database Service, or RDS. And this one, as I mentioned before, provides hosted versions of MySQL, Oracle, and other common database software. So this is one level up. This is ***Platform as a Service***. You're not just getting [INAUDIBLE] machines. Instead, Amazon is managing and running this application that's a

component of many end products you may want to build on top. The hosting is still public. It's still running in the same data center as other tenants. And the interface here is a standard in the sense that both MySQL, Oracle, and all these databases have a standard interface. And you can move to any other provider of MySQL if you want and get the same thing.

Another example of Platform as a Service is DynamoDB. As I mentioned before, DynamoDB is a Key-Value Store It has a data model, and consistency guarantees, and storage properties that are specific to Amazon. Amazon engineered a Key-Value Store that would meet certain types of high-skill workloads. So this is still Platform as a Service. It's still public, but the interface is now proprietary. This is an interface you can only get in Amazon, because only Amazon really hosts the Dynamo system. So there's a choice here. OK, do you want the performance characteristics of this interface, or do you want a more standard one that may perform less well, but be common across providers.

And finally, we have software like Tableau Online. This is an end-to-end visualization and reporting software. You don't need any developers at all to use this. This can be used directly by, say, data analysts, or business users. And this is the last level of abstraction, Software as a Service. The hosting, in this case, is actually still public. So potentially, multiple tenants could be in the same data centre all running Tableau. Technically, the hosting for this kind of software could be private as well. It just depends on the software in case. And here the interface is proprietary in the sense that it's this one

product from the one company Tableau. You can't exactly get a substitute that has the same interface.

So hopefully this has given you an overview of the types of cloud services available today. And these are just some of the things to consider when choosing which cloud to use.

2.2.5 Challenges and responses

So in the final segment, we're going to talk about challenges and responses of cloud computing. And in particular, I'm going to list four of the challenges that come up when using a cloud service, and also some of the things that users have done, and providers have done, to respond to those challenges.

2.2.5.1 Security

So the first challenge, and maybe the most talked about, is security. And the reason for this is very simple. If you're ***outsourcing computation or storage to a third party***, security and confidentiality can be a lot harder to guarantee than if you're doing that computation in-house.

This issue of security becomes especially important when combined with ***legal compliance***. So for some workloads, applications and companies that run those workloads need to follow specific laws to protect the information. For example, HIPAA is a set of laws governing the storage and computation on health data. So this means that if you're working with the sensitive information and you need to do it in a cloud, you have to make sure that the cloud services you're using also follow all the right guidelines to maintain HIPAA compliance. PCI DSS is another example. PCI is the payment card industry, and this is their data security standard. So this is something you have to follow if you're doing credit card transactions online. And so you have to make sure that with outsourced storage or outsourced computation, the provider also follows these same guidelines.

And security can also become a challenge, because your provider might be in a ***different legal jurisdiction***. So for example, if you have a company in the United States that's using a cloud hosted in Europe, the servers in that cloud have to follow the local laws in Europe. These laws might be different.

They might require information to be protected in a different way, or to be given over to authorities under different circumstances for law-enforcement reasons. And so the organization has to make sure that these legislations are followed. And same thing happens if you're a European provider with a cloud in the US.

Because of the concerns about security, there's actually been quite a bit of response to these challenges in both industry and in research. And providers and users are actually working to make the security of the cloud quite a bit better, quite a bit easier to offer.

So the first example of this movement is that providers over time have given users more and ***more control*** over the security properties of their data, and just more controls to let them ensure that the computation happens securely. For example, virtually all providers today offer encryption of the store of data. So if you place data, say, in Amazon's S3 or in a hosted database, you know that even if someone breaks into Amazon's data center and runs away with the machines or the disks, they won't be able to actually use that data.

The second feature related to encryption is key rotation. When you encrypt data, you've done so with a specific key. And over time, you may want to change the key, either because the first key was compromised, you know, someone got a copy of it, or you had a security breach, or just want to update to a newer type of encryption standard. So providers are now starting to let you do this remotely, without having to download the data locally and re-encrypt it and send it back. For example, Amazon's Redshift database has key rotation as just a built in feature.

And finally, more providers are offering fine-grained access controls-- access roles and user authentication for users within a customer organization that are using the cloud. For example, if you have, say, a retailer company that's trying to host some computations in the cloud, there might be different people within the retailer that should have different level of access to the services there. Some people should not be able to see customer data, maybe managers should only be able to see data about their own division, and so on. So using these mechanisms, it's possible to sign up different users within the same organization that have access to different parts of the cloud service. And it's also possible to acquire richer authentication mechanisms from them. For example, two-factor authentication, where the user has to carry a physical device with part of their key on it, so that it's harder for someone else to login as them. And so most clouds are starting to offer these features as well.

The next direction in which we've seen changes in work from providers is in compliance. So most providers now, for example, are PCI DSS compliant. This means that they let users build up the security properties that the payment card industry needs in order to do secure payment. So for example, providers might give control over firewalls, they might give control over encryption, so that users can securely process credit card transactions.

And there's movement to increase this to other types of compliance as well. And there's certainly interest from legislators in being able to certify providers as offering these.

So finally, on the research side, the cloud is motivating quite a few interesting developments

in cryptography and encryption that may make it possible to more securely perform computations on encrypted data. One example term you'll hear there is homomorphic encryption. Homomorphic encryption means that, at least for some operations, it's possible to do the operation on the encrypted data and get the encrypted result without the provider ever learning what the real value of the data was.

So for example, if a homomorphic encryption scheme supports sums as an operation, you might have two numbers, a plus b. And if you have encrypted values of a and encrypted values of b, the provider can just add those up or perform some other operation on them to get an encrypted value of a plus b.

And in doing this, the provider learns neither what a nor b was. It can just give you back this value of a plus b. But the customer can perform the computation using the provider's servers without having to gather all the data back locally and do it on site. So homomorphic encryption for just one operation like this has existed for a while. And lately, something called fully homomorphic encryption has also been developed, which, although it's currently highly inefficient, allows, in theory, arbitrary computations to be performed in a homomorphic way on encrypted data. So this is still and area of very active research.

Similar property is encryption schemes that let you do comparisons on the value. So for example, order preserving encryption. Here, the idea is that if you have values that you're encrypting, say, a and b, and a is less than b, then the encrypted values of these numbers follow the same property. So here the encrypted value of a is less than the encrypted value of b. And what this lets you do is, it lets you run queries on the data that compare values, without the provider ever seeing the actual values. They only see the encrypted one. But you can still tell them, hey, give me the values where the encrypted value is less than this. And so you run a query on the underlying data without revealing it. Now, order preserving encryption, unlike the homomorphic one above, necessarily has to reveal some information to the provider. So in this case, it does reveal the order of values to the provider. And depending on the domain, the provider might still be able to figure things out. For example, if these are the salaries of employees, and the provider sees the encrypted salaries, they'll be able to figure out which employee had the highest salary. Or maybe they don't know anything about them, but they can track this employee through other fields that they have in their database, and maybe understand something about them. So here, there's a bit of a trade-off between being able to do these computations and leaking information to the provider.

And finally, our third type of computation on encryption that is possible is searching on encrypted data.

So there have been several proposed algorithms and schemes to let you do text search on encrypted data without the provider learning what the original text was or what the query was. And this is another example where you might outsource computation to the cloud. So these are just some examples of advances in cryptography as applied to the cloud. And I think over the next few years, you'll

see quite a bit more interesting theoretical and practical work in this area as we begin to understand to what degree computation really can be outsourced to a third party.

2.2.5.2 Availability

The second challenge with the cloud model is availability. When you're running computations in the cloud, you give responsibility for availability of the data and of the service to a third party, namely the

cloud provider. So that means that the provider you choose must make sure that the data is stored ***reliably***, that it recovers from disasters, from power going down or a data center going away, and that any service you're running is up all the time when your users need to use it.

And a specific challenge that may come up with this is actually also ***business continuity***. So if your cloud provider ever goes out of business, the question is,

- What happens to your data?
- What happens to your computation?
- Can you easily move that somewhere else?
- Or have you lost it?

These are both important things to worry about with the cloud. Given these concerns about availability, there are actually a number of responses that users can take to minimize the risks. And there are also things that providers are doing to let users control their availability of their computations and more easily employ the infrastructure that the provider has.

So the first thing is ***location diversity*** within a provider. Most large providers, such as Microsoft, Amazon, and Google, have data centers in many locations across the world. And they expose the users, where specific resources are, so that users can make sure that their computation is geographically distributed and redundant. Most providers have support for data application across these zones. And they may also expose what are called availability zones as a concept even within one data center that tell you these racks of machines, for example, are on different power supplies. So if you put your machines across them, it's unlikely that they'll all go down at the same time.

Second response you see here is user is ***using multiple providers***. There's a lot of interest in this, a lot of pressure on providers to inter-operate well, to provide similar interfaces. And there are also third-party services that actually will manage different providers transparently for you and let you run a computation across all of them without worrying exactly about which provider you're currently on.

And finally, the third aspect to consider with availability is that even though you are outsourcing availability to a third party, the ***scale*** of these providers may let them do quite a bit more to ensure availability than on-site hosting would. So for example, having 24/7 security guards around the data center or setting up distributed application or setting up local generators in case of power failure-- these are all things that are fairly expensive for a company to do themselves until they operate at a specific scale.

But they are things that the cloud providers are doing. So in some sense, because the providers are doing this, it may also be possible to get higher availability using the cloud than by hosting your own infrastructure.

2.2.5.3 Data Transfer

Third challenge I want to talk about is data transfer. This is especially relevant because we're talking about big data. And the problem here is that ***moving data over the internet is actually really slow***. Just as an example, say you want to transfer 10 terabytes of data, that's actually not that much data over a T3 line, which is a relatively fast business connection. It's about 45 megabits per second. So transferring that much data over a T3 line will actually take about 20 days even though 10 terabytes today is only five disk drives. It's something you could stack up on your desk very easily. You can carry it in your backpack. And it costs about $400 to purchase. So the gap between storage density and internet bandwidth is very quickly increasing. And it makes it difficult to actually move these volumes around even when these volumes are quite reasonable to store and to collect.

Because of this, there have been some really interesting responses from providers. One response is that data transfer into most providers is actually free. Usually when you do network communication with the cloud, the provider has to pay for the network. And so you have to pay for part of the networking.

But moving data into them is usually free. And this is just to encourage you to load in data without covering their bandwidth cost needed to receive it. Of course, once you've loaded the data in, they're hoping that you actually run some computations in there or store it long term so that they can benefit from it.

And the second interesting response is that in addition to sending data over the internet, there are now services that let you import data by just shipping physical disks. So for example, Amazon has a service called Import/Export where they use Amazon's existing network of trucks and delivery vehicles and warehouses that can already go around and pick up and deliver packages to import just disks, like the five disks I mentioned above, into Amazon and then load the data onto servers. So using these services, you can actually move large volumes of data relatively quickly and relatively cheaply into the cloud.

2.2.5.4 Lock-In

So the final challenge we'll talk about is lock-in. And there are two types of lock-in,

- Interface lock-in
- Data lock-in.

Interface lock-in means that if you begin using an interface to write your applications that's specific to one provider, it may be difficult to then move these applications on site or across to different providers, which have different interfaces. So at the very least, you need to take care when writing you application, so that it could easily migrate to say a different storage service, or a different way of acquiring resources.

Data lock-in means that in the same way that data was expensive to move into a cloud, it can be expensive to move out. So the same problem that uploading 10 terabytes would take 20 days, downloading them from the cloud would take an equal amount of time. And so just having a lot of data put into one provider may then make it difficult to switch providers, and may put you at the discretion of essentially the provider your chose. And this is especially tricky because most types of computation you do need to be near the data. You can't just read the data across the internet and compute on it at any reasonable speed, compared to actually running computation in the same data center on the same machine.

So to deal with lock-in, there are a number of responses and different steps that customers can take to minimize their lock-in.

The first and most important one is probably just a preference for ***open or standard APIs***. If your cloud application is written against an API that you can easily replicate on site, or that you can replicate on another cloud, it becomes much easier to move that application around and to deal with the case where the provider you chose at the beginning is no longer ideal.

If using open API is impossible, a second response is to use ***wrappers over the proprietary APIs***, so that the same code written by the user can work over many different providers. And many software efforts out there today to build these common wrapper libraries over the provider interfaces and let applications move around. Just as an example, jclouds is an open source library in Java for accessing different cloud providers. It makes it possible to launch and shut down virtual machines across these providers using the same API.

And finally, to deal with the data lock-in issue, the same physical disk import and export that we talked about before for uploading data can also be used for downloading it. And this can be a way to quickly move data across providers when needed.

2.2.5.5 Conclusions

So to conclude, although they are still relatively new technologies, clouds are very exciting environment to manage and process big data. They give most organizations the ability to run much larger scale computations than they could in house, manage much larger data volumes in a reliable fashion, and get started with this technology quickly to really understand what it can do for them.

Several challenges remain with the cloud model, including legal challenges like compliance and technological challenges like availability and security. But these challenges are actively being worked on, and I think we can expect to see a lot of change across all of these domains.

Finally, as a parting thought, clouds are just the latest instance of resources and business operations being outsourced. As an example, in the 1900's, large companies generated their own electricity.

For example, the Ford Motor Company had its own power plant and decided that creating its own electricity and distributing that electric power was more efficient than trying to use some kind of public grid. But since then, things have changed, and essentially almost every company today just uses the public grid. And this essential service without which almost all business operations would no longer work is now outsourced to a third party.

So it wouldn't be at all surprising if a few years or maybe a few decades from now, computing which is another highly necessary and ubiquitous resource, also becomes a utility, and most of these operational challenges become outsourced to clouds as well.

3 Modern Databases: Michael Stonebraker

3.1 Introduction

So the first thing I'm going to do is give you an introduction, which is going to be a history lesson of where this all came from. Since I have a lot of gray hair, I was around at the time. So the world pretty much started in the 1970s.

It started with a pioneering paper by Ted Codd in CACM that, to a first approximation, invented the modern relational database model. During the '70s, there were some prototypes built.

But the next significant thing happened in 1984, when IBM released a system called DB2, which is still sold today. And they by that one act declared that relational database systems were a mainstream technology, they were behind them. During the remainder of the '80s, relational database systems gained a market traction.

And in the '90s, they basically took over. All new database implementations were basically using relational systems by the 1990s. And a concept evolved called ***"One-size fits all"***, that relational database systems were the universal answer to all database problems. Put differently, your relational database salesman was the guy with the hammer, and everything looked like a nail. So that was pretty much true.

Relational systems were the answer to whatever data management question you had. And that was pretty much true until the mid 2000s. And if you want to pick one event, I wrote a paper that appeared in the ICDE proceedings. At the time, I claimed that one-size did not fit all, that the days of universal relational database system implementations were over, and they would have to coexist with a bunch of other stuff.

Now it's seven years later, and I'm here today to tell you that one-size fits none, that the implementations that you've come to know and love are not good at anything anymore. So before I tell you why, I've got tell you what it is that I'm complaining about. So unless you squint, all the major vendors are selling you roughly the same thing.

They're selling you a disk-based SQL-oriented database system, in which data is stored on disk blocks.

Disk blocks are heavily encoded for a variety of technical reasons. To improve performance, there is a main memory buffer pool of blocks that sit in main memory. It's up to the database system to move blocks back and forth between disk and the main memory cache.

SQL is the way you express interactions. There's a bunch of magic that parses and produces query plans. These query plans figure out the best way to execute the command you gave them by optimizing CPU and I/O. And the fundamental operation, the inner loop of those query plans, is to read and or update a row out of a table. Indexing is by the omnipresent B-trees that's taken over as the way to do indexing. All the systems provide concurrency control and crash recovery, so-called ACID in the lingo. To do that, they used dynamic row-level locking and use a write-ahead logging system, invented by a guy named C. Mohan at IBM called Aries. Most systems these days support replication. The way, essentially, all of the major vendors work is that they declare one node to be the primary. They update that node first, then they move the log over the network and roll the log forward at a backup site, and bring the backup site up to date. So that's what I mean by "the implementations of the current major vendors", who I will affectionately call "the elephants." You are almost certainly using "elephant" technology. And the thing to note is that all of the vendors implementations date from the 1980s. So they are 30-year-old technology.You would call them legacy systems these days. So you're buying legacy systems from the major vendors. And my thesis is they are currently not good at anything, which is whatever data management problem you have, there is a much better way to do it with some other more recently developed technology

So the major vendors suffer from The Innovators Dilemma. That's a fabulous book written by Clayton Christensen who's on the faculty at the Harvard Business School. It basically says, if you're selling the old stuff, and technology changes to where the new stuff is the right way to do things, it's really difficult for you to morph from the old stuff to the new stuff successfully without losing market share.

But in any case, the vendors suffer from the innovator's dilemma. That keeps them from moving quickly to adopt a new technology. And therefore, their current stuff is tired and old, and deserves to be sent to the home for tired software. So the rest of this module is going to be my explanation of why the

"elephant" systems are not good at anything anymore. So I will claim that there's three major database markets, so I will segment the market into three pieces.

- About one-third data warehouse
- About one-third transaction processing, so-called OLTP
- One-third everything else.

So I'll talk about each of these markets and how you can beat the "elephants" implementations by a couple orders of magnitude in every single one of them. And then I will have some conclusions at the end to leave you with.

3.2 Data Warehouses

In this segment, I'm going to talk about data warehouses. And I'm going to explain to you why the elephants' products don't work well in the data warehouse market.

So what about the data warehouse market?

Well, almost all of you, your enterprises, have a data warehouse somewhere in your enterprise.

Typically, it has customer-facing data in it, things like sales, products, customers. And typically, you record that stuff over time, and you load that in a data warehouse. And the fact of life is that are so-called ***column stores*** are well along at replacing the row stores that you are buying from the elephant.

And why are they taking over? Because they're two orders of magnitude faster. So I'm going to explain to you in this segment why they're two orders of magnitude faster.

So the way to think about a data warehouse -- and I'll just use Walmart as an example -- every time an item goes under a wand anywhere in a Walmart store, a transaction record is ultimately loaded into a data warehouse in Bentonville, Arkansas that says who bought what, where, and when. So that's what's called a fact. And at the center of most data warehouses is a thing called a fact table that just records transactional data. So surrounding that fact table is a collection of tables that are called dimensions.

So in the case of Walmart you have a table for every store, indicating the address of the store, who the manager is, its square footage, that kind of stuff. You have a timetable indicating the time at which any transaction took place, namely in days, minutes, hours, seconds, that kind of stuff. You have a product table, which is all the products that a company like Walmart sells, giving their ID and their description.

You sell nails that are number 10 nails in aluminum, that kind of stuff. And then you often have a customer table that you get through loyalty cards, getting whatever information you have on a customer from the loyalty information. So you have a fact table in the center, and you have sort of so-called

dimensions surrounding them that are connected to the fact table by foreign key/primary key relationships. And this is what's called a star schema, meaning a thing in the center and then a bunch of edges. There's a concept of snowflake schemas, which is if the star extends more than one level deep.

And if you want to read about snowflake schemas and star schemas, check out anything written by Ralph Kimball. If you have a data a warehouse and you're not using a star schema, you should be.

So why are column stores so much better than row stores on data warehouse stuff? Well, let's look at the fact table, because that's where all the data is. So in a typical warehouse you might have 100 columns recording the individual transactions. And a typical warehouse query might be something like, well, there were four hurricanes in Florida during the 2007 hurricane season, so I want to figure out how to provision my stores for the next hurricane. So I want to figure out what's sold by department in Florida stores in the week before every hurricane contrasted with the week after every hurricane, and compare that with same-store sales in Georgia. So such a query might read four or five attributes out of 100. So if you have a row store, the problem with a row store is that you store data on disk record by record by record. So when you pull a block off the disk and into the buffer pool, you're reading all 100 rows in any given record. And even though you only want four or five of them, you're reading the other 95% because they come off the disk, because that's the way data is stored. In a column store you have to rotate your thinking 90 degrees.

You store the data column by column by column and not row by row by row. At which point, if you want to read four or five attribute, you read exactly the four or five you want and never bring the other 95% off the disk. A quick calculation shows that I just won by a factor of 25 with a column store.

So other things that column stores do is the they encode the heck out of data. Because if you have a 20 terabyte fact table you really would like to encode it as much as you can. So in a column store you only have one kind of attribute on each block of data. And so you can very easily compress one kind of thing.

In a row store, you've got 100 kinds of things on the same block. It's much more difficult to compress data. So compression is way more productive in a column store. That wins me some more.

Furthermore, it turns out that all the elephants' products have a big header on the front of every record.

Well, if you designed a column store from scratch you don't want any big headers, because they take up a lot of space and they don't compress well. So the recently written column store systems don't have any record headers. That saves you some more.

And then the last thing is that an executor does not pick up data row by row by row. It picks up data column by column by column. And it turns out a column executor is way faster than a row executor because of vector processing. When I pick up a row, I basically look at, say, the department. And then I either like it or I don't like. I either keep it or I don't keep it. Then I go on and pick up the next row. And I do row by row by row by row. In a column executor you pick up a column and you blast through all the department data, keeping only the ones you need. And so you process the whole column rather than processing stuff a row at the time. That's just way faster.

So for all of these reasons, column stores are just wildly faster than row stores in the data warehouse market.

So let's look at the participants in this market. Well, there are the elephants, who are native row stores vendors that we talked about in the introduction. They sell you a row store. Then there are more recently written new technology column stores.

Vertica, which was a startup recently bought by HP. Hana has been making a lot of press. It's a new system written by SAP. There's a system called Redshift that Amazon sells. It turns out to be a product called Paraccel that they're relabelling. And Sybase IQ was a long ago column store now owned by SAP. So they're native column store vendors, way faster than native row store vendors. And then there are companies that have realized they're on the wrong side of a technological change.

And so in transition are some companies that used to be row store vendors and are trying to convert to column stores, solving the Innovator's Dilemma in the process. So the thing to note is that the top bucket of people are 100x the second bucket.

And the third guys are somewhere in between. They're in transition. So let me tell you why column stores don't look like row stores. So you can't be both a row store and a column store. There is no easy way to take a row store and wave a magic wand and convert it into a column store.

Why is that?

Well, let me just give you three quick slides on the way Vertica works, which was a system I was involved in, now owned by HP. So column stores store data column by column by column. So think of it in the case of Vertica, think of a table as stored column by column by column and sorted on all the attributes in left to right order. Just happens to be way Vertica does it. So the leftmost column is sorted. The next one over is sorted only for those matches in the one to its left and so forth. The way Vertica stores a column is in 64K blocks that I'll call chunklets. So in 64K will fit however many attribute values will turn out to fit. The first one is stored uncompressed, and all the rest are stored compressed. Delta encoding, which is you encode each one against its predecessor, is one way they do it. Lempel-zipf is another way. Huffman coding is another way. If you have a lot of repeated values, like male and female as an attribute, well, you can just suppress the repeated values. So you're encoding the heck out of 64K chunklets. The leftmost column is in sorted order, so it's usually delta encoded. The rest of them are encoded however turns out to be the best way to do things. Chunklets are only decompressed when necessary. You can do quite a few operations without decompressing a chunklet. So lazy decompression. And the fundamental operation is pick up a chunklet and process the attribute values in that column. Looks nothing like the standard row store technology.

Now, you might say, well, column stores are kind of nice, but how fast can you load data? Because if I load a new record, well it has 100 columns in it, and those 100 columns all go in different places. So, I certainly don't want to do 100 disk writes every time I want to load a record. So of course, Vertica doesn't do it that way at all. There is a main memory row store in front of the stuff I just talked about. All the newly loaded tuples go there as main memory adds. And every once in a while Vertica picks up a whole bunch of these main memory tuples, rotates them 90 degrees, compresses the heck out of them, and writes out big, long runs of stuff. So in bulk, sort, compress, and write out. And every once in awhile you have these segments that you've written to disk. You merge them to make bigger segments, and they get bigger and bigger and bigger in an asynchronous process. Queries to Vertica are answered from both places, both the column store and from the row store.

So to a first approximation, this is what Paraccel looks like. This is what Hana it looks like. And it has nothing to do with the way row stores are implemented. So the elephants code has nothing to do with the way these systems work. And these systems are wildly faster than row stores. So over time, the only successful data warehouse products will be column stores. And the elephants are either going to go out of the warehouse market or they're going to have to morph their current row stores into column stores, which requires them to solve the Innovator's Dilemma problem. And that's the end of what I have to say on data warehouses, except for a couple of remarks I'll make in the conclusions.

3.3 Online Transaction processing

The purpose of this segment is to talk about the online transaction processing world and to explain to you why the elephants' systems are no good at this market either. So in order to talk about transaction processing, I should first do a quick example.

The way to think about transaction processing is to think of our favorite retailer, which is Amazon. You don't hit a transaction processing system until you say, buy it. When you say, buy it, your shopping cart is turned into a transaction. You actually own the stuff. And there's a transactional database implemented, by the way, in Oracle which keeps track of what you bought, figures out when to ship it, and charges your credit card for whatever the amount is, and so forth.

So this is an update-oriented world, where you want to do updates very quickly. There's essentially no big block-oriented reads like in the data warehouse market. So this is an update-oriented world. And you can never lose anybody's transactions because the last thing Amazon wants to do is send you stuff and not have you pay for it or have you pay for it and not send you stuff.

Either way, it's a big lose. So there are three big decisions to make when you're thinking about how to implement transaction processing systems.

- Main Memory or Disk
- Replication Strategy
- Concurrency control strategy : How do I sort out parallel transactions to make sure that I support so-called ACID?

So I'll talk about each of these. And I'll talk about the new way. And I'll talk about the old way and why the new way's going to clobber the old way.

So a reality check on transaction processing databases is that a one terabyte database is really, really big by transaction processing standard. And, yeah, I know there's Facebook, which has a much bigger one. But by and large, a terabyte is a really big TP database because your TP database only grows at the rate that you sell more stuff.

So if you want to store one terabyte of data, well, you can store it on disk. That's always been an option.

But you can also store it in main memory, as long as you're willing to pony up about $30,000 or less. And the price of a terabyte of main memory is dropping like a rock. So if your data doesn't fit in main memory now, then wait a couple years, and main memory prices will drop to where it'll make sense for you to put your data in main memory.

So unlike data warehouses that are really, really big and your business analysts want to keep making them bigger transaction processing databases just aren't that big. So if your data fits in main memory, well, if you're using one of the elephants' products, that will mean it's in the buffer pool cache that's in main memory. The real data is stored on the disk. So if you're doing that, here's what you're up against. So this is some data that we actually got about five years ago.

And we instrumented the Shore DBMS, which looks exactly like the elephants' products. But we had the source code for it. And so we could instrument it. And also Dave DeWitt, who wrote Shore, was here at the time. So that made it even easier to get what I'm going to talk about to work. So the data from Shore should look a lot like what you'd get from the elephants if you could, in fact, instrument their product.

And this is on a benchmark called TPC-C, which is the standard transaction processing benchmark from the Transaction Processing Council. So we measured elapsed time, which is to a first approximation CPU cycles, because everything's in main memory. And so useful work is less than 10%, which is finding the data you want and updating it 10% or less. In the artist's rendition of what I'm talking about, it's 4%. It varied depending on exactly which TPC-C command you are actually doing. But the big enchilada was four different pie slices that collectively took up more than 90% of the total time.

The first one was ***managing the buffer pool***. It shouldn't surprise you that if the data is highly encoded in the buffer pool, well, to update it, you've got to find the right buffer pool block, then find the right place in the buffer pool that your record exists, decode your record into main memory format, update it, and then reverse the process, put it all back. The elephant systems all maintain LRU ordering of the blocks in the cache in case they have to throw stuff out. You have to pin blocks in the cache sometimes. All that code, about a quarter of the overhead.

The next piece is you do a ***row level locking***. Every time you touch a record, you have to set a lock. Every time you update a record, you've got to upgrade that row level lock to a write lock. And managing the lock table is another about a quarter of the overhead.

The third thing you have to do is you have to write in ARIES style, ***write-ahead log***, like I talked about in the introduction. And that means every time I do an update, I have to assemble the before image and the after image of the record, write the data into the log, and then force the log to disk. That's about another quarter of the overhead.

The last one is a little bit surprising. And you probably haven't thought about it because the elephants' products are all ***multi-threaded***. And the whole idea long ago behind multi-threading was, if I had to do a disk read, well, then there was nothing else to do. I wanted to switch to run something else. To run something else, I needed some more threads. So all the major database systems are multi-threaded.

And long ago, they were running on one CPU. So everything was cool. These days, you've got 8, 16, 32 CPUs that are sharing main memory. You got a lot of threads. So you're running a lot of parallel transactions. What that means is that whenever you touch shared data inside the data manager, you have to latch it to make sure that it doesn't get corrupted. So when I'm setting a lock in the lock table, I got to make sure nobody else is changing the lock table while I'm changing it. So what happens in multi-threaded worlds is that everybody starts setting latches, and you end up having to wait until the previous guy gets out of your way. And the latching overhead is about a quarter of the overhead.

So there are four big slices of the pie. And the useful work is under 10%. So now if you want to go fast, it doesn't take a rocket scientist to figure out two major things.

The first one is that if I have some fancy ***improvement on B-trees*** and I can do indexing better than the next guy, well, that only impacts the useful work piece of the pie. And that's way under 10%. So better B-trees will improve overall performance by 1% or 2%. So there's no sense worrying about useful work.

You've got to get rid of overhead. Now to get rid of overhead, if you get rid of the buffer pool, well, that gets rid of one slice of the pie. But there are still three more left. So for example TimesTen is a main memory database system that's been around for about 20 years. It stores data in main memory. So there is no buffer pool. That piece of the pie is gone. But the other three pieces are still there. They do record level locking. So that overhead is there. They do ARIES-style logging. That piece of the pie is still there. And they're multi-threaded, so the latching overhead is still there. So you think about it for a minute. And you say, well, that'll go marginally faster because you're only getting rid of one of the four slices of pie. You want to go really fast, you got to get rid of all four.

So how do you do this? Well, first of all, you got to get rid of the latch overhead. Now, latches are everywhere in all the major vendors' products. So either they have to rewrite all their systems to get rid of shared data structures somehow.

Or you've got to get rid of multi-threading which, again, would be major surgery to the elephants' products. Or you somehow got to get rid of the queuing delays that latching entails. One way or another, you're toast unless you can get rid of the latching overhead.

So current systems often do this by being single-threaded. So a system I was involved in that was written here, the prototype H-Store, which turned into the commercial product VoltDB is single-threaded.

So it gets rid of the latching overhead by not being multi-threaded. So one way or another, you've got to get rid of the multi-threading overhead. You're toast unless you're a main memory database system. So that much is obvious.

Any disk-based system, you're going to die. So I'm often asked at this point, what happens if my data doesn't fit in main memory? Well, there's a great paper that I, of course, was involved in that will appear in next year's VLDB conference.

And it's called ***"Anti-Caching."*** And what it does, it says run a main memory database system and then archive the cold tuples in main memory format to a slower storage device. And anti-caching is way, way faster than caching. So you can extend past main memory. But you've got to run a main memory-oriented system.

Well, you're toast, if you do dynamic locking, which is what the elephants all do. So what do the new systems do?

Well, a thing called ***multiversion currency control*** is popular. A Hekaton is the system coming out soon from Microsoft. Timestamp order is popular. That's what VoltDB and H-Store did. And there's some other interesting ideas from the research community that are combinations of very lightweight locking and timestamp order.

And there is no system written for the OLTP market in the last decade that's using traditional dynamic record level locking. It's just too slow.

Now, what about logging? Well, how do I do crash recovery if I don't log something? Well, the answer is yeah, that's all true. But you've got to make the run time overhead go down, and down by a lot. So again, we wrote a paper that's going to appear in ICDE 2014 by Nirmesh Malvaiya. And basically, he suggests just do ***command logging***.

In other words, if I pull up to McDonald's and I say, I'll have a number six with fries and a Diet Coke, well, you can log six fries, Diet Coke, rather than log all the data changes that that transaction actually turned into. So do logical logging at the highest level possible. And again, it's a wildly faster than doing data logging. So that's augmented by the current reality, which is no one is willing to take downtime.

Back when I actually had some non-gray hair, systems would crash, and you'd try and get back up as quickly as possible. That's not what people are willing to do anymore. They all are willing to pay for the extra hardware and failover to a backup.

And so Tandem pioneered this concept in the late '80s. And so failover to a replica is now absolutely required. So most of the time you just failover to a replica and keep going. So log is rarely invoked these days. It's only invoked if you get disasters. And so you want to make the overhead of logging go way, way, way down. So what I've said is there's an old way and there's a new way. So the old way, which is the elephants code. The data is based on disk.

The new way, it's in main memory. The elephants cache disk blocks in main memory, my claim is that you want to archive cold tuples on disk, and it works much better. You don't want to do data logging ARIES style, you want to do Command logging. You fail over to a backup and keep going, and only occasionally do you recover from a log.

You implement multiversion concurrency control timestamp order. Something other than dynamic locking. And you run a single-threaded system or another mechanism to avoid any critical sections, any shared data structures, multithreading threading is a disastrous performance problem.

So there are some new-way systems. Hekaton that I mentioned a few minutes ago coming to you from Microsoft. Hana is an OLTP system, a little further out, being promised by SAP. And there's a bunch of startups I mentioned, VoltDB. There's also MemSQL, SQLFire, Clustrix, a few others.

These are all new-way systems that implement OLTP using the new-way concept. And they are a factor of 100 or so faster than the elephants. So new is way better than old. And the elephants aren't going to be able to easily change their implementations to implement all these totally different ideas. So again, they're up against the innovator's dilemma. So if you don't care about performance, if you want to run 50 transactions a second, run them on your wrist watch. Anything will run 50 transactions a second. If you care about performance, then one of the new vendors is going to be a factor of 100 faster than what you're currently running. So if you don't care about performance, do whatever you're currently doing.

Otherwise, a changeover to better technology is in your future. And that's the end of what I have to say on transaction processing.

3.3.1 Everything else (Other technologies)

So now, I'm going to talk about everything else. By now you get the drift, the elephants are no good at this stuff either. So everything else is a potpourri of interesting stuff.

Some of these which will be discusses are

- NoSQL systems
- Array databases
- Graph databases
- Hadoop - And because Hadoop-- everybody somehow thinks that's synonymous with big data, and I'm going to tell you why it's not.

3.3.1.1 NOSQL Systems

So let's start with NoSQL. So there are 75 or so vendors of which probably the ones you most likely to have heard of are a thing called MongoDB and another thing called Cassandra, and then there's 73, at least, others.

So there are three concepts that all of these NoSQL guys implement.

Give up SQL

The first one is, as the name on the tin suggests, give up on SQL. So don't use SQL. Now the basic message here is that ***SQL is too slow***. And so what you want to do is code in lower level utterances, which is what SQL gets compiled into. So basically, get rid of the whole SQL layer and code the algorithm yourself. Now, it turns out there's been 40 years or so of database research, and in the '70s there was a big debate between low-level codicil, like record at a time systems, and high-level SQL systems. And over 40 years, codicil is completely gone and SQL systems are the answer. So anybody who says ***bet against the compiler is a lunatic***.

So the standard wisdom, when I had mostly black hair, the version of this argument that was prevalent then was, you want to go fast, you got to code in IBM assembler. If you can't allocate your own registers, you're never going to go fast. Now you guys today would never advocate coding in assembler because the compilers, they get good enough that they get way better than you at doing all of this low-level stuff. So SQL is compiled at compile time into the same kinds of utterances that the NoSQL guys advocate.

Never bet against the compiler. Never ever bet against the compiler. High-level languages are good.

The database community has figured that out over 40 years or so.

So why don't the NoSQL guys realize this? Well, most of them don't know much about databases, so they are reinventing the wheel, as we'll see in a minute. But anyway, give up on SQL. In my opinion, high-level languages are really good. Do not ever get forced into writing your own low-level record at a time code.

Give up on ACID

Second thing the NoSQL guys advocate is giving up on ACID. ACID is too expensive. Well, we saw in the last module that ACID is expensive when it's implemented by the elephants in their legacy systems.

So that's true, what they're saying is true, but as we saw in the last segment, there are all kinds of new ideas on how to make ACID go fast. And so when the NoSQL guys complain about ACID, they're complaining about 35-year-old implementations, not about modern stuff.

So giving up on ACID because you think it's too slow is just a statement that you're not paying attention

to the marketplace and modern systems these days. Second thing is that if you need ACID, then do not write it yourself in user code. Your hair will be on fire if you try and get ACID straightened out with user-level code. It's complicated, messy stuff.

Let the database experts do it. So if you are guaranteed that you won't need ACID, you won't need it now, and you won't need it three years from now when your company , which used to be in the plumbing supplies business, suddenly decides to buy beauty salons. So if you are guaranteed that you won't need it now, and you won't need in the future, then fine give up on ACID. Otherwise, you have a fate worse than death on your hands, which is you're going to have to implement ACID in user code.

So the NoSQL guys advocate giving up on ACID. Number one, you don't need to if you run one of the modern DB systems, and number two, you're making a bet that you won't need it in the future, and if you bet wrong, I don't want your job.

Schema Later

The other thing, the third tenet of the NoSQL guys, is schema later, which is they complain that the relational database vendors require you to say create table. You've got to think about your data upfront,

decide what the table structure is, and before you can load a record, you've got to decide what your data looks like. So schema later says just start loading data,and you'll figure it out eventually.

I'll give both sides of the argument.

First one is, I'm not quite sure what my data's going to look like. I'm just going to start loading, and I'll figure it out later.

The second point of view is to say, if you are supporting a production enterprise application, and you don't think about your data upfront, you're going to dig yourself into a horrible pit and a modicum of thinking about it upfront will save you a disaster later.

So depending on your point of view, schema later is either a good idea or a terrible idea.

And that's kind of all I'm going to say. I come from the camp that says, think about your data upfront,

and you'll be way better off downstream. But not everyone agrees with this.

So now let's look at the NoSQL guys in aggregate. And I would say Cassandra and Mongo are probably the most prominent vendors, so let's look at what they're currently doing.

Well, they're both inventing, or have invented, high-level query languages, which look like SQL. So they are moving quickly toward SQL.

Secondly, let's talk about ACID. Well one of the biggest proponents of no ACID is a guy named Jeff Dean from Google. So he's largely responsible for MapReduce and BigTable and most of Google's database offerings. And he just wrote a paper on a system called Spanner and basically said, I was misguided. I thought ACID wasn't necessary, but, boy, it's certainly a good idea, and Google's most recent system has it in it. So the world is moving toward ACID as people realize that ACID is an awfully good idea.

So in my opinion, NoSQL used to mean NoSQL, then the NoSQL guys said, well, what it really means is not only SQL. We want to do this stuff-- maybe SQL's good for some things, but not for everything.

And in my opinion, NoSQL currently means not yet SQL. So I think there's going to be a convergence, which is the NoSQL guys are going to move strongly toward the transaction processing world, which is SQL, or things that look like SQL, along with ACID, and they're going to sort of look more and more

like conventional database technology off into the future.

So if you're running NoSQL, mostly what people seem to use NoSQL for is low-end applications that aren't particularly mission critical, sort of webby things, if you want to store some web data. If you've got to store everybody's password, that's one table that you always look up by row.

So if you have some low-end simple applications, I'm not against NoSQL. Most of the systems are open-source. They're free or very cheap. You can get them up quickly. They're easy to learn. So there's certainly a market here, especially at the low end for these guys. And, of course, NoSQL guys don't particularly look like SQL engines. Their implementations are moving toward the implementations from the elephants. But there's certainly a separate market here, so it remains to be seen how big this market will be.

3.3.1.2 Complex Analytics and Array Databases

So let's move on to a different topic. So I want to talk about complex analytics. What that means is the world of the quants. Any time you hear machine learning, predictive models, recommendation engines, estimators, dot dot dot, that usually goes under the rubric 'data mining.' And your quants and your data scientists are the people who do this stuff.

So what is this stuff all about?

Well, by and large, all of this is computations that are defined on arrays as collections of linear algebra operations. And they don't look at all like SQL. And often, you've got to do them on big amounts of data at scale. So you want to do this kind of stuff on big data.

So let me just give you a really quick, easy to understand example. Let's look at, suppose you're a quant on Wall Street, and you're trying to write an electronic trading engine. So what you do is you've got 20 years worth of data on the closing price of every stock for a long time. And you say, well, if I'm going to write a trading engine, I've got to find out what patterns there are in this sea of data. So let's assume that I start off by saying, well, I don't know anything yet. So let me pick two stocks say Oracle and IBM and say, I wonder if they're correlated. So I can compute the covariance between the closing price of those two stocks. And all the orange stuff at the bottom is covariance. And if you don't know what that means, have a quant explain it to you. Anyway, that's what it is. But that isn't really what you want to do. You really want to do it for all pairs of stocks. I don't know what patterns relate to what patterns, so I want to do some data mining. So I want the covariance of all pairs of stock.

So what it is is I have a big array, which is I have order 15,000 stocks, that's how many publicly traded stocks there are in the United States. You multiply it by 4 if you want them everywhere in the world. And if I've got 20 years worth of data, that's about 4,000 trading days. So there's a row for every 1 of 15,000 stocks that has 4,000 closing prices. So I have a 15,000 by 4,000 matrix. It's a matrix. And what is covariance on this table?

Well, to a first approximation, ignoring constants and ignoring subtracting off the mean, it's that array stock times matrix multiply times its transpose. So this is a matrix calculation on matrices. And it's a program that involves matrix multiplication and transpose. So you want to do this at scale. You've got this orange stuff to do, and do fast.

What do you want a database system to do? Well, you want it to be able to do these kind of calculations. You want them to scale to lots of cores, many nodes, out-of-memory data, run it on big data.

But you want to be able to mix this with data management. Because I want to leave out outliers. I might only want those securities with a market cap over $10 billion, or only those securities with a headquarters in New England. So I want to be able to mix and match this kind of stuff with data

management operations, and make it work at scale. So that's the purpose of array databases.

If you fundamentally have to do array calculations, linear algebra, well, if you're running a table system, then you've got to convert back and forth between tables and arrays, because tables are not arrays. So the general idea is to say, well, if I've got to do a lot of this kind of stuff, let's have an array database system.

Let's not do tables. Let's build arrays as the native storage object. And to do data management, I need SQL. That's what everybody wants. But I wanted defined on arrays so I can do joins and filters. So you've got systems that give you a race equal with all kinds of quant-oriented linear algebra built in. And if you don't like what's there, they all give you user-defined functions so you can write your own fancy analytics.

So SciDB is an example of this kind of stuff. And this world is going to get traction as you guys move from standard SQL business intelligence to this more complicated world of data scientists. So this will get traction if and when, well, as you move to more complicated analytics. And the systems don't look at all like the traditional wisdom.

They're built on arrays. They're not built on tables. They don't look anything like what the elephants sell you. So there may or may not be a sizable market for array database systems, depending on how fast and if people move to complex analytics. And this doesn't look like the traditional wisdom at all.

3.3.1.3 Graph Databases

The last thing I want to talk about is graph databases. And there's lots and lots of interest in this topic in the research community, and there's starting to be some commercial implementations. So the whole idea is that if you look at Facebook's data, you know, it's I like Sam Madden, and both of us happen to be followers of Madonna.

Facebook is one big, gigantic graph. And when Facebook does calculations, like the average distance between me and Sam Madden as such and such, it's the distance in that graph. So Twitter is a graph.

Facebook is a graph.

There's a lot a graph data in the science world. So graphs are, conceivably, a very interesting data structure. So the first system that I'll just mention is a system called Neo4J. That stores native graph data and is focused on being able to update a graph quickly. So you want to be able to add nodes and edges to a graph quickly.

The other thing that you often do with graphs is you compute analytics. You know, like the shortest path between me and Barack Obama is whatever it is, but what's the distance of separation between us? So shortest path, minimum cut set, that kind of stuff, there is a dozen or so standard analytics that people want to run. And you could build a system that focused on doing analytics on graphs fast as opposed to doing updates on graphs fast.

In this analytics market, it's interesting to note that you can recast a graph as a sparse array. Meaning that if there's an arc from i to j, all that says is that in the ij-th cell of an array, there's something there as opposed to nothing.

So array database systems can simulate graphs at, perhaps, very high performance. Relational database systems can also simulate graphs. So the jury is still out whether there is an interesting system to be built focused on analytics.

It's real clear that if you want to do OLTP, things like Neo4J are a pretty good idea.

3.3.1.4 Hadoop

Well, we've talked about data warehouses and we've talked about transaction processing. And now we're on to everything else. And everything else was four different kinds of systems. There was NoSQL.

There was array databases. There was graph databases.

And now, Hadoop. And Hadoop deserves its own segment, because somehow a bunch of marketing

people have somehow decided it's synonymous with big data. So what do I mean by Hadoop? Because in the marketing world, it means various things to various people.

So Hadoop was written originally by Yahoo. It's an open source version of Google's program called MapReduce. So that's what it is. It gives you, as you might surmise, two operations

- Map
- Reduce.

And map is basically things that look like filters, things that look like transformations.

And reduce is basically what you would call a roll up.

So Hadoop is very good at what I'll call ***embarrassingly parallel operations***. So let's say I have a million documents. And I want to find all the documents that have a strawberry followed within four words by a banana.

Well, I can divide up the million documents over as many nodes in a computer system as I want. And then a map operation, I can write it in any scripting language that I want. So you could write it in Java.

You could write it in Python, whatever. And then send that Python script as a map operation to all the various nodes. Each of them goes through their documents that in parallel picking out the qualifying fruit records.

And then as they roll up at the end to assemble the result, which is a single reduce operation.

And so document search is very easy to parallelize. It's embarrassingly parallel. Hadoop is very, very good at this kind of stuff.

Now, Hadoop has morphed over the years into a complete stack. So this thing called MapReduce, which was originally called Hadoop, is in the middle. But what people have found is that lots of times, they want to code queries in, guess what? SQL.

So there's a layer on top of a Hadoop. Hive is an implementation written by Facebook. Pig was an implementation written by Yahoo. Think of them as SQL look-alikes.

So that's at the top layer.

In the middle, you process SQL by compiling it into these Hadoop MapReduce operations.

And MapReduce runs on top of HDFS, which is a file system. So file system at the bottom, and MapReduce in the middle and SQL at the top. And this runs across any number of nodes. So this stack will scale to as many parallel nodes as you want to set up.

And in fact, Facebook is running a gigantic Hadoop stack over, I think, 2,500 nodes. So people are setting up very big Hadoop stacks across many, many systems.

So what is this stack good for? Well, it's very good at embarrassingly parallel computations like document search. We already talked about that.

The second thing you might say it's good at is, what about warehouse style queries? We talked a couple segments ago about Walmart and provisioning stores for the next hurricanes. Now if you run that sort of query, well, you write that in Hive. So from now on, I'm going to call Hive SQL. So you write that in SQL. So that SQL, when you run it on the Hadoop stack and compare the performance with a column store, you're going to run 100 slower. So you're going to run a factor of 100 slower than current high-performance warehouse technology.

So you can have the answer in an hour on Hadoop, or you can have the answer in a minute on your favorite data warehouse. Which one would you rather have?

Or put differently, you can run it on 2,500 nodes in Hadoop, and you can run it on 2.5 nodes in a parallel database system. Do you want to run 2,500 or 2.5 nodes?

So if you're doing SQL aggregates, the Hadoop stack is disastrous. Let's suppose you want to do the covariance calculation that we talked about in the previous segment.

So you want to do linear algebra. You're going to be a factor of 100 worse than an array database system. So again, have the answer in an hour, have the answer in a minute, your choice.

So complex analytics, Hadoop is a disaster. These are usually coded in a thing called Mahout, which runs on top of Hadoop. It's a performance disaster.

Another thing you might be doing is scientific kinds of codes. So if you want to do computational fluid dynamics simulations, well, there are lots of scientists who do that. They use so-called MPI-based systems that are, again, going to beat Hadoop by a factor of 100.

So the net net is if you've got an embarrassingly parallel calculation, by all means, use the Hadoop stack. Otherwise, you can beat Hadoop by a factor of 100 by something else.

So let's take a quick look at Hadoop usage.

While Facebook is a very big user of Hadoop, more than 95% of their traffic is in hive, meaning SQL, meaning aggregate, meaning factor of 100 slower than a parallel database system. I have a colleague at Lincoln Labs, Jeremy Kepner. Jeremy has estimated that Lincoln Labs, which is 4,000 or so scientists, 95% of their usage is not embarrassingly parallel.

So around the world, it looks like 5% or less is embarrassingly parallel, and 95% would be way better served by something else.

The Hadoop proponents aren't stupid. So Cloudera and Hortonworks, Hortonworks is the Yahoo guys moving out to their own company, and Facebook are huge proponents of the Hadoop stack. They are all doing exactly the same thing.

So Cloudera has a system called Impala. Impala is an execution engine that implements hive, ie SQL.

And it has nothing to do with a MapReduce layer. It's a complete execution engine built on top of the file system. And it looks exactly like modern data warehouse systems. So it looks, to a first approximation, a lot like Vertica or a lot like HANA.

So Hortonworks and Facebook are both doing exactly the same thing, which is they're defining and building an execution engine that processes SQL without ever using this MapReduce layer. So these guys are effectively moving to an execution engine that's Hive-oriented, ie data warehouse oriented, and they're going to compete in the data warehouse market.

And guess what? All the data warehouse vendors support Hive. So if you want to process Hive, you will have a lot of options. You'll have column stores from the data warehouse guys. And you'll have stacks from these guys. So life will be interesting. But in any case, it's not going to involve MapReduce. It's going to involve processing SQL in data warehouse-like worlds, because that's 95% plus of the market.

So the most likely future is there is a small market, say 5%, for embarrassingly parallel applications. The current Hadoop stack is pretty good at that, 5% of everything. 95%, there's a much bigger market for a Hive SQL framework. And the data warehouse guys are focused on this. The Hadoop guys like Cloudera and Hortonworks are focused on this. And there will be a great collision as they fight for market share.

In my opinion, it's unlikely that HDFS is going to survive. Because it is also horribly inefficient. So the warehouse guys are currently will support HDFS if you want to run that file system. That's called the go slow command against what they're currently doing.

So HDFS has a huge performance penalty. So the whole stack is rotten.

The MapReduce layer is rotten. It's going away. HDFS is rotten. It may or may not go away. This whole market is in a considerable amount of flux. And I guess you have to hold onto your seat belt. And I'll have some conclusions for you in this world and in some other worlds in the conclusion section.

3.3.2 Conclusions

In the last four segments, we've talked about the various database sub-markets, and concluded that the elephant's products are no good at any of them. So I now want to summarize with some closing thoughts.

The first one is data warehouses will be a column store market. There is no question in my mind that that's going to happen. It may take another decade for it to completely unfold, because database systems are very sticky. It takes a while to change.

So if you are not running a column store now, I guarantee you, you will be sooner or later. So what do you have to do? Well, ask whoever your current vendor is what his column store plans are. If he doesn't have any, then you better switch, or better plan on switching. And that will be a costly and time-consuming operation. But you're going to have to do it because there's a factor of 100 to be gained. And you're going to see that that's worth it sooner or later. So find out if your vendor is moving to a column store fast enough that you can just wait and move with him. Or you're going to have to switch.

OLTP is going to be a main memory database market, with anti-caching if your data is too big. If you're not running a main memory database system now, you will be in the future. And ask your vendor if main memory is in his plans. If he's going to implement a main memory system, and it's not compatible with what you currently have, then you have a costly conversion on your hands. At which point, you might as well look at lots of other technologies. So if you're not running one now, you're going to have to switch or switch with your vendor as he switches, if he switches.

Arrays database systems and graph database systems may or may not gain traction. At the very least, you should understand what they're good for, what they're not good for, and whether you ought to consider one.

NoSQL is going to be popular for low-end applications, especially things that look like document management, web stuff, and especially places where you really do want schema later and you think that schema later is a good idea. And they're going to be ACID-less for a while. So don't ever consider NoSQL for in an area where ACID might be required. Otherwise, your hair will be on fire.

The Hadoop stack is going to morph into something that isn't going to look at all like its current stack. So hold onto your seat belt. At the very, very least, look at your proposed and current Hadoop applications. See if they're embarrassingly parallel. If they are embarrassingly parallel, they'll continue to run on the current stack without a disaster. Otherwise, what's going to happen as you scale up, you're going to hit a wall. And you're going to have to switch to something else.

And there will be lots of products there to pick up the pieces. So you're in deep doo doo if you are contemplating non-embarrassingly parallel applications on the Hadoop stack. You're guaranteed to have to switch as you try and scale. Think about it now. Don't think about it later.

The current products from the relational elephants are only going to survive in low-performance applications. If you don't care about performance, run DB2, run Oracle, run SQL Server. They're only going to survive where you don't care about performance. In other cases, something else is going to run circles around them, and you're going to switch.

The curse is that may you live in interesting times. In my opinion, the relational database world was pretty dead in the '90s. One size fits all relational databases were the answer. There was one kind of system. Everybody used it. In the last decade, there's been lots and lots of new database ideas and products.

NoSQL has come into existence. New TP databases have come into existence. Graph databases, array databases, Hadoop has come into existence. Lots of new database ideas. The world is very vibrant, and the database world is very vibrant.

One thing that's going happen for sure is your business intelligence folks will keep putting more and more and more stuff in their data warehouses. My favorite example is a decade ago or so, I got to visit a beer company in Milwaukee. And they had a traditional data warehouse of sales of beer by brand, by time period, by distributor and all that sort of stuff. So this was in November of a year where there was an El Nino event predicted by the weather guys. So El Nino is equatorial upwelling of Pacific warm water. And it screws up the weather in the United States in the winter. And so it becomes wetter than normal on the Pacific coast and warmer than normal in New England. So I said, well, El Nino's coming.

The weather guys say so. Are beer sales correlated with either temperature or precipitation? And the business intelligence guys scratched their head and said, boy, that's a good question. I wish I could answer that question. But of course, weather data wasn't in the warehouse. So this is the kind of stuff that just generates pressure to add more and more and more stuff to warehouses.

So they will get bigger and bigger and bigger. There are probably two dozen petabyte size data warehouses, that I'm aware of right now, running in production. And they're just going to get bigger.

There's going to be a sea change from simple analytics to complex analytics. Why is that going to happen? Well, right now, you have your business intelligence guys running SQL queries to say, what's my average sale by department, by store? So that'll give you today's data. You'd much rather have your data scientist build a predictive model to predict next month's sales by store, by department. And that's complex analytics. So you get much more sophisticated reasoning about what's happening. But you've got to move to complex analytics. You've got to move to the world of the data scientists. You've got to move away from your business intelligence guys.

Now, I got to make a call on a very large insurance company. And I said exactly this, which is, you guys are going to put a sensor in everybody's car to record how and where and when they drive. And you're going to build complex risk models based on all these variables. And you might as well get going.

And they said, yeah, we should do that. But our business analysts aren't up to this level of sophistication. And they're really hard to hire.

So the sea change is going to happen. It's going to be gated on you being able to hire competent data analysts. And that will take however long it takes. But you better get going, because it's in your future.

The internet of things is a force to be reckoned with. Everything on the planet of material significance is going to get sensor tagged. I already mentioned your car by your insurance company. Other things that are going to happen, the MIT library would like to sensor tag every book. They don't want to do it because undergraduates might walk out without checking them out. That's not it at all. If a librarian misshelves a book, it's lost forever. So they want to be able to find it if a librarian misshelves it. Parking spaces on the MIT campus want to be sensor tagged, so that instead of driving around and around in the parking structures looking for an empty space, you'd just be directed to one that exists. Marathon runners get sensor tagged. All kinds of animals get sensor tagged. Your kids get sensor tagged if you want. So the world is going to get sensor tagged. And that's going to generate a data deluge like you haven't seen before. So if you think data is a deluge now, just wait for the internet of things to kick in.

Future times are going to be really interesting. There are going to be all kinds of changes that you are going to have to cope with. And my best advice to you is to hire a really good chief data officer to help you sort all this stuff out.

In conclusion, the future is going to be really different. If you aren't able to move with the times, you're going to die. Because your data is your most valuable asset. And if you can't leverage it, you're going to lose.

4 Distributed Computing Platforms: Matei Zaharia

This segment would be discussion about distributed computing platforms to run general computations on large clusters. So let's start with the motivation for distributed computing.

Today, in many domains, large data sets are inexpensive to collect and store, but unfortunately processing these data sets is requiring higher and higher degrees of parallelism. Just as an example, one terabyte of disk space today costs about $50 if you just go out and buy a terabyte disk. So it's very inexpensive to collect large amounts of data if you have an application that generates them.

However, reading through this one terabyte disk end to end takes about six hours. To do anything quickly with these large data sets we need to parallelize them. And for example, even if we parallelize the one terabyte of data across 1,000 disks, it would still take 20 seconds to read. It's actually still not exactly interactive.

Now in other parts of this course, you've seen work on speeding up query processing. But in fact, not just query processing but all computations that we do on data will need this scale.

For example, loading the data, transforming it, and indexing it into a format that enables fast queries is also a large scale computing platform and needs to be done in parallel.

In the same way, complex analytics functions that might require custom code will also need to be parallelized over large clusters.

So in this lecture, we're going to talk about how can we program these large clusters. Just as an example, here's a picture of a Google data center that Google put out this year.

And you can see in there it has thousands of computers organized into these racks. There's actually a fairly detailed network topology that dictates which computers can easily talk to other ones. And applications needs to spread out across this whole data center to run their computations. So Google's internal applications regularly use over 1,000 nodes just on one application and these same kind of data centers are available externally as a cloud.

Now traditional network programming in the past has been based on message passing between nodes. This is a very natural way to write distributed programs. You can have a program on each node that sends messages to other ones. Unfortunately, message passing is very difficult to be done at large scale due to several problems.

The first problem is ***how to split the computation across nodes***. The way we split the computation has to consider the network topology as well as data placement throughout the data center because otherwise moving excessive amounts of data over the network will be too slow.

The second problem is ***how to deal with failures***. Typically if you have a single server that you're running computations on, failures are a very uncommon event. For example, average server today might fail once every three years. However when you put together more nodes, the probability of a failure happening in a small time interval increases.

So for example, if you put 10,000 of these same nodes together, you start to see ten faults per day. And so any long-running computation needs to handle failures within the computers running it.

Finally, our third problem that happens as you put more and more nodes together are ***stragglers***. This is a case when a node hasn't outright failed but it's simply going slower than the other nodes. And if your goal was to parallelize the computation across 1,000 nodes and 900 of those nodes finish but 100 remain slow, then the computation as a whole will still be going slowly.

So as a result of these problems, almost nobody uses the message passing model directly to write computations on large clusters.

Instead a wide array of new programming models have been developed.

In this lecture, we're going to talk about one of the more popular classes of parallel computing models designed to solve this problem, which is ***data-parallel models***.

In data-parallel models the programmer simply specifies an operation that it wants the system to run on all the data. And the programmer doesn't care where exactly the operation runs. The system gets to schedule that. And in fact, it's even possible for the system to run parts of the operation twice on different nodes to do things like recovering from a failure.

The main example of such a model in use today is MapReduce which is the model that was introduced by Google and popularized by the open source Hadoop platform. There's quite a bit of software in this space. Even open-source software to handle these kinds of computations, offers a lot of diversity. So these names here on the slide are some of the names you might see associated with big data processing.

I'm just going to cover these systems a little bit in turn and group them into classes.

So first of all, at the top left we have Hadoop. Hadoop is the open source implementation of Google's MapReduce model. And Hadoop has also been used to build higher level system on top that compiled down to MapReduce, including Hive which runs SQL queries, and Pig. We're going to cover these in more detail later in the lecture.

Next there are some generalizations of MapReduce which are able to express more types of computations efficiently. And one of the main ones is Microsoft Dryad model, and then systems built on top of that like DryadLINQ. Another system that generalizes Dryad further is Spark, which also supports in-memory data sharing betwen computations. And this has also been used to build systems on top, such as Shark, which does large-scale SQL queries. For specific domains of computations there are also more specialized systems. For example, after developing MapReduce, Google developed the Pregel computation model for large-scale graph processing. And there are several different open-source implementations that either follow this model or extend it, including Giraph and GraphLab. Lab For large-scale, interactive SQL queries, Google developed Dremel, which is a large-scale parallel SQL tool. And there are open-source systems such as Impala and Tez that implement similar models.

Finally, for real-time stream processing, systems like Storm and Samza follow a different programming model from the batch systems you'll see, but it is still a data-parallel model. These systems are used for a wide variety of applications.

First and most importantly, in most types of organizations these general purpose platforms are used for ***extract, transform, and load, or ETL*** workloads, which means taking data that arrives in potentially an unstructured format or just an arbitrary external formats, such as say, log files or JSON records, and transforming it into forms that enable faster queries, for example, indexing it.

Google's MapReduce was first developed to support its ***web indexing*** pipeline and similar systems are used at Yahoo, Microsoft, and other large-scale search engines.

Yahoo uses Hadoop, among other things, for ***spam filtering*** and Yahoo Mail.

Netflix uses it for ***product recommendation***, to figure out which movies you are likely to want to watch.

Organizations like Facebook use it for ***ad hoc queries***, where maybe there wasn't time to index the data into the right format to support the query and they want to go back to the raw data.

And finally, financial organizations are using these systems to do ***fraud detection*** with large-scale data.

So in the rest of this lecture, I'm going to start by talking about

- The challenges in large-scale computing environments.
- MapReduce model and show you some examples of how it can be used.
- Limitations and extensions of MapReduce.
- Few other types of platforms that have been developed to tackle the large-scale programming challenge.

4.1 Large Scale Computing environments

In this module, I'm going to talk about the goals and challenges of large-scale computing environments and the typical software stack to handle them. Let's start with the goals of large-scale data platforms.

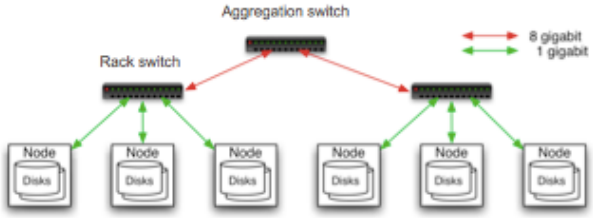
Platforms like MapReduce really have two goals.

The first one is ***scalability*** in the number of nodes. These platforms need to parallelize input and output across hundreds or thousands of nodes to quickly scan large data sets. As I mentioned at the beginning of this lecture, scanning large data sets on a small number of disks can easily take many hours.

The second major goal is ***cost efficiency***. When you move to a large scale, every source of cost starts to matter in the efficiency of the overall computation. In particular, these systems are designed to run on ***commodity machines***. These are cheap but less reliable than, say, the highest-end enterprise systems. They're designed to use a ***commodity network*** which is widely available but has low bandwidth compared to the fastest networks available today. They're also designed to perform ***automatic fault tolerance*** so that fewer administrators need to run around and fix servers that will inevitably go down. And finally, they're designed to be ***easy to use*** so that fewer advanced programmers are needed to run computations on these clusters.

Here's a picture of a typical Hadoop cluster that might be used on MapReduce today. These clusters are usually organize into racks, as shown here, and usually you may have 20 to 40 nodes with in a rack, and thousands of nodes within the cluster. Now, the network connection between these nodes is also heterogeneous. Within a rack connections are usually fast. For example, with a new cluster today, this

might be a 10 Gigabit-per-second connection. But across racks, the bandwidth out of a rack is usually much smaller than its internal bandwidth. For example, a rack with 40 machines in it might have only a 40-Gigabit-per-second out link, which means only one gigabyte per second per machine.



- **40 nodes/rack, 1000-4000 nodes in cluster**
- **10 Gbps bandwidth in rack, 40 Gbps out of rack**
- **Node specs (Cloudera): 8-16 cores, 64-512 GB RAM, 12×2 TB disks**

The nodes themselves usually have many CPU cores, fairly large amounts of memory, as well as many disks. For example, for Hadoop clusters today, it's recommended to use at least 12 disks. And often, organizations use more in order to store and scan more data per node.

These specs here at the bottom are recommended by Cloudera, one of the main Hadoop vendor today. Here's a picture of one of these clusters. This is a Hadoop cluster at Yahoo. And you can see how the servers are organized into racks and the whole set of racks together form the cluster. So this cluster environment comes with three main challenges.

Cheap Nodes Tend to fail

The first one is that when you put together cheap nodes, cheap nodes tend to fail. And this becomes especially true if you have many of them partaking in the computation. As I mentioned earlier in the course, mean time between failure for a single server today might be every three years. But when you put 1,000 of these nodes together, the meantime between failures become one day.

So the solution for this is to build fault tolerance into the system at all levels. So that each piece of the computing stack can recover from failed nodes.

Commodity network → Low Bandwidth

The second problem is that these commodity networks in use today, which are cheap, also offer low bandwidth. So as a result, placement of the data and computation is important. And the solution to this problem is to have systems automatically push computation to the data. The runtime systems know which machine has each piece of data and send computations for that data to that machine.

Programming distributed systems is hard

Finally, the third problem is that programming distributed systems is hard. And without any care taken, these systems would only be accessible to very advanced programmers. So the solution to this is data parallel models, where users just write simple functions to apply on the data. For example, map and reduce. And the system automatically handles work distribution and failures.

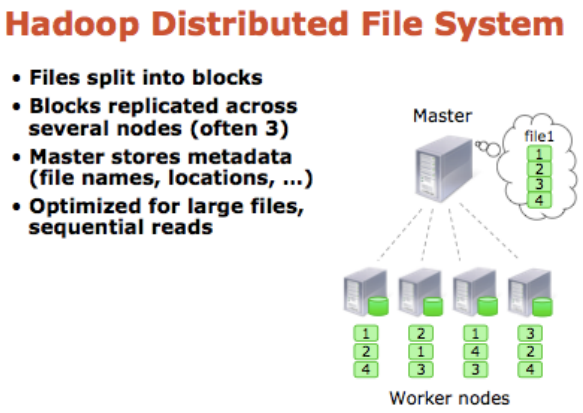
Finally, I went to sketch, quickly, the typical software components used in these systems today. And I'm going to talk, in particular, about the Hadoop stack, which is the most widely used one. Usually on these clusters you have two systems.

Distributed File System

First is distributed storage system such as a distributed file system. And one example of this is Hadoop's HDFS. This allows you to have a single name space for all the data in the cluster and to refer to files that potentially span many machines. These systems also typically replicate the data for fault tolerance. For example, the most common setting is to replicate the data three ways.

MapReduce System

On top of the file system, you also have a computing system such as MapReduce. And Hadoop comes with a MapReduce implementation to do this. The computing system runs jobs that are submitted by users and manages the work distribution and fault tolerance inside each computation automatically. It's co-located on the same nodes as the file system so that it can push computation directly to the machines that store that data and finish it fast.



Lastly, I want to say a bit more about distributed file systems such as HDFS so that you can understand how they interact with the computing platforms on top. These systems support very large files and automatically split each file into blocks. And the blocks are then spread out across multiple machines.

Each block is also replicated to several nodes, usually the number is three, so that it can be stored reliably. And then a master node in the system stores metadata such as the file names, the locations, and the set of blocks that make up each file.

For example, in the picture on the right here, we have a master for HDFS and one file in it, which has four blocks. And each block has three replicas on the cluster. For example, block one is on these machines over here. And block two is on these machines over here.

Finally, these distributed file systems are optimized for a very different workload than the file system you have on your machine. They're designed specifically for large files in the gigabytes to terabytes in size. So they're not going to perform very well for, say, collections of small files. And they're optimized primarily for sequential reads as opposed to random access or writes throughout the file. So now that we have this background on cluster computing environments and storage systems we're ready to jump into the processing models on top.

4.2 MapReduce Model

Now that we've seen the challenges of large scale computing environments, how do we tackle them?

I'm going to talk about the MapReduce model introduced by Google to address this task.

MapReduce is a simple data-parallel model where the user specifies two functions to run on the data called ***Map and Reduce***.

And in MapReduce, both of these functions operate on key-value records.

The Map function takes input records from an input file which have a special input key type and input value type. And given each record, it produces a list of intermediate records which have different intermediate key and value types.

After that, the reduce function receives each key in the intermediate data, as well as a list of the values produced for that key by all Mappers. And it produces a list of key value pairs as output in response.

As an example, here's a really simple MapReduce program that we might write in Python to compute Word Count. In this program, we just want to find out how many times each word occurs in an input file.

In this case, let's say that our input value to the Mapper is a line of text. What we'll do in the Map function is, we'll split each line of text into words. And we'll output the word as the key and 1 as the value for our intermediate data.

So now for each word in the file, we have a value of 1, potentially for every instance of that word. Then in the Reduce function, we receive the key, which is the word here, and the set of values. And we just output for each word the sum of those values to get the total count of that word in the file. Let's look at how the Word Count application executes.

Given this application, MapReduce will first look at how the input file is distributed across the cluster, and automatically break it into blocks. Then it will launch a Map Function to process each block. Now in this case, the input file is text, and each block contains several lines of text.

The Map Function is going to apply the user's code, which, as we saw before, is splitting the line of text into words. So in this example, it's going to take each line and output a record for each word. For example, the first Map task produces four records. "The," "brown," "fox," and "quick" here at the bottom.

And for each record, the key is the word, and the value is a 1. Each Map function, as it runs, saves its output to the local disk on the machine that ran it. And that's important because that will allow us to potentially re-run the Reduce task later and re-play output from the Maps into them if they fail. Once these output files are written, MapReduce launches Reduce tasks which are going to fetch them across the network and shuffle together the data for each key to the same machine.

In this case, we launched two Reduce tasks.

The first Reduce task over here is going to receive a subset of the words. So for example, it receives all copies of "the." It gets one from here, and two from the one down here. And then it applies the user's Reduce function, in this case, adding up all the 1's, to produce an output record. Other keys might be sent to other Reduce tasks. For example, words like "quick" and "mouse" I sent to the second Reduce

task, and its output here at the bottom. Each word will go to only a single Reduce task, so all the values for that can be processed together.

Now in general, in MapReduce programs, users can choose the Partitioning function across the Reduce task, but the most common choice is just simple hash partitioning where each word goes to an essentially random, but consistent, task.

Finally, once both the Map and Reduce tasks have finished, MapReduce writes its output into a file and completes.

So let's go into more detail on how MapReduce executes computations.

First, MapReduce automatically splits the words into many small tasks.

In the picture before, we saw the Map task and Reduce tasks. Then it sends the Map task to nodes on the cluster based on data locality. So MapReduce knows which nodes have copies of each input block, and it automatically schedules Map task for that input blog done on those nodes so that it can read the data locally without sending it over the network.

Finally, MapReduce systems automatically load-balance these tasks as they finish. Usually, the system will split the computation into hundreds of thousands of tasks. And as each node finishes it's current task, it's assigned a new one from the queue so that slower nodes and faster nodes get different amounts of tasks. And tasks are executed as soon as new nodes become available.

In addition to splitting up the work, MapReduce also automatically recovers from failures. I'm going to talk about the failures cases here.

First of all, there's the case when a task crashes. If a task crashes, MapReduce will simply try it on another node. Now this is OK for Map tasks that read the input data because they had no dependencies. They can run on any node, and just take the input data and read it again. For Reduced tasks, re-running the task is also OK because the Map task in MapReduce buffered the output on disk, so that a new Reduce task can go back and fetch the same outputs and start processing the same Reduce function. Of course, if the same test repeatedly fails, that means that there might be a bug in the task itself. So in that case, we can either ignore this task and the whole job. Note that for this Fault Recovery mechanism to work, the user code given in the Map and Reduce functions needs to be deterministic. That means it needs to produce the same result each time you write it. And it also needs to be idempotent. It needs to be OK to run the code multiple times. For example, code that updates an external database or maybe writes something out to our storage system needs to ensure that it will produce the same answer, even if a task runs twice.

The second failure case is if a whole node crashes. In this case, we lose not only the current tasks on the node, but also the outputs of Map task that the node was previously on. So in this case, MapReduce is going to re-run the lost tasks that were on the node, but it will also re-submit these previous Map tasks. And these will run in parallel across other nodes of the clusters to re-build the output, so that this output can then be served to Reduce tasks that haven't yet fetched it.

Finally, there's the case when a task hasn't outright crashed, but is just going slowly. This is called a ***straggler***. And in this case, if we don't do anything about the task, it might slow down the whole computation as the other nodes in the system wait for it to finish. So in this case, MapReduce will automatically launch a second copy of the task on another node and let the two copies race to finish. Whichever copies finishes first is the one whose outputs we'll use, and the second task will be canceled. And doing this is OK in MapReduce because it's designed to be able to run each task multiple times in order to also tolerate faults. So user code should be able to deal with this case when a task is running twice at the same time.

So in summary, by offering the data-parallel model, MapReduce can handle many of the issues of distribution automatically. It can automatically figure out how to place tasks based on data locality, how to load-balance them across nodes, and how to recover from both stragglers and faults.

In the next segment, we're going to use the MapReduce model to implement a few example applications. By writing the right map and reduce functions, you can make the system compute several things.

Example : Search

Our first example is search. In this case, the endpoint is a large text file. And we split it into records, for example one record for each line of text. Here the key might be the line number for the line. And the value might be the actual line of text itself. As output, we want the numbers and text of all the lines matching a given pattern. To do this, we can have a map function that applies the pattern to each line and only outputs the lines and line numbers that map the pattern. Then for the reduce we can simply use an identity function. Or as an alternative, in many MapReduce platforms it's possible to have no reducer and just run a map-only job. As a result the maps will filter the data set in parallel and only the records matching the pattern will be produced.

Example : Sort

Our second example is a sort. In this case, we have a set of key value records. And we want to output the same records but sorted by the key. To do this, we can use MapReduce's control for partitioning the data between the map and reduce phases. In particular, for map we can just pass an identity function. We don't need to change the records. And we can do the same for reduce. But for the partitioning function p that chooses which records go to which reduce tasks, we pick a function that preserves order, in this case that maps records with lower keys to a lower numbered reduce task than other records. In this case, for example, we've split up the records by first letter. So records from A to M go to the first reduce tasks and the ones from N to Z to the second. Now all the words with letters at the beginning of the alphabet, like ant and aardvark, are going to the first reduce task. And words at the end of the alphabet, like zebra, are going to the second one. Now most MapReduce systems automatically sort the input that's read to each reduce task in order to group the records by key. So as these records

pass through the reduce tasks, the records on each task are already in sorted order. And so the whole output across the cluster is sorted.

Example : Inverted Index

Our third example is building an inverted index. This is a simplified version of what you might do, for example, to build a web index for Google search. So in here, input is one record for each file, like say each web page. And the key is the file name. And the value is all the text in that file. And as output, we want a list of files containing each word. Once we produce this list of files, we'll be able to search for a word and quickly find all the documents matching it. To implement this application, we use the following map and reduce functions. In the map function, we take the text for each file and split it into words. And we output a record where the word is the key and the file name is the value to say that that word occurred in that file. Then in the reduce task, we receive for each word the list of file names that include it. And we output the word and maybe just a unique set of file names, in case a word occurs in the same file twice. This picture here shows how inverted index executes. We have two documents here, Shakespeare's plays Hamlet and Twelfth Night. And they each contain a bunch of words. When we run the map function, for each word we get a key value pair showing which file it was in. So for example, "to" occurred in hamlet.txt. "Be" also did. "Or," "not," and so on all happened inside this file. From the second file, which may be running in a different map task, we get a different set of words. For example, "be," "not," and "afraid." Finally, when we reduce these results together, we'll get a list of the file names for each word. So for example, "be" happened in both files. But words like "to" happened only in Hamlet. And "greatness" happened only in Twelfth Night.

Example : Most popular words

Lastly, as a fourth example, suppose we want to define the most popular words in a data set. In this case, the input is again these file name and text records, one per file. And as outputs, maybe we want the top 100 words that occur in the most files. So how are we going to run this computation? This is an example of a computation that can't easily be done with just one MapReduce pass. But it's possible to do it in two passes by combining our solutions for inverted index and sort. So in this case, we're going to submit 2 MapReduce jobs.

In the first pass, we're going to build an inverted index giving us this data set of words and list of files that they occur in.

Then in the second pass, we're going to map each record with a word and list of files to just counting the number of files in there as the key and passing in the word as the value. And now we have key value pairs showing the count of each word. Then we're going to just sort these by counts, using the sort job I showed before, and start reading the first 100 records in the output to get the final result.

This pattern of combining together MapReduce jobs to perform a bigger computation is very common.

And it's motivated a lot of the work we'll cover next on extending and generalizing MapReduce.

4.2.1 Limitations of MapReduce

While MapReduce was extremely influential in distributed computing, it's not the end of the story. In this section, I'm going to talk about some of the limitations of the MapReduce model and systems that people have built on top of it or extending it to address these challenges. Let's first start by looking at some of the limitations of MapReduce.

Now MapReduce itself is great at building single pass analytics applications. But most real-world applications end up requiring multiple MapReduce steps. For example, Google's indexing pipeline uses 21 different steps that put together components such as sorting or index building that we saw before to actually produce Google's ranking of web pages. Even simpler analytics queries, like breaking events into sessions or taking the top K can require two to five MapReduce steps. And iterative algorithms, which are algorithms that need to make multiple passes over the data, such as PageRank, can take tens of MapReduce steps.

Multi-step MapReduce applications create two problems, ***programmability and performance***.

Programmability

On the programmability side, multi-step jobs can lead to convoluted code. So for example, if your application takes 21 MapReduce steps, that's 21 mapper and reducer functions that need to be put together in the same program and need to be chained in the right order to produce the right result.

The second problem is that in writing these functions separately, there's a lot of repeated code for common patterns such as sorting, grouping, or joining data sets.

Performance

On the performance side, MapReduce only knows about one pass of computation. So if your job needs to perform multiple passes, you need to write data out to the distributed file system in between these steps. And that can often be quite expensive and slow.

MapReduce is also inefficient for applications that need to re-use data, for example multipass algorithms like PageRank that iterate over the same data set or interactive data mining, where most queries focus on the same subset of data.

Given these programmability and performance challenges, there have been two reactions.

The first one was to build higher-level programming interfaces over MapReduce. These translate the higher-level language into MapReduce steps and make it easy to hook together operators and produce a more complex computation. At the same time, these systems will automatically merge steps to optimize performance so that they produce as few MapReduce jobs as possible and minimize the impact of the expense of data sharing across jobs. Two examples of such systems are Hive and Pig.

A second step people have taken is to generalize the model itself. So there are now programming models such as Dryad and Spark that support more than the one pass of MapReduce and can run other types of computations more efficiently.

In the rest of this section, I'm going to talk specifically about the higher-level interfaces. And then our next section is going to cover the generalizations.

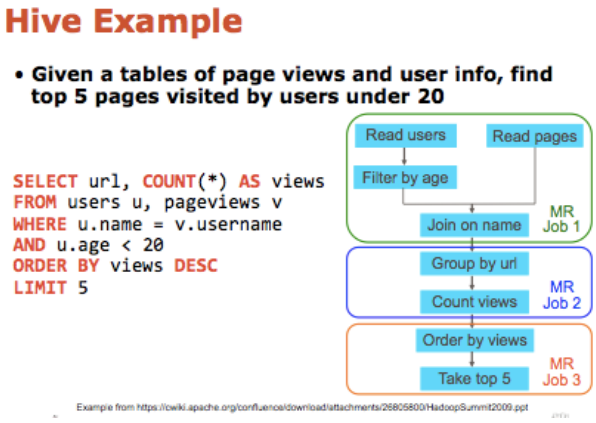
4.2.1.1 Hive

So let's start with a quick look at Hive. Hive is a relational data warehouse built over Hadoop. Much like an analytical database, it supports a catalog of tables with schemas. And it knows the storage format and records of each table so that it can optimize queries on them.

Hive runs queries written in a subset of SQL. So it's immediately familiar to SQL programmers. And it supports many of the features of a traditional database, including query optimization, as well as several features specific to the MapReduce environment, for example complex data types, such as nested structures, and calling into MapReduce scripts that can then be combined with SQL.

Hive was initially developed at Facebook and is currently used on more than 90% of Facebook's MapReduce jobs. Let's look at an example Hive application.

In this application, we have a table of page views, as well as a secondary table of information about users, such as each user's age. And we want to find the top five pages that are visited by users under 20. So this is the SQL query we'd write in Hive. If you're familiar with SQL, this is a standard query to compute this count and then sort the pages by views. In particular in this query, we join the users stable and the page views so that we bring together the user record for each user with the page view for it based on the user name and we filter out only page views where the user's age is less than 20.



Then we take these page views. We aggregate them with a count. And we sort the result to get the URLs that are visited by users under 20. This diagram on the right shows how this application will execute as a graph of operations. In particular, we begin by reading the users and pages tables.

And we can then filter the users by age. Once we do that, we do a join to bring together user and page view record with the same username. We can do a group by by URL, a count, and then an order to find out the top five. Now executing this as a series of MapReduce jobs is fairly complicated. There are many operators in here that can be implemented as MapReduce. But you'll have usually at least three or four jobs. When you pass this application to Hive, it automatically comes up with an efficient plan to execute this as MapReduce steps.

So this first box at the top shows the first MapReduce job that Hive will run. And this actually combines four of the operators. It combines reading the users and the pages, which can be done as map functions, and then doing the filter, and then doing the join as a reduce. These have all been fused together into one job. The next job is going to do a group by on the URL and then count the views for each URL. And finally, the final job we'll do a sort, similar to the sort we saw before, and take the top five. So given this query, Hive has automatically generated these jobs.

Pig Example

```
Users = load 'users.txt' as (name, age);

Filtered = filter Users by age < 20;

Pages = load 'pageviews.txt' as (user, url);

Joined = join Filtered by name, Pages by user;

Grouped = group Joined by url;

Summed = foreach Grouped generate group, count(Joined) as clicks;

Sorted = order Summed by clicks desc;

Top5 = limit Sorted 5;

store Top5 into 'top5sites'
```

The example above here shows the same (hive) example in Pig. I'm not going to go into a lot of detail around Pig. But I just wanted to give you a sense of what it looks like. So similarly to Hive, Pig provides a set of high-level operators and groups them together automatically into MapReduce jobs. And if you look at the example here, you see many of the operators that we had on the diagram before.

For example, you can load files. You can filter. You can join, group by, or sort. Unlike SQL systems, Pig's language looks a lot more like scripting. So it's immediately familiar to scripting programmers, although it provides the same SQL operators.

Pig also differs from Hive in that it doesn't have a central catalog to manage metadata. Instead, you specify the schema of each data set as you load it. This can make it very flexible for bringing in a new data set and immediately starting to process it. Pig was initially developed at Yahoo and is widely used inside it to run most of its MapReduce jobs.

Hive and Pig are examples of systems that are built on MapReduce to make MapReduce programming easier.

4.2.2 Generalizations of MapReduce

Having seen MapReduce and some of the models built on top of it, we're now going to talk about generalizations of this model. Although compiling high level operators to MapReduce makes it easier to program, and more efficient, multi-step applications still have limitations because the MapReduce runtime supports only one pass.

In particular, sharing data between MapReduce jobs is unnecessarily inefficient. MapReduce doesn't know how data was constructed across the jobs, so it needs to hide it through this replicated storage system in between steps. And in many applications, writing data to this replicated storage system can easily take 80% or 90% of the time. In addition, some applications need to do multiple passes over the same data. And because the only storage primitive in MapReduce is a distributed file system, they have to go out and read the data from disk on each pass.

So in this one, I'm just going to show an example of the first problem of a multi-step application. And in this case, we're talking about doing a word count and then doing a sort to find the most common words in a file.

We can easily do this as two MapReduce jobs, just applying the word cloud on the first step and then the sort on the second one. But between these jobs, all of the records with the words and their counts have to be written to a file. This has to be written to a distributed file system like HDFS, where it's replicated across the network for fault tolerance, and also persisted to disk. This can add quite a bit of overhead in writing this application.

To address this problem, one of the first models proposed to extend MapReduce was Dryad. And Dryad generalizes the models to support arbitrary graphs of tasks. So in particular, instead of being limited to the two passes of Map and Reduce, Dryad lets you chain together multiple classes of operators into an arbitrary graph. For our word count example, for instance, we might do the grouping of words in a first set of operators here, and then immediately sort the output, and do this all as a single application without writing to a file in between. This can save a considerable amount of time, because we don't need to replicate and persist to disk all the intermediate data. Within the Dryad graph, the system uses a similar recovery mechanisms to MapReduce. In particular, if a task in the graph goes missing, Dryad can replay it, based on input from the parent tasks. And in fact, if any subset of the tasks go missing, it can look at the whole graph and figure out an order to replay them in, to recompute the result.

Similarly, to deal with stragglers, Dryad can replicate a slow task onto another node and take the output of whichever copy finishes. The Dryad model was developed at Microsoft and is widely used to power applications such as Bing. Spark is a second model that further extends Dryad to support more types of applications. The problem is that even though Dryad supports these richer operator graphs, the data flow within a Dryad graph is still acyclic. So data begins at one end of the graph, and passes through, and comes out of the other end.

The issue that Spark addresses is applications that need to efficiently reuse data. For example, in interactive data mining, a user might run multiple queries over the same subset of the data. The queries are not known in advance, so they can't be specified in advance as a Dryad graph. But they still need to share data efficiently between them.

Similarly, in iterative algorithms, the algorithm will make multiple passes over the data, and we may not know in advance exactly when it will finish. It may depend on the computations it's performing on the data. So this is also difficult to express as just a single Dryad graph. I'll show you an example of how iterative algorithms executive at the bottom.

In this example, we have a master that's driving the computation, and a data file. The master launches multiple passes over the data. In each pass, it sends some tasks to process the data. They each run some computation and send back a result. And the master only converges when a specific condition is met. For example, when the result hasn't changed across passes. So after doing the first pass, the master might send out a second pass to run over the same data. You know, do the same processing and get back the same result. This is an example where it's important to share the data efficiently between these passes. So the Spark model lets users explicitly build and persist distributed data sets that can be shared efficiently across arbitrary computations.

In particular, it offers an abstraction known as Resilient Distributed Datasets, or RDDs. RDDs are collections of objects that are partitioned across the cluster and can be stored on the nodes, either on disk or in memory for very fast use. They're built through graphs of parallel transformations, such as map, reduce, and group-by, similar to the graphs that I use to compute results in Dryad. And RDDs are automatically rebuilt on failure by the runtime system. Spark offers this abstraction embedded in several programming languages, including Java Scala, and Python.

Interactive Log Mining Using Spark

As an example, I'm going to show Spark code that you can use for interactive log mining. In this example, we're going to have a set of log files distributed across a cluster, and we're going to just load the error messages from them into memory into an RDD and then interactively search for various patterns. In this example, I'm going to show Python code through Spark's Python API. This code can actually be typed interactively at the Python console to search through the data interactively.

I'm going to show this example as Python code, and you can actually type this code interactively at the Python console to search for data in an ad hoc manner. So, here's a picture of the cluster we'll be hunting on. We have a master node that's called the driver, and a set of workers, and then on the left I'm going to show the code that we type in. The first thing we're going to do is load a text file. This is a file sitting in the Hadoop file system, and we load it as a collection of lines of text. So, this represents our first distributed data set, or RDD. After that, we're going to apply transformations to this file.

For example, here we're going to filter the file to pull out lines that start with error. The code in orange here is Python syntax for function literal or a closure, so you can actually type in any Python expression as a lambda function and pass it through the cluster, or you can type a function earlier in your program and pass it in by name. And this filter returns a new data set, or RDD, built by transforming the previous one. Maybe after filtering out the error messages, we want to do more computation. For example, maybe these are tab separated messages and we want to pull out field number two, which is the actual message. We can do that using a map operator, and again, we pass a lambda function that splits each line by tabs and then pulls out field number two.

Finally, we can tell Spark to persist these messages in memory. So, in this case, Spark is not going to load the whole file into memory, but it will load just the error messages to let us do queries on them quickly. Now, at this point nothing's executed on the cluster yet. Spark builds up these transformations lazily and only executes them when you launch the first query so it can come up with an efficient execution plan. So, let's do that next. We're going to do another filter and then a count to count the number of error messages that contain foo. Now, once it sees this operation, Spark knows that it has to launch a computation on the cluster. Count is a special type of operation known as an action, and count has to return back a number as a result. So, it can no longer just lazily build these. It has to actually computer number and give it back. To do this, Spark starts by looking at where the data is laid out across the cluster, and much like MapReduce, Spark sends out tasks based on data locality.

Each node then processes its blocks and sends back a result. In addition, each node builds up a cache of the partitions of cached RDDs that it's built along the way. So, in this case, each node has just its local error messages in memory. Next time we do a query on this data, maybe foo wasn't in the problem, so we search for bar. Spark is going to know that the messages are cached in memory, so it

can send tasks to the nodes again based on memory locality. It can process these blocks, and it can send back the results much faster. As a result, interactive queries on Spark can be quite a bit faster.

For example, doing a full text search of Wikipedia, which is about 60 gigabytes of data on 20 machines, takes about 30 seconds with on disk data. If we load this data into memory into an RDD, Spark can do this search in about one second. This makes it possible to work interactively with large data sets that may be too big to process on a single machine.

The RDD model in Spark also automatically provides fault tolerance. In particular, RDD is built by a graph of transformations that's called its lineage, and it automatically tracks this graph in order to

rebuild lost data. In this example I'm showing a slightly more complicated computation where we pull out records from a file and we group them by type, and then we reduce by key operation to add up the number of records for each type. So this is actually just counting how many records in the file have each type. After that, when we've completed this count, we filter out, we want to find the types of record whose count is bigger than 10.

So each oval here is a distributed data set, or RDD, and they are chained together by this graph of transformations. Whenever any node in the system fails, or even if multiple nodes fail at the same time, Spark can use this graph to automatically recover lost data. So in this case, it's going to see that two blocks are missing. And it's first going to rerun one reduce task, and then a filter task to rebuild the missing data. So Spark takes the same kind of recovery model that is available in Dryad or MapReduce, but makes it available for these persistent memory data sets.

The RDD model provides several benefits. First of all, it can express general computation graphs similar to Dryad. It's possible to chain together these high level operators to build up a computation, and the operators available are very similar to the ones in Hive and Big that let you build many common data processing tasks out of the box.

Second, because RDDs remember their lineage graphs, there's no need to replicate the data for fault tolerance. Instead, when we build a new data set, we can write it onto the machines at memory speed.

We don't have to hide it across the network at the much slower speed of the network to provide you reliability, because we can always re-compute it on failure by rerunning the computations. Because of this property spark also uses less memory than replicated systems, because we only need one copy of each block.

Finally, the persisting in Spark is only a hint. The system knows the full lineage of each piece of data, so it can keep computing it even if the whole data set doesn't fit in memory. In this case, the system gives users the option to spill data to disk, or it can just drop all data and re-compute it if it's needed in the future.

I've shown Spark so far on a simple interactive mining example, but the model also provides significant benefits through iterative algorithms. For example, here I'm showing two common machine learning algorithms. K-means clustering and logistic regression that both need to go over the data multiple times.

I'm not going to go into these algorithms in detail, but these algorithms all follow a similar model to the picture I showed at the beginning of this section, where they make many passes over the same data.

Typically they would make 10 to 20 passes. The first batch of these algorithms, reading from disk, can take one or two minutes, as shown here by the red bars for Hadoop MapReduce. But with Spark, at least, subsequent passes can be much faster, in this case just a few seconds, due to the much higher bandwidth of memory compared to disk. So for iterative algorithms like this that go over the data many times, if the data can fit in memory, models like Spark can run much faster.

In conclusion, both Dryad and Spark are examples of generalizing MapReduce, but they follow many of the same underlying principles of data parallel computation and re-computation on faults.

4.2.3 Other Platforms

In this last segment, I'm going to cover several other types of platforms that are available for distributed computing today. Earlier in this lecture, we saw MapReduce and its extensions, and these are the most popular models in use today.

But this area of parallel programming models for clusters is still seeing very rapid change and innovation, and there have been a lot of proposals and approaches for doing other types of computation or approaching the problem in different ways. So I'm briefly going to sketch two related questions.

The first one is how these models that we've seen today relate to databases and SQL, which are another major way of doing data-intensive computation at scale.

And second, what are some of the models emerging that are very different from MapReduce?

And here I'm going to talk about asynchronous computation and streaming.

On the database side, one of the most interesting things is that as soon as users started building large MapReduce clusters, they immediately wanted to run SQL. And so there's been a lot of work on making SQL execute efficiently in this commodity cluster environment, as well as work that's brought these MapReduce and traditional database architectures closer and closer together.

So first of all, several systems have built SQL engines over MapReduce-like cluster architectures. These go beyond the systems that translate SQL into MapReduce in that they're specialized engines that look a lot more like parallel databases, but still aren't on commodity clusters.

One of the most well-known examples is Google's Dremel, which Google built to run short queries on the order of a few seconds over their distributed file systems.

Cloudera, Impala, and Apache Drill & Tez are examples of open source systems that follow a similar model.

Usually, these systems diverge from MapReduce in that they don't provide fault tolerance within a query. But they target such short queries, on the range of seconds, that it's unlikely that something will fail during the job. And if it does, they just re-execute the whole query.

Other systems implement many of the optimizations in traditional database engines within a MapReduce-like context. For example, Google's Tenzing implements many database optimizations

and techniques such as indexes on top of MapReduce so that it can run large SQL queries on Google's clusters.

And Shark implements an in-memory column-oriented database on top of Spark. The reasoning behind these systems is that several different factors make analytical databases fast, and many of these factors can also be implemented within MapReduce or other runtimes to achieve similar benefits. In particular, there are three factors these systems target.

First is efficient storage formats, for example, column-oriented formats or compression. These systems implement the same data formats within MapReduce records.

Second is precomputation, such as building indices, partitioning the data along a key in advance, or computing statistics about it. Many of the SQL and MapReduce systems support at least rudimentary versions of these.

And finally, there's the powerful idea of query optimization, given a query plan and these statistics, figuring out the right execution strategy. And multiple systems attempt to perform query

optimization on MapReduce.

Just as an example, Shark, which is an implementation of the Hive SQL language on top of Spark, implements column-oriented storage and processing within records seen by Spark. So the Spark system doesn't need to know anything about column-oriented storage or any of the compression formats necessary. It just sees large records that contain bytes. But within each Spark record, Shark includes hundreds, maybe thousands, of database records that it's compressed into an efficient form.

Spark also supports partitioning of the data by key to speed up searches over it and statistics when it gets loaded into the system so that it can come up with efficient query plans.

Finally, an interesting result of implementing SQL on top of the Spark engine is that it becomes possible to combine SQL with Spark code. For example, from Spark programs, users can load the Shark library and run a SQL query to load it as an RDD that they can then run Python or Java code on. And from SQL, users can call programs written in Spark such as k-means clustering and apply them to SQL data, such as fields from a table. This is a powerful way to combine the complex analytics made possible by

MapReduce with the convenience and efficiency of SQL.

The second class of systems I want to cover is asynchronous computing. If you look back at MapReduce and the related models, such as Spark, all of these models use deterministic and synchronized computation in order to enable fault recovery. So if any node goes down in MapReduce or Spark, the system wants to make sure that it can recompute the exact same data and pass it through as if no failure happened. While deterministic replay is convenient, some applications can actually converge without this.

For example, numerical optimization methods and many machine learning algorithms that are based on them can actually tolerate losing parts of the state doing the computation just fine. In general, these systems run as a series of steps that try to make progress towards a goal. For example, they're trying to optimize a function through gradient descent. Now, if we lose some state on one of the steps, that just moves us farther away from the optimal solution, but it doesn't mean that we have to cancel the whole computation. So losing state slows the system down, but there's no reason to hold everything back and recompute. We can just start up where we left off and still make progress towards the goal.

Asynchronous systems break the determinism on the MapReduce model and use this property to improve their speed. For example, when they launch a parallel computation, these systems don't need to wait for all the nodes to finish in order to advance or in order to communicate their partial results between themselves. If a node is slow or if a node has failed, the other nodes can keep making progress and keep converging towards a result. Examples of asynchronous systems include GraphLab for graph processing, Hogwild, which is a machine learning system, and Google's DistBelief, which is Google's implementation of deep learning.

Finally, a third class of applications that can require very different runtime engines is stream processing. In these applications, we want to process streams of data as they arrive in real time, and because of the latency constraints, the systems that run these applications often look quite different. One popular example of a streaming system is Storm, which provides the ability to run continuous stateful operators that process parts of the data stream. In this system, the user builds a graph of operators much like in the Dryad model that we saw before. But these operators are long-running, and the system plays streams of records through each operator in real time. As they process these records, nodes can update internal states that they can track across records. For example, a node might be trying to aggregate data across time-- like, for example, finding the most popular tweets within a small time window.

In these systems, fault recovery guarantees are often fairly different from how they were in batch systems. For example, in Storm, the system only guarantees that it plays each records through at least once, which means that, if one of your operators fail, the new copy of the operator might see records that were already processed by the previous copy. This makes the application, perhaps, more challenging to program, but it also makes the system easier to recover and more efficient, because it doesn't need to track exactly which records were processed by the failed node.

Finally, even in the stream processing model, several options exist that implement fully deterministic fault recovery. For example, Storm includes a layer called Trident that uses transactions to update the state as records come through and implements exactly-once semantics, which means the result you get at the end is the same as if each record passed through each operator exactly once. Another approach is to run the streaming computation as a sequence of batch jobs. This is implemented, for example, by Spark Streaming. So here the idea is to use the same mechanism for fault recovery and execution within each step and hold the state in memory using the Spark system between steps. So in conclusion, large-scale cluster environments are difficult to program using traditional methods like streaming processing but are necessary to quickly process big data.

As a result, the past few years have seen a profusion of new programming models to simplify writing these applications, including the popular MapReduce model and many derivatives that extend the model or change aspects of the computation to run new workloads more efficiently. Across these models, the main idea is to capture computation in a more declarative, data parallel fashion that tells the system what the programmer wants to do without specifying exactly where each piece of the computation has to run. This lets the runtime automatically handle distribution, work placement, and fault recovery. Together, these systems are making it easier and easier for ordinary programmers to tackle the large-scale infrastructure needed for big data.

5 NoSQL/NewSQL : Samuel Madden

I'm going to talk to you in this module about a very important topic in big data, namely, how do we store that data? I'm going to start off by doing a quick survey of the traditional way that people have stored their data for their applications, namely in database systems.

I'll talk about the properties that traditional database systems provide, and then I'll talk a little bit about how those properties aren't a great fit for some applications in the big data era.

Before I do that, I just want to start with a little bit of terminology. I want to define two terms, ***transactions*** and ***analytics***.

A ***transaction*** is a database operation that fetches or updates a small piece of information inside of the database. Transactions are a really important workload the database systems have to support, and they're used very widely in things like web applications. So, for example, in your bank, you would be running a transaction every time you logged in and accessed your bank account. The transaction would go read your bank account balance. Another transaction might transfer some money from bank account A to bank account B, so on and so forth.

So in a transactional system you have lots of users concurrently using the system, each running some set of transactions on their behalf in order to fetch or manipulate their data.

OK, let's contrast transactions with analytics.

So in an ***analytic*** database what you're trying to do is to access or read a large number of historical records. So unlike a transaction, which only looks at one record a time, an analytical workload might look at, for example, the entire history of banking transactions, in order to analyze how much money has been transferred over some period of time, or to understand what investments a set of customers are currently making.

So in this module, we're mostly going to focus on this transactional workload, and then in future modules we'll talk a lot more about analytics.

All right, so let's understand a little bit about some of the properties that these conventional transactional database systems provide. They provide a number of really powerful features.

Record Oriented Persistant Storage -

So the first one is what I call record-oriented persistent storage. So by a record, I mean instead of storing data in a large collection, like a file where, for example, all the bank accounts might just be stored as a collection of uninterpretable bytes, in a record-oriented system each bank account balance would be stored separately and independently. And the system would provide the ability to access those records one time or manipulate them one at a time.

Carefully Structured Data (Schema)

In addition, the database system typically requires and enforces that every record inside of a particular collection, sometimes these collections in the database system are called tables-- the records inside of one table all conform to what we call a schema. So a schema just means that it describes the structure of each of these records.

So for example, in a bank account each record might correspond to a particular customer, it might represent that customers balance, maybe something about their interest rate, so on and so forth.

Powerful Query Language (SQL)

Third, these database systems provide a query language. A query language is just a way to access the data and manipulate it. The most common query language in the world is SQL, or "SQL", and this is used in almost every database system in existence.

SQL queries are sort of a high level way of describing what data you would like to fetch from the database. So here's a very simple query that fetches my bank account balance. Queries can get much more complicated than this. They can do things like access records from multiple tables, modify records, and so on and so forth. We'll see some more examples later in this module.

Transactions (ACID Semantics)

Finally, database systems provide a really powerful tool, transactions. Sometimes transactions are called ACID semantics or ACID properties. And this stands for ***Atomicity***, ***Consistency***, ***Isolation***, and ***Durability***.

For our purposes, we're really most concerned with this atomicity property. Atomicity lets us take a collection of queries and group them together with something that I'll call the all or nothing property. So what this means is if I have a group of statements or a group of operations on the database that manipulate a collection of records, we can ensure that either all of those statements successfully complete or none of them do.

And I'll give you an example about why this is important, again going back to our bank account balances. Imagine I have two bank accounts, A and B, and I want to do a transfer of some money from bank account A to bank account B. If I'm not careful, the database system might withdraw the money from bank account A and then crash. So at that point, we've effectively lost money if we're not careful,

and we really would like to avoid this from happening. So what I'd like to do is to ensure that the withdrawal from A and the subsequent credit to bank account B happen together. Either they both happen definitely, or none of them happen and we didn't lose any money from bank account A, right?

So if the system crashed, in sort of halfway through this operation, what transactions ensure is that we're able to recover somehow from this crash.

These transactional properties hold, not only in a single system, but developers have even figured out how to enforce them in a distributed setting, where the database is spread across multiple machines. And often this comes at the cost of some complexity inside of the database system. We'll talk about this a little bit more.

All right, so let's understand how these conventional properties map into this new world of big data.

In big data, there really are 1,000 flowers blooming. We have a ton of different types of data that people want to store to represent. And it's no longer the case that these properties that these conventional transaction-oriented database systems provide are exactly a perfect fit for each one of these applications.

They're a good fit for some applications, like banking, but others they may not apply so well. So let's try and understand a little bit about what some of the properties we might like in this big data era are. In big data, what has happened is in effect, we've had a proliferation of systems, new data storage systems for all different types of requirements.

And some of the new capabilities that these new systems provide are things like really, ***really high throughput, high performance***, the ability to run operations on thousands or even tens of thousands

of database records per second. And this is something that these conventional systems weren't necessarily very good at.

In addition, these big data requirements have introduced the requirement that our ***data be spread across multiple machines***. Sometimes these databases are just too big to fit on a single computer. Or, we may require the data to be stored on multiple machines, in order to be able to mask the failure of one of the machines inside of the database system.

Third, there's a requirement that these database systems be really, really ***highly available***. By that I mean that no matter what, they continue to provide answers. So rather than enforcing this ACID semantics, this idea that the data is perfectly consistent, in these highly available systems sometimes we'll emit inconsistent data, data that doesn't necessarily reflect these sort of groups of transactional operations, in order to ensure that the system is online.

Finally, this notion of the SQL query language, even though it's a very powerful language, it's not always the right language for some of these new big data applications.

So what we're going to do next is to try and understand some of these properties in a little bit more detail, and look at why it is that conventional database systems don't necessarily provide them.

5.1.1 Big Data Requirements

Now that we understand some of the properties of conventional transactional database systems. And we've talked a little bit about how those properties aren't a perfect fit for the new era of big data. I want to talk a little bit about some of these big data requirements in more detail.

5.1.1.1 Requirement : High Throughput

So the first requirement that these big data applications have is really, really high throughput.

So imagine that you're running Google. Google needs to be able to process millions of web searches per second. That's not something that a conventional SQL database system is going to be able to do.

These conventional systems were designed in an era where all the data is stored on disk. Every time you manipulate a record or modify it, you have to go do a disk write. And that can take milliseconds.

There's no way you're going to be able to process millions of such operations per second. So these new big data applications require really unusually high throughputs.

I've just listed a few examples here, things like financial trade processing or serving ads in addition to things like running a search engine.

Conventional databases are not engineered for such high rates.

5.1.1.2 Requirement : Distributed, Highly Available

The second requirement is what I call distributed, highly available operation. So distributed is a natural requirement when you have big data. When you have lots of data, you may just not be able to put all of it on a single machine.

Or when you need to process a really large number of requests, you may need parallelism in order to satisfy all of them. In addition though, these systems, especially when you start spreading data across multiple nodes, need to be able to tolerate the failure of some of their machines. If I have hundreds of different computers all processing a given query, it's probably not OK if one of those systems goes down, for me to simply to refuse to process the query.

So let's again go back to Google. Imagine that their search index is spread across 1,000 nodes. If one of those nodes fails, it's probably better to return the answer from the other 999 of them rather than refusing to serve any searches at all, even though what we've done by doing this is to give up a little bit of consistency.

Now some of the users who don't see that one machine, may not see all the search results reflected. But in a system like Google, it's probably OK. Because none of us really has a perfect idea of what the results to any given search are that we send to Google anyway.

5.1.1.3 Requirement: New Programming models

So the third major requirement of these new data storage systems is that they sometimes require new programming models.

So for example, rather than having all of my data conform to this record oriented structure that we talked about in our relational database systems, I might want to store something like a collection of json documents.

Here, for example, is a json document that represents information about a movie being sold in a products database. So in this particular document, you can see that not only do we have information about the sales price and the skew number and other things like that for this product that is being sold, but we've actually nested hierarchically information about the movie itself inside of this record.

This is very different than the way that we would have represented this information in our conventional record oriented SQL database. If we say what we're doing is storing a new type of data, a json, we're going to need a new query language in order to access data that's represented in this new way. In addition there's been an awareness that some conventional SQL systems are not very efficient at processing these SQL queries. See SQL is very powerful, but it can also sometimes be a little bit slow.

And so some of these new SQL systems provide a slightly less rich programming interface than SQL that can be considerably more efficient.

And third, these conventional systems impose what we call a schema first requirement on the data. That is, you have to describe the structure and schema of your data up front. And the system requires that all of the records that it stores exactly conform to that schema that you provided. In conventional applications this may be fine, but in a big data era where we have data coming from all over the place,

it may not be desirable to require that every record have exactly the same set of properties or describe

its attributes in exactly the same way.

So for example, if I'm storing a collection of json documents about movies, some of those documents

might call the title of the movie title, and some of them might call the title of the movie, name. And I might want the system to be able to store both types of records. So now that we've looked at some of these requirements for these new big data storage systems, what we're going to do in the rest of this module is to survey just a few ideas and systems that have been developed over the past few years.

Unfortunately, in the space of just a few minutes, we don't have time to survey all of the ideas that are out there. There have really been a massive number of these new data storage systems that have been built, HBase, Cassandra, Riak, Neo4j, and so on and so forth.

And each one of them has their own interesting and new set of properties associated with them. So we've provided a list of further readings that you can look at in order to get some information about these different systems. And I encourage you to go read up about them.

What we're going to do in this class is to just to talk about a couple of the sort of interesting properties that are common to many of these systems.

So the first one is this ability and requirement to store non-tabular data, things like json documents.

And second, we're going to talk about this eventual consistency property, this idea that the database system may be able to provide answers even when some part of the system isn't available.

And then finally, we're going to look at a new class of systems that have emerged, that I'm going to refer to as new SQL systems. So new SQL systems are sort of a reaction to this no SQL movement, where what web developer said was, I don't want SQL. I don't want a conventional databases. I don't want these transactional properties. Because I feel that they're either not a good fit for my application or, more frequently, they felt that these conventional systems were simply inefficient.

In the new SQL world, what developers have said is well we can provide a lot of the really nice properties of SQL, things like consistency and a high level query language while still maintaining these efficiency properties that the NoSQL developers say they want.

So I'm going to talk about a system called H-Store that we've developed at MIT that provides these properties.

5.2 Alternative Data Models

In this segment, what we're going to do is understand a little bit about how data can be represented in these new data models or representations that have been proposed for big data.

So first of all, I want to define this term "data model." So a data model is really just a way of storing data in a database system. For example, in conventional database systems that we talked about already,

we referred to the relational data model.

The relational data model is simply this tabular record-oriented structure, where we have a collection of records, all of which conform to the same schema. And we call them relational, because inside of our schema, for example, this table represents some information about courses. In this case, we've got the course identifier, the course name, a reference to the professor that is a professor ID, and some other

information about the course. This model is called relational, because the way that we represent connections between this table and other tables is via these relationship identifiers, in particular, this professor ID column encodes the fact that this table is related to some professor's table, which I haven't

shown here through this ID attribute.

So data models are important to understand and important to think about, because the choice of data model is going to affect the way in which we manipulate and access the data that we're storing inside of our database system.

So by choosing this relational data model, I am going to need some query language that allows me to reference the attributes of this table to manipulate them and also to follow these kinds of relationships, for example, from professor ID to the professor's table.

So in most relational database systems, the query language of choice is SQL. You saw an example of a SQL query earlier in the introduction, but it was very simple. We just did a look-up of my bank account balance.

I want to show you a little bit more sophisticated SQL query now over this professor's table. So this example is a query that finds the professors who teach more than one class. Let's try to look a little bit how at how this query works.

```
SELECT profName FROM classes, profs  WHERE profs.ProfID = classes.ProfID

AND classes.ProfID in

( SELECT ProfID FROM classes HAVING count(*) > 1 GROUP BY ProfID )
```

So in the inner part of this query, what we do is we do a look-up on this classes table to find the classes that are taught by more than one professor, OK?

The result of this inner query is a list of professor IDs. If we want to find the names of those professors, well, we don't have those available to us in the classes table, we need to go to the professors table

and pull them out.

And that's what the outer part of this query does, it connects up the professors table to their classe table and finds the professors that are in this list of classes that are taught by more than one professor. And then it returns their names.

So "Select profName" returns the name of the professor from the professors table, where the professor satisfies this particular inner query that I've shown here. Notice that we're able to do relatively sophisticated computation over the data.

For example, we can do these aggregate operations that find the professors who teach more than one class. So we're able to extract and derive some data sets from the base data, no just look up records that satisfy some particular condition or have some particular key value.

So let's look now at some of the data models that are used in this NoSQL world. So if SQL is the relational query language, then there's a natural question says, well, in the NoSQL world, what's the query language. What is that if we have these non-relational data models, what's the NoSQL query language going to look like? I'm going to talk about two specific NoSQL query languages.

The first one is what I call key value language or ***key value store***.

And the second one is a document store and a ***document query language***.

KeyValue Store

So in a key value store, we're going to represent the data not as typed records of relation with relationships embedded in them, but simply as a key and some uninterpretable string of bytes that is the value.

So for example, my courses database would look like this. I just have the key as the course ID. And then, in this case, I've shown that the information about the course is just some ASCII string, some arbitrary ASCII string. So in the key value store, there's no requirement that the data stored in the values conform to any schema at all. And of course, this leads to a very different programming language than SQL. We can't run SQL queries, at least any very meaningful complex SQL queries,

over at this kind of a representation. Instead, really, all that we can do in this kind of key value representation is to look up records that conform to a particular key, that is to do a "get", or maybe to add or modify some of the records that are stored in the database, and we call those a "put." OK, so this is a very, very simple interface that these key value stores provide.

In addition, these key value stores typically don't provide the strong kinds of transactional properties that relational database systems provide. They really provide, basically, a very, very simple interface, which is the ability to just do these gets and puts.

We can't talk about groups of operations, groups of atomic operations over multiple records. We can only process one operation at a time. So we don't have these transactional guarantees that our conventional relational database systems provided.

Document Data Model

In the new era of big data, of course, there's lots of different data models that one might propose. One that has become popular recently is what I call the document data model or a document store.

So in a document store, it's sort of like a key value store, except for that instead of the values being arbitrary uninterpretable bytes, there's some document, which has some common structure to it.

So for example, you might say, all documents are JSON documents or all documents are HTML. Those would be examples of types of documents that we might want to store. And what this document store model lets us do is impose a little bit more structure on the data.

But notice that in this particular case, we do not have the same strong schema properties that we had in the relational world. For example, here, you can see that there's some courses they don't have professors at all, and some courses they called the professor "PROF", and some courses that call the professor "PROFESSOR."

Typically, these document stores allow us to look up values by key, just like in a key value store. And they may provide some slightly more sophisticated look-up operations as well. Like, we might be able to look up the courses taught by a professor with a specific name.

However, these systems typically also don't provide any kind of multi-record operations. They don't provide these kinds of atomic transactions that relational database systems provide.

They just provide the ability to operate on or manipulate a single record in an expression at a time. OK, so why are people in this new big data era interested in key value stores?

They're much simpler, and in some ways, they provide less functionality than we had in a SQL database. So it might be surprising that in the big data era people are turning to these lower functionality systems. Well, there are a couple of reasons for this.

One main reason is that they're ***relatively easy to implement and program***. They're easy to build, and programmers understand what they provide, and they can understand this get-put interface very effectively.

And this has allowed a very large number of systems to proliferate. So there really are tens or hundreds of these key value stores that have been built and introduced.

In addition, remember that one of the requirements in this big data era is really, really high throughput. We need to be able to process lots and lots of transactions per second. And one of the problems with SQL is that SQL is complicated, the SQL query processors are quite difficult to implement, and sometimes they can be slow. So it's relatively easy to build a key value system that can provide very, very high throughputs. It's much harder to provide a SQL system that can do this.

Third, these key value systems provide the ability to very e***asily distribute data across multiple nodes***.

So one of the challenges of these SQL-based systems is that when you start spreading data across multiple nodes, you have to provide these transactional consistency properties across multiple record stores on multiple computers. And this requires complicated coordination protocols. In contrast, if you don't have the ability to run atomic operation that would manipulate multiple records at a time, you know that you can just spread your keys across a whole bunch of different machines and you have this sort of guarantee that every one of your operations will only need to be processed by a single machine.

So it's pretty easy to spread the data across a whole bunch of machines, but you don't have these nice transactional consistency properties anymore. I want to just leave you with one comment, which is that it's, of course, true that you can represent any data in any of these data models. So data models is not really a discussion about expressivity of representation.

As you've seen, we can represent our course database information in any of these three models-- a document model, a key value model, or a relational model. They're completely equivalent.

The data model really is most relevant because it dictates the query language that we're going to use to manipulate the data.

All right, so what we're going to do next is look at a little bit more detail at the sort of implications of this lack of consistency that some of these key value systems provide.

And then, we'll turn our attention to the problem of whether we can build a SQL-based system that provides performance without giving up these consistency properties.

5.2.1 Eventual Consistency

In this segment, what we're going to do is look at the implications of this Eventual Consistency property of these NoSQL systems. So as you will recall, NoSQL systems often don't provide these transactional properties that our conventional relational database systems provide.

So the reason for this, as we mentioned in the previous segment, is that by eschewing transactions, we can provide much higher throughputs. But this introduces a problem. Because when we don't have transactions, we can't impose any kind of consistency across operations that manipulate multiple records.

So going way back to our example of a bank account and a transfer of money from bank account A to bank account B, in these NoSQL systems, if the system crashes in the middle of that transfer, we can't say anything about whether money has been preserved by our database system.

So this is sort of a problem. And a conventional way that these systems try to deal with this problem is by using ***replication***. So they mask the failure or fault of one of the nodes by replicating the data at the nodes across multiple sites. But in this replicated world, they have a little bit of a problem. Which is that one of the replicas might fail. And now, we need some way to keep these replicas in sync or consistent with each other.

So let's look at replication in a little bit more detail. So first of all, why do we replicate? Well, we use it to ensure availability, as I mentioned. So let's imagine we have a database system that's replicated on three different machines here.

So I've called these Replica 1, Replica 2, and Replica 3.

Now, let's imagine we have our user. So as we all know, on the Internet, nobody knows if you're a dog. So our user is a dog. Let's imagine that our user does a query against one of these tables and gets an answer back. So now, if one of these tables or databases fails, well, the user is free to go to a different database. He can try that database. And if he doesn't get an answer, well, he's free to go to a different database and get the answer. So if these replicas are perfectly in sync, this is great. Because now, we can sort of mask the failure of this first replica.

The challenge comes when we introduce updates into the system. So the question is, what if this user wants to, for example, withdraw some money from a bank account. Well, he issues his update, maybe to all three of the replicas. But say before he can issue his update to the third replica, it crashes. So now, we have a problem, right? What should we do in order to deal with the fact that this replica has gone offline? So we really have three options.

The first option is the option that would happen in the conventional relational database system world. Which is that we would wait for the third replica to come online. The problem here is that that means the system is not available, right In effect, our user can't process his queries in the system anymore in this world.

A second option would be the ***Eventual Consistency*** option. It basically would say, let's just not worry about it. Let's keep going as though the system hadn't crashed. We'll just run the update on the two nodes that we could run them on. And then sometime later, we'll hope that node comes back. And we'll do something to allow that node to catch up. This is what eventual consistency means. Eventually, that node will figure out that it was missing that update. And its state will reflect that update.

The challenge is that now, when that new node comes back, we have a problem. Which is that some subsequent query might go to that new node and might find a stale version of the data that's there.

So how do we deal with this problem? How can we deal with this problem that these new recovering nodes might have stale data? And the user needs to know, or might be concerned with, whether they're reading stale data or not.

And it turns out that there's a really cool idea from Distributed Systems called ***Majority Write/Majority Read*** which we can use to deal with this issue. And that's what we're going to look at next.

5.2.2 Majority Read-Write Protocol

In this module, we're going to talk about this majority read-write protocol that we can use to ensure consistency in a distributed setting, even in the presence of node failures. So in order to understand how majority read-write works, let's look at our simple example of our dog querying the internet again.

So now let's imagine that we have some update that we want to process. What we're going to do is process it very much as before, except for where we're going to annotate every write that we do with a version number. So in this case, imagine that our three replicas begin with some data item X on them that's initialized with version number 0.

Now, when we want to process an update, we increment the version number and send the new value to each of the replicas. So in this case, we set the version number to 1 and send the updates to the nodes.

Now, let's again, imagine that the third node, in this case, third replica, crashes.

So what the majority read-write protocol allows us to do is to continue going forward, as long as we're able to process the write on a majority of nodes. So even if this third update can't be processed, because this notice failed, well, we say, we're going to continue going forward, because, at least, we're able to do the write on a majority.

OK, so now let's see how this fact that we did the right on the majority helps us by looking at what happens when we do an update. So imagine our third replica here now recovers, it comes back up, and it has version 0 of the record X on it. And the other two replicas successfully process the write and they have version 1.

So the cool thing about this majority read-write protocol is that what we'll do is we'll read from a majority of the nodes. In this case, maybe, we read from replicas 2 and 3. And when we read from a majority, we're guaranteed that we will see one of the replicas that has the most recent version of whatever data that we are reading.

OK, so in this particular case, we read from replicas 2 and 3 and we see that replica 2 has version 1, which is the most recent copy of the data, and that's what we're going to end up returning back to the user. And so even though we sent the read to node replica 3, it has a stale version it's send us back to it. We don't give that to the user, we give the most current version back to the user.

The reason that this is the majority read-write protocol works is as follows. We can guarantee that if we've written to a majority, so in this case, we wrote to two of the nodes, and we also read from a majority, so in this case, we read from two of the nodes, we're guaranteed that the read set and the write set overlap, when we have three replicas and at least one node.

OK, and so that means that we're guaranteed that we'll see the most recent version of any update that we sent into the system. So this is a really cool property. And why is it a cool property? Well, it gives us some strong guarantees about the system, while still being able to tolerate the failures of one of the nodes.

So let's just recap where we are and recap why this particular majority read-write protocol is really nice.

So we've discussed three possible ways that we might process updates and reads in a replicated distributed system. So the first one is what conventional relational database system might do, which is what I'll call ***strict consistency***.

Basically, it means in the event of a write, you're going to wait for the failed replica to come back online in order to process the write. And the good thing about this is that it's fully consistent. We're guaranteed that any read we're going to do is going to see the most recent version of the data. And it's nice, because we only have to do read from one node. We're guaranteed that whatever node we read from has the most current version of the data. But obviously, it has this problem that we can't tolerate the failures of any of our replicas.

The second option would be the ***eventually consistent approach***, or purely eventually consistent approach, just keep going, OK?

If you can't process or write at one of the nodes, well, don't worry about it. Just let that node come back up with whatever inconsistent version of the data it has, and then we'll just issue reads to one of the nodes.

The problem is that if you issue the read to the node that failed and came back online, then you might read and old or stale version of the data. So the advantage of this is we only have to read at one node.

And we're able to tolerate the failures of nodes. The disadvantage is we give inconsistent answers.

So the majority read-write is really a nice balance between these two. So what majority read-write allows us to do is to be fully consistent, at least, with respect to a single data element. We can ensure that when we read or write a single data element, we always read a consistent version of it.

I haven't told you yet how to make this work for transactions.

And the topic of doing multi-record updates in a fully consistent way in a distributed system is quite complicated, and we're not going to have time to go into it in this particular module. But it's possible even to do that if we wanted.

So what this majority read-write does is for a single record lets us ensure we read a consistent version of the data. It's also able to tolerate the failure of some nodes. Not as many nodes as in the purely eventually consistent world, but at least as long as we have a majority of the nodes available, and any

non-majority can fail, and we can be consistent.

Of course, the disadvantage of this is well, now, we need to run reads from multiple nodes, and we can just read from one node. And in addition to that, if there's lots of failures, we may not be able to continue operation, we'll be unavailable in the case of lots of failures. OK, so just one little aside, I want to talk about how we actually would bring nodes up to date when they fail.

So this applies either in a purely eventually consistent state or in the majority read-write state. In those cases, a node fails, it comes back online, we have to ensure that that node receives whatever updates it missed while it was crashed. So the simplest way to do this probably would be to simply just copy all the state of the replica from some other replica.

But there's a variety of things we might do. For example, we might keep a log of all the updates processed at any one of the nodes on each of the nodes. When the failed replica comes up, it might go contact one of the other replicas in order to transfer this part of the log.

Or you can imagine some background process, which is continuously comparing the states of all of the replicas and ensuring that they're in sync with each other, meeting this goal of eventual consistency by

background synchronizing all of the nodes together.

OK, so what I've described to you in this segment is this notion of eventual consistency. I talked a little bit about some of the challenges of eventual consistency. So the big challenge is that provides this possibility of inconsistency. But it's big advantage is that it requires very little coordination

between multiple replicas. It allows us to be quite efficient in this distributed context.

This majority read-write protocol is a nice way of making eventual consistency more practical or ensuring that at least we read, always read a consistent version of a single record.

And it has this advantage that allows us to tolerate the failures of some of the nodes. So we don't have to wait for all the nodes in order to process a given write.

5.2.3 H-Store

In this segment, what I'm going to do is give you an overview of the H-Store system, which provides awesome throughput while preserving the transactional properties of conventional relational systems.

So we understand something about how traditional database systems work. In the previous segment, we talked a little bit about how these NoSQL systems work.

H-Store is an example of a NewSQL system. What it does is provide, really, the best of both of these worlds. It provides the transactional properties of these traditional systems with the performance properties of these NoSQL systems.

I'm going to illustrate the performance properties of H-Store using a very simple benchmark which we call the voter benchmark. This is modeled after the Japanese version of American Idol. So on the Japanese version of American Idol, there's a database system that runs that's used to tabulate votes. So when users, or watchers, call in to vote for their favorite artist, the database system does a little increment of the number of votes for each artist. And then at the end, we have to tabulate the votes or report the votes totals for each one of the artist in order to compute the winner. So votes come in, get fed into the database, results come back.

So let's look at the performance of this benchmark in a conventional database system like MySQL or PostgreS. So these are both conventional relational database systems, log the data to disk. Data is predominantly stored on disk. They can provide several thousand transactions per second. Unfortunately for this Japanese version of American Idol, these kinds of throughput simply aren't high enough. You might have millions of votes coming in in a very short period of time when a popular artist performs.

So let's understand why these traditional database systems are a little bit slow. Well, we did some analysis of a traditional database system in order to understand where it spends its time when its processing a query or request. It spends about 30% of its time in what we call buffer pool operations. So buffer pool operations are parts of the query processing that have to do with moving data from disk into the memory of the system. So these conventional databases predominantly store their data on the disk of the computer, and then cache copies of it in memory as needed.

Another 30% of the time is spent in these locking operations. So these locking operations are designed to allow the system to provide transactions that concurrently access the database while still preserving the strong transactional consistency properties. So we use locking to make sure that transactions don't see the intermediate state of other transactions.

For example, when we're transferring money from some bank account A to some bank account B, I might not want to expose the fact that money has been withdrawn from bank account A until I know that that money has been represented in bank account B. So we set locks on data items in order to do that.

Third, another about third of the time in the system, or 30% of the time in the system, is spent in what we call recovery. This is writing data to disk while transactions run in order to allow us to recover.

In particular, traditional systems use what's known as record-level logging. Basically, every time a record gets modified in the database system, we have to do a write to the disk of the system in order to ensure that we'll be able to recover or replay or undo the effect of that write in the event of a system crash.

That leaves only about 12% of the time. And that 12% of the time is really what's spent doing the real work, actually modifying the data structures in order to reflect the effects of transactions, or actually reading the data from the disk of the system.

So the idea in H-Store was to see if we could eliminate these overheads of buffer pool, locking, and recovery, and be left with more of just the real work. And by doing that, our hope is to get at least an order of magnitude speed-up out of the database system.

So to put this in context of what we'd like to do, we've got, on one hand, traditional database systems, which provide very strong guarantees, these transactional guarantees, but provide quite poor performance.

On the other hand, we have these NoSQL systems that provide really good performance, really high scalability, but don't provide transactional guarantees.

These NewSQL systems like H-Store aim to give us the best of both worlds-- both high performance and strong transactional properties. So our question, the real challenge with H-Store was, can we build a database system that can scale up without giving up these transactional properties?

Let's see how we do that.

So the key idea in a store was to say, let's build a new database system from the ground up that's really designed for running these transactional operations. Let's not worry about running, for example, the kinds of analytic applications that I briefly alluded to several lectures ago. Instead, we're going to focus, really, on just processing those little transactions that manipulate a few records.

And I'll talk about a number of assumptions that that allows us to make that simplify the design of the system and let it run really, really fast.

So in a conventional database system, the data has to reside on disk, because the data might be massive. It might be many petabytes big. We never know. And so it predominantly lives on disk.

In H-Store, we say, we're going to try and assume that the database predominantly lives in memory.

We're really going to assume that we can fit all of the database in memory. And the advantage of this is that we can get rid of all these buffer pool operations that we had to do in order to move data to and from disk.

And when the data is in memory, the system is just going to be a lot faster. In a transactional world, this turns out to be a pretty safe assumption, because these transactional databases are meant to reflect the current state of some running system, for example, the number of bank account balances in the world, or the number of products in my product catalog.

And if you think about those real-world things that these transactional databases often represent, they aren't really growing all that fast. We don't have exponential growth in the number of products that we're selling in our database system, typically, or the number bank account balances we have.

In contrast, we have exponential growth in the size of memory in the systems that we have. For example, we can now store almost a terabyte of data in the memory of one computer system. And a terabyte is enough, probably, to store the entire customer and product catalog a company as large as Amazon.com.

The second assumption that we're going to make and way we're going to simplify the design of H-Store is to eliminate concurrent execution. So conventional database systems assume that many transactions are running concurrently.

In H-Store, we're going to do something, actually, that at first may sound like it's not a good idea, which is to only run one transaction at a time. It turns out, as I'll explain in a minute, that this allows us to improve performance considerably. And the gist of the reason why is that it lets us eliminate a part of that pie chart that represented the locking that represented about 30% of the time that we spent when we were running queries.

And finally, to get rid of the last 30% of overhead in our pie chart, we're going to replace this heavyweight recovery mechanism where we did this fine-grained logging of records to the disk with a much more compact logging mechanism, where we only logged the fact that we started executing a particular command rather than logging all the details about the transactions that we executed.

So I'm going to describe those three properties in a little bit more detail in a minute, but I just want to quickly give you a gist of how H-Store actually works So the idea is in H-Store the data is partitioned across into multiple segments.

Each one of these segments has a transaction executor associated with it. So although I've said that we only run a single transaction a time, a more precise statement would be to say that we only run a single transaction at a time per partition in H-Store.

Users of H-Store don't submit SQL text. Instead, they submit the name or identifier of a particular what we call stored procedure that they would like to run. These stored procedures themselves can be expressed in terms of SQL. Here, I've shown an example of some stored procedures. For example, there's a VoteCount stored procedure that computes the count of number of votes that have been submitted for a particular artist in our Japanese American Idol application. And there's an InsertVote operation that adds a new vote to the Vote table.

The advantage of using stored procedures like this is that we're able to optimize the design of H-Store in a number of different ways which I'll explain in a moment by not assuming or not having to process arbitrary SQL that gets sent in by users. We know that the only operations that the user's going to run are these SQL statements that they've predeclared when they set up their H-Store system.

So when a transaction starts running, as it does its processing, it will go ahead and write data out to a command log. This command log is used for recovery purposes in the event of a failure. And as I mentioned previously, this command log only has the identifiers of the stored procedures that need to be run in them. It doesn't have the fine-grained kinds of information about every single update the system did.

Once a transaction has been successfully written to the command log, the result can be sent back to the user. In addition, as the system runs, it take snapshots of the state of the database to disk. These, in addition to the command log, are used for recovery.

In particular, this allows us to recover the state of the database system in the event of a failure without having to replay the entire history of transactions that are in the command log.

So let's see the bottom line about what this has gotten us. So let's go back to our performance results with MySQL and PostgreS, and let's add the H-Store system to this graph. So with just a single core, we can run about 2 and 1/2 times many transactions per second as we could run in PostgreS or MySQL with eight cores. Better yet, with eight cores in this system, we can run about 25 times as many transactions per second as we could run in MySQL or PostgreS.

So this is, I think, a really impressive property. What I'm going to do next is to talk a little bit more about the details of how we actually provide this property in the H-Store system.

In particular, in H-Store there's no use of disk at all during query processing. There's no buffer pool operations that have to be performed to move data from disk into the memory. We just assume that the data in the database is resident in memory.

Furthermore, we're able to eliminate concurrency control, as I'll explain to you in a minute.

And finally, we don't use these locking operations or these heavyweight logging operations that add lots of overhead in these traditional database systems. So what I'm going to do in the rest of this segment is to explain to you how we're able to eliminate these bottlenecks, specifically looking at the no disk, the no concurrency control, and the no locking and no logging pieces of this puzzle.

5.2.3.1 No Disk

Right so let's start with no disk. So the idea in H-Store is that we're going to take our database system and we're going to partition it, split it up into a set of RAM-sized chunks. And then we're going to spread those chunks across multiple machines in a cluster of computers.

So the key insight here that we have in H-Store is that in many, many database applications, particularly the database applications that are supporting what are known as transactional workloads-- that is, things like what your bank runs or what amazon.com runs, the amount of data that has to be stored is not really growing all that fast.

The total number of customers that Amazon has is bounded by the number of people on the planet, right, as is the number of customers that your bank has, as are many other applications. And so this means that the total amount of storage that many database systems have to maintain isn't growing all that dramatically.

However, RAM sizes on modern machines are growing dramatically. So it's now possible to acquire a single machine with one terabyte of RAM in it or a cluster of machines with hundreds of terabytes of RAM in them. And such clusters are capable of storing almost any transactional database system in their memory.

So this whole idea from traditional database systems, that we would store our data primarily on disk and bring some data into memory when we need to operate on it, introduces overheads that we don't really need. Because actually all the data would have fit in memory anyway.

So that's the idea in H-Store. So we built this system that holds all the data in memory.

5.2.3.2 No Concurrency Control

In addition, the data is partitioned in H-Store. So we've taken the data and we split it up into these different chunks and put those different chunks on different machines.

And the reason that we're able to do this is that taking advantage of another observation about these transactional database systems. Which is that they're typically supporting many individual users or storing data about many individual products.

And the kinds of operations that people run on these database systems can be split up very well across different say, customers or products. So for example, in an email database or amazon.com's online shopping database, almost all the operations that have to be performed are to one user's data. It's infrequent that you have to do operations that do updates to multiple users' data.

And we're able to exploit this in order to partition the data across multiple machines in H-Store. So this observation is actually how we're going to be able to eliminate concurrency control and locking in our database system as well.

So the idea is that instead of, as in a traditional database system, where we would run multiple transactions concurrently at the same time, in H-Store we're going to run one transaction at a time on each one of these database partitions. And the reason that this is beneficial to us is that now we can eliminate these overheads that were related to locking and concurrency control in that little pie chart that I showed you.

Because you don't really have to worry about multiple operations simultaneously trying to access data from one partition. And that's what this locking is all about. So by eliminating that locking we can eliminate a significant source of overheads.

Now it's interesting to ask, is this really going to perform well? You might wonder, well why do traditional database systems need this concurrency at all? Why do they need the ability to support multiple users concurrently accessing any piece of data? And there really are two reasons for it.

The first one is that on modern machines you may have multiple processors that could be simultaneously executing parts of a query workload that's running on the system. And the most natural way to take advantage of multiple processors would be to run one query or one user's operations on each of the processors. And so that's a reason you might want concurrency. You can see however though that by using this partitioning approach that I've described we can preserve that kind of

concurrency in our H-Store design. Because now we have one processor allocated to one partition and we run one transaction or query at a time per partition.

The other reason that traditional database systems use concurrency control is to avoid what are known as stalls. Basically, give the machine something else to do when it's waiting for some operation to complete such as a very long latency operation to read some data from disk.

In H-Store, because the data is resident in the memory of the computer and not on disk at all, these stalls essentially go away, which allows us to avoid the need for this particular type of concurrency in H-Store.

5.2.3.3 No Disk-Based logging

All right so the third and final key component of H-Store I want to talk about is how we eliminate what I call fine-grained logging. So you remember in our pie chart our third pie slice that we wanted to eliminate was related to logging.

So logging is used in database systems to allow them to recover from crashes. In H-Store we have two different ways of recovering from crashes that are substantially different than the way the traditional database systems work.

So the first one is that we can recover from replicas. So rather than assuming that there's only one machine that holds the data in the database system, in H-Store we're going to replicate the data across multiple different machines. So if one machine goes down we can copy its data from some other replica. And this allows us to keep the system highly available and to recover from faults.

In addition in H-Store, rather than using the type of logging that traditional database systems use where they had a very fine granularity log every individual write or update that happens to the database, in H-Store we do keep a log but we keep a much less granular log, a log that only contains the list of the transactions that we executed. And this allows us to dramatically reduce the overhead of logging in the

run time of the database system. In order to make this particular kind of recovery with a coarse granularity log work, we need to use what's known as asynchronous background checkpointing.

And so what happens in H-Store is that the disk is actually running some of the time and it's writing out the state of the database system periodically but it's writing it at a relatively infrequent and slow rate. And this allows us to dramatically bound the amount of time that's spent in disk-related operations for recovery purposes in the H-Store system.

All right so here's the bottom line, which is that we're able to go from a picture that looks like this to a picture that looks like this, where almost all of the time in the database system is spent doing this productive work that we've shown in green here and that these other pie slices have been dramatically minimized. And that's how we get that 25 times speed up that we saw in the previous slide.

5.2.3.4 Many optimizations

So in particular I told you about how we're able to eliminate our use of disk, how we're able to eliminate our use of concurrency control, how we're able to eliminate the fine-grained logging that traditional database systems use.

To make this work we've had to do a lot of optimizations inside of H-Store. Here's a list of three papers that we've published on this topic. We'll have these available for you on our website.

But just to give you a flavor of some of the ideas that we've explored, one of them is something called ***speculative execution***. So the idea in speculative execution is that when we do have operations

that need to do updates on multiple partitions, we have a way that we can process those updates very efficiently.

A second idea is something we call ***automatic partitioning***. So this is a way that we can analyze a database system and figure out how to partition it into multiple pieces in a way to support whatever workload users want to run on top of it. This allows us to get these benefits of H-Store partitioning without the user having to explicitly tell us how the database should be partitioned.

And the third piece is ***Multi-Threaded DB On Multi-cores,*** on making the database run really, really efficiently on multicore machines to be able to take advantage of machines with tens or even hundreds of cores on them in order to be able to get very, very high levels of parallelism out of a system like H-Store.

5.2.4 Final conclusions

OK so to recap this entire module on NoSQL versus NewSQL, I've introduced to you, I think, a set of challenges that big data presents for database systems. Really there's a set of new requirements that database systems must provide in order to deal with big data.

The first one is that database systems need to support ***new data models***. And we talked a little bit about things like MongoDB and document stores and key value stores that represent data that's different than the kind of traditional, relational, table-oriented data that conventional database systems store.

The second requirement of the big data database systems is that they provide what we call ***high availability***. They need the ability to be able to be online all the time, even if there is a failure in one of the systems that's running.

And I talked about a couple of techniques, including replication, for providing this kind of high availability. And I talked about some of the trade-offs in how you implement high availability.

There's a notion of ***eventual consistency***, which provides high availability at a lower cost, but doesn't ensure that all of the replicas are in sync with each other.

And I talked about this ***majority read/write protocol***, which ensures that these replicas can be kept in sync. And then finally, these big data database systems have to provide really excellent performance.

And we talked about a couple of different ways that you might implement that performance.

The first one is through a simplified query language, a so-called NoSQL language, that doesn't allow you to express expensive operations that might be very slow in the system.

The second idea is the one that we just talked about in this most recent segment. And that is this idea of using a new kind of database system, one that's really modern and optimized for main-memory databases like H-Store.

6 Big Data Systems

6.1 Security: Nickolai Zeldovich

In this module, I'm going to tell you about how to build secure computer systems that handle large amounts of data.

The reason that computer security is a hard problem is because it's a ***negative goal***. And what I mean by this is that in order to build a secure computer system, you have to make sure there's no way for an adversary to violate your computer system's security policy.

For example, it means making sure there's no way for an adversary to obtain confidential data from your computer system or any way for them to corrupt that data.

And the reason it's so difficult to achieve computer security in many computer systems is because adversaries have so many different avenues of attack to choose from.

And as a defender, as a builder of a secure computer system, you have to defend your system against all of these possible avenues of attack.

To understand what this means, let's take an example of a database storing lots of confidential data.

Typically this looks like the following diagram on the slide.

We have an application server on the left and a database server on the right, storing all of our data on disk. And the application server is going to issue database queries in a language, typically like SQL, over to the database server. And the database or will return results to the application.

So how can an adversary try to steal confidential data from this database server?

Well in fact, there's many ways an adversary could attack here.

One possibility is that the adversary could look on the wire on the network and watch either the queries or the database results going over the network and steal the data that way.

Another alternative that's often exploited by adversaries is to find some sort of a software bug in the database server software and exploit that vulnerability to gain access to the database server that way.

Another option is to focus on the hardware. So maybe if the adversary can get a copy of the data on disk, maybe when the database server is decommissioned or thrown away, or to somehow access the in memory contents of the database server, they can get a copy of the data there.

Yet another route is to attack the humans that manage this database server. Presumably there is some sort of an administrator that has access to this database server and can log in and maintain the database server but also have the privilege to read all the data on disk.

And finally, another concern that you might have is government agencies that might file subpoenas and require a database server to disclose information to law enforcement officials.

So how do we prevent adversaries from compromising our data if there are so many different kinds of things we have to worry about?

One promising approach that can actually take away all of these avenues of attack is to use ***encryption***. What I mean by encryption is imagine placing some sort of a proxy in the middle between the application server and the database. And the proxy is going to store a key of some sort. So whenever queries go from the application server to the database, they'll first be intercepted by this proxy and all the data in those queries will be encrypted before being passed onto the database server. So where we have regular data being sent over here, all of the data being sent to the database server is actually encrypted.

And when the database server provides results back to the proxy, maybe these results will be encrypted but the proxy will be able to decrypt them using its key and send the plain text data back to the application server. The advantage of this approach, if we can somehow pull it off, is that everything on the right of this diagram, as I'm indicating by this shading, is encrypted. That component, the database server and the disk, never store plain text data, never have access to the decryption key. And even if they're compromised, no matter in what way, they will not be able to decrypt our plain text data and leak our confidential information. So this is the power of encryption.

It allows us to address a very broad threat model even if the server happens to be compromised without having to enumerate all of the different avenues of attack. Of course, the problem with the simplistic approach that I've sketched out on this slide is that database servers often need to decrypt the data

in order to process it. If they need to run queries over the database, find aggregate statistics,

select certain records, et cetera.

So in the rest of this module, we're going to look at techniques for how to be able to process queries over encrypted data without giving the database server the decryption key.

So in the rest of this module, as I mentioned, we'll first go over some of the recent theoretical results in the space of execution over encrypted data and look at a scheme fully homomorphic encryption, which is a theoretically promising scheme that's a good thing to know about, but unfortunately is not a practical solution to the problem I posed so far.

Then we'll look at a practical approach that I have been developing with some of my colleagues at MIT called CryptDB, and we'll look at how CryptDB uses specialized encryption schemes to efficiently execute database queries over encrypted data.

I'll give you a sense of what all these schemes look like by illustrating how to construct an order preserving encryption scheme. And then I'll describe a technique called onions of encryption that

allows us to dynamically adjust the level of encryption at run time.

And finally, I'll conclude by showing you all the promising results from a prototype of the CryptDB system that we've built with our colleagues here at MIT.

6.1.1 Fully Homomorphic Encryption

So as we saw in the previous segment, one of the problems in building a computer system that uses encryption to protect confidential data is that computing on the data often requires decrypting it.

The cryptographic community has studied this problem of fully homomorphic encryption for a long time now. What fully homomorphic encryption means is being able to compute over encrypted data without having to decrypt it, which sounds like exactly the kind of thing we're looking for. And in fact there's been, recently, a breakthrough result by Craig Gentry in 2009 that shows us that it is in fact possible to construct such a thing.

To give you a sense of what fully homomorphic encryption provides, consider the setting where you have a client that has some sensitive data and a server that wants to execute a certain function over

that sensitive data. So if the client has sensitive data x, the client can encrypt it to produce an encryption E[x], which is the ciphertext corresponding to that data.

Now if the client sends the ciphertext over to the server, the server, without having access to the decryption key, cannot actually figure out what the data is.

However, if the server has a certain function f that it wants to apply to the data, then the server can actually use this fully homomorphic property of the encryption scheme to come up with the encryption of the function f applied to x without ever decrypting x or learning anything about it. It sounds quite magical, and I'm not going to dive into the details of this scheme, but what it now allows us to do is for the server to return the results back to the client.

And the client, initially, of course, gets the encryption of f(x) from the server. But since the client has access to the decryption key, the client can now decrypt this value. So now that the client has the encrypted value of f(x), it can use its decryption key to decrypt it and get the value of f of x in plain text form.

So this sounds like a hugely powerful construction. We're able to take our data, send it to a server we don't trust, and allow the server to run functions of our data, potentially running database queries.

Now, this is a promising construction, and there has been a lot of excitement in the crypto community about it. But one problem is that running computations over encrypted data using fully homomorphic encryption is quite slow in constant terms.

So for example, applying function f to the encrypted data on the right of the slide takes roughly 10 to the 9 times slower, or takes 10 to the 9 times more execution time than applying the function f to regular, plain text versions of x.

So that's quite unfortunate, but the bigger problem that I want to illustrate to you is that the model of fully homomorphic encryption is itself not very well suited to the problem of running queries over

an encrypted database.

So let's get back to the scenario we saw earlier where we have an application server talking to a database server through a proxy, which has access to our key. And imagine the proxy sends our database query to the database server as some kind of a function f that we'll try to evaluate over the database using fully homomorphic encryption.

Now, if we use fully homomorphic encryption, then the entire database shown on the slide is going to be encrypted as far as the database server is concerned, so it will not be able to see inside of it. And the database server will have to take this entire database and run it through function f using fully homomorphic encryption.

Now, this already is hugely inefficient if you're paying attention, because it means that the entire database, regardless of what the query looking for, has to be fed through this function. So even if the query just wanted to find Alice's salary, it has to read the entire database from disk and feed it through the function f.

Worse yet, fully homomorphic encryption doesn't allow the server to learn anything about the result, not even the size of the result. So you have to, ahead of time, guess how big your result might be and

pre-allocate that size as the output of function f, even if the result contains no answers whatsoever.

So fully homomorphic encryption requires the database server to learn absolutely nothing about the data, which is at odds with achieving good performance, even leaving alone the infeasibility of current fully homomorphic encryption constructions.

So in the next segment, we're going to look at a more efficient approach to running database queries over encrypted data.

6.1.2 Crypt-DB approach

In this module, we're going to look at how to more efficiently execute database queries over an encrypted database. So in order to run database queries efficiently over encrypted datas, we want to be able to achieve comparable performance to running these queries through a plain text database. And this means that we have to execute queries fairly similarly to how a traditional database server would do it.

For example, this means that we have to be able to use efficient data structures to look up the data being accessed on the database server. So this means constructing data structures like B+ trees over

encrypted data and hash tables so the server can quickly figure out which rows are being accessed.

It also means that the server has to be able to figure out which row matches or doesn't match the query because we want the server to return only the matching rows to the client instead of returning the entire

database as a worst case scenario. So how do we do this in practice?

Well the approach we take in our system called CryptDB is to trade off some of the generality and security that we saw possible in the fully homomorphic encryption approach to achieve better performance. In particular what we're going to do is we're going to expose the internal structure of a database to the server.

This means we'll allow the server to see different rows in the database and different columns. And what we're going to encrypt is not the entire table of data but in fact individual cell values.

And when we're encrypting things in individual cell values, we're going to use a range of different encryption schemes that will allow us to perform different kinds of functionalities over that encrypted data. So this necessarily requires us to reveal some of the information in our confidential data to the server because the server must be able to decide if a row being considered in the database matches a query or not because it needs to determine whether to return it or not as the results said.

However, as we'll see, the amount of confidential information lost in this way is going to be relatively low and we're going to be able to achieve very good performance by taking this approach. So let's take a look at how we might actually do this in a database server.

Consider this following scenario. We have an application and a proxy server which you both saw in the

previous segments. And on the right, we have our untrusted database server storing an encrypted copy of a table of employees in some hypothetical company which contains the rank, name, and their salary.

And here all the individual cell values stored in the database server are encrypted, indicated by this orange color. And I'm demonstrating here on the slide that the salary values are encrypted using a randomized encryption which provides the highest level of security possible-- semantics security.

And just for your information here, on the very right of the slide, I'm showing the actual salary values being encrypted in this table-- 60, 100, et cetera. So suppose that an application issues a query to the proxy asking for the set of all employees whose salary is equaled to 100.

Now if the database really is encrypted using randomized encryption for the salary column, then there's not much we can do because theoretically semantic security proves that there's nothing a database server can learn about those salary values.

However, if we use a different encryption scheme, we might be in luck. And in particular, let's consider using a deterministic encryption scheme.

Deterministic encryption is just what it sounds like. It's a deterministic function of the plain text data to a cypher text. So this means if we encrypt the same value twice, we'll get the same result. And you can see this on the table. The two rows containing the salary of 100 have the same encrypted cypher

text value.

So if the database is actually encrypted using deterministic encryption, we can actually run this query on the encrypted database without getting access to the plain text data. What this requires us to do is for the proxy to rewrite this query. It's still going to ask the server to select all of the rows in the database

from this anonymized table name, table one-- which corresponds to our table of employees. And the proxy server is going to ask the untrusted database to filter the data in a different way.

It can't just ask for salary equals 100. However, it can ask for the encr ypted column storing the encrypted value ofthe salary to be equal to the encrypted deterministic value of 100,

which in our case, corresponds to x5a8c34.

And now, this database server can find that there is in fact two matching rows for this query, namely the second and the fourth row here which really are the rows containing salary equals to 100 and the database server can now return these encrypted rows to the proxy. So now that the proxy server has received the encrypted values of the resulting rows, it can use its encryption key to take these encrypted results and decrypt them before passing it onto the application and now the application can continue processing them as if there was no encryption going on at all.

So this deterministic encryption scheme seems to work well for equality queries but what if the application is asking for all the employees where the salary is greater or equal to 100 instead of equal to 100.

We can see that using deterministic encryption is not going to work here because the person with a salary of 60 has an encrypted salary value that is actually very large here.

But instead what we can do is use a different kind of encryption scheme called ***order-preserving encryption***. Order-preserving encryption has this fascinating property that if x is greater than y then it means that the encryption of x will also be greater than the encryption of y.

In the next module, I'm going to tell you a little bit about how to construct such a scheme but for now just assume it exists. If we actually manage to encrypt our salary value using an order-preserving

encryption scheme, now we can process our query just as before. The proxy server can still ask the database to select all of the rows from that anonymized table name of employees called table one. And the condition for selecting the rows is going to be where the encrypted column storing the salary field is greater or equal to the order preserving encryption of the value 100, which in this case, happens to be

638e54 and here the database server can again find all the right matching rows which are rows 2, 3, and 4, return them through the proxy server in their encrypted form and finally allowing the proxy server to again decrypt them and pass them up to the application using its description key.

So we've seen how we can execute a number of different kinds of queries-- equality, greater or equal to, et. cetera using several different encryption schemes.

So in the next module, let's take a look at how we're exactly going to construct such an order-preserving encryption scheme as just one example.

6.1.3 Order Preserving Encryption

Let's talk about how we can construct an order-preserving encryption scheme, which is what allows us to run order and comparison queries over encrypted data. Recall that order-preserving encryption has two goals.

One is the functionality goal that I already showed you in the previous section. Namely, we want to make sure our encryption scheme is ***order-preserving***. If we take two integers where x is greater than y, we want to make sure that the encryption of x, the cypher text value of the encrypted x, is going to be greater than the cypher text value of the encrypted y. This is a rather strange property, admittedly, for an encryption scheme, because typically you don't want to leak any information at all about the data in

this encryption. But as we saw previously, having such a scheme is going to enable us to very efficiently run queries over an encrypted database.

Now in addition to providing this functionality of being order-preserving, we want our encryption scheme to also provide ***security***. And what this means for an order-preserving encryption scheme is

indistinguishability. This means that if we take the encryptions of three different values, or any number of different values, for that matter, let's say encryption of 1, encryption of 3, and then encryption of 2, then these should be indistinguishable from an encryption of any other three values that have the same order. So let's say it's the encryption or the value of 5, the encryption of some other value that's larger than 5. So maybe a very large number like a 1,000,000 or 100,000, and some other value in-between them, like 24.

So an adversary presented with the set of encrypted values on the left should not be able to distinguish it from the set of encrypted values on the right, because they both have the same order.

So how do we construct such an encryption scheme that allows us to learn the order of elements, but not distinguish what the elements really are? I'm not going to go into the full details of how to do so, but I will give you some intuition about how to construct such a scheme. And this is a construction we call ***mutable order-preserving encryption***, or ***mOPE***.

So I'll illustrate this by example. And the way this encryption scheme is going to work is going to be different from a traditional encryption scheme. One thing that's going to be different about it is that it's not going to be a pure algorithm that a client can execute on its own. It's actually going to be an interaction between a client and the server.

And the way this interaction's going to go is the client is going to be building up a balanced binary search tree at the server containing the encryptions of all the items it's ever encrypted. So suppose the client is going to try to encrypt the four values shown on

the left of this slide.

The first thing the client is going to do when encrypts 20 is going to start building up this binary search tree. There's nothing yet at the server, so it'll simply construct a single node containing the deterministic encryption of the value 20, of the value being encrypted.

So it's the complete tree of the server. And the deterministic encryption you can imagine as being AES encryption using a fixed initialization vector and the client's encryption key. If you don't know those terms, don't worry about it. So the client encrypted one value. What happens if it wants to encrypt the value 18? Well, it wants to insert it into the binary search tree through the server, and 18 is less than 20, so this actually goes to the left of the existing node. So we're going to place the encryption of 18 in the second node in the tree located as follows.

And in fact, the server can't figure out where to place this node on its own. The server can't tell whether a deterministic encryption of 20 is greater or less than the deterministic encryption of 18. So the client has to interactively help the server walk this tree from the root to the new position of the node being inserted. And of course, now if we encrypt the value 100, we're going to place the deterministic encryption of 100 to the right of this tree.

And finally, when we increase the value 3, that node is going to go to the left, as shown on the slide here. So how does this help us encrypt these values in an order fashion? Well, what we're going to do is we're going to make the encrypted cypher text of a particular value the path down the tree towards that value. So the cypher text of 20 is just going to be the empty path. The cypher text of 18 is going to be the cypher text "go left." The cypher text of 3 is going to be "go left," and then "go left" again. And the cypher text of 100 is going to be "go right."

So these strings look sort of ordered, but if we want a binary in your presentation, it's actually relatively easy to do so. We'll just call L 0 and R 1, and then we'll encode them in a suitable fashion. So we'll take the empty string for the root node, append the value 1 and a bunch of 0s at the end, and we'll do the same with every other node. So L here is 0, and we'll append 1 and a couple of 0s. And here, it's 00101. And on the right, the value R is 1, followed by 1 and a couple of 0s. So in this case, we can see that these binary encodings of the paths down the tree actually form the right order. So the order-preserving encryption of 3 is less than all the other order-preserving encryptions, and so on. So it seems like a nice property to have.

One problem with order-preserving encryption is that, of course, we sometimes need to rebalance this binary search tree. If we insert items in an increasing order, the tree will be very lopsided to the right.

Then in a typical binary search tree, the way you solve this is by doing tree rotations. So in this case, if we want to rotate our tree, we might perform a rotation like this, which places the 18 node over here, and places the 3 node up there where the 18 node was. So after this rotation, the problem is that the encodings of the nodes might actually change.

So this is the other strange property of our encryption scheme, the cypher text of a particular value can change over time. But luckily, in this database setting in which we are trying to construct the scheme,

it's actually OK for us to change the encryption of a value, because we have the complete database available to us at the server. So if we need to perform this rotation, which is going to be relatively infrequent, we can simply ask the database server to update the contents of its database, changing any old encryptions of 18, which used to be 0100, to the new encryption of 18, which is going to be 0110.

So as we can see, at least in this intuitive sketch of anorder-preserving encryption scheme, it's quite straightforward to construct such things.

And they provide reasonably good performance, because we're using fairly standard building blocks, like binary search trees and symmetric encryption schemes.

So in the next topic, we're going to try to put all of these different encryption schemes together, and see how we can build a complete system out of them.

6.1.4 Multiple Encryption Schemes

So as you saw previously, we can construct a different kinds of encryption schemes to execute different operations.

For example, order preserving encryption allows us to execute ordering and sorting operations.

But to run different kinds of database queries, we're going to need to use multiple encryption schemes to handle different kinds of queries.

For example, to simply store data, we want to use semantic security, like-- if you're familiar with these terms, AES and CBC mode of encryption. And this particular encryption scheme is not going to allow us to perform any computation at all on the encrypted data, but it provides a very high level of confidentiality.

Now there's a different kind of encryption scheme called homomorphic encryption, such as Paillier or ElGamal or RC, which allow us to perform different kinds of operations. In particular, we can add numbers together, or multiply them-- not both, unfortunately. That requires fully homomorphic encryption. And these singly homomorphic encryptions that allow us to either add or multiply numbers are reasonably efficient.

Now there's also other kinds of encryption schemes out there, such as searchable encryption, which allows us to perform keyword search over encrypted data, looking for a particular word among a large text field.

And you already saw deterministic encryption. That allows us to perform equality kind of checks on encrypted data. This could be implemented using AES and CMC mode, if you're familiar with these terms. And there's other encryption schemes, as well, that we've come up with. For example, a scheme for joining multiple tables with each other, and a order preserving encryption scheme that you saw in the previous segment, which allows us to do things like sort values, compare them, find the max, and so on.

So this is a nice collection of schemes that allow us to do a reasonably interesting set of different operations. One limitation to this approach of using different encryption schemes, of course, is that we cannot perform computations that require two different encryption schemes to be used in a single expression. So what I mean by this, of course, we can perform queries like a equals b AND c is greater than d. That's easy enough to do.

We'll just deterministic encryption to check for a equals b, and order preserving encryption to check for c greater than d. That works just fine. The thing that's difficult to do in this multiple encryption kind of approach is things like a plus b is greater than c, because computing the value of a plus b requires us to use Paillier encryption, which is a homomorphic scheme.

But then the resulting cypher text is not order-preserving, and we cannot compare it to the encryption value of c. So that's one limitation, but it turns out to be not a big deal for most database queries.

The thing that is kind of worrisome here is that, as you might have noticed, functionality sort of runs down the slide. So schemes lower in this slide are actually more functional. So we'd like to be down there, to be able to run more different kinds of queries.

On the other hand, security actually runs up the slide. So the most secure schemes are semantic security and homomorphic schemes, while the least secure schemes that reveal a lot of information are things like order-preserving encryption and deterministic encryption.

So how do we revolve this tension between functionality and security in these multiple encryption schemes? The problem is that we need to somehow encrypt the data when we send it to the database server. And as we saw, the encryption scheme we should be using depends on the query that we're going to be able to run later on that data. And the issue is that it many database settings, you don't actually know the queries the application's going to issue over your database ahead of time. It might depend on what kind of analysis your application or user wants to perform at a later point in time.

So one approach that you might take in order to resolve this seeming conflict is to simply encrypt all the data with all the encryption schemes. So as we can see here, we might take a column storing the rank of an employee, and in encrypt it with all six encryption schemes that I showed you on the previous slide. This will allow us, of course, to run any possible query that our schemes support, because we have all them available on the server.

But it's kind of passable from a security standpoint, because our weakest encryption schemes, order-preserving encryption, for example, on the right here, is going to be present for every value.

So for example, if you're storing credit card numbers or social security numbers, it means every day, the server will know the order of your social security numbers, which, in practice, you probably don't want to reveal to an adversary.

So how can we resolve this problem of not knowing the queries ahead of time, and yet wanting to support all these different kinds of queries? One answer is this construction we've come up with called onions of encryption.

So the way these onions of encryption is going to work is by stacking encryption schemes on top of each other, and allowing us to reveal different encryption schemes at run time depending on what queries come in. Suppose we have some integer value x, let's say, equals to 5. The way we're going to construct our onion is we'll first take this value, and encrypt it using order-preserving encryption with some particular key, let's say k1.

So now that we have this order-preserving cypher text, we might be able to run order queries, but we don't necessarily want to reveal it to the database server yet. What we're going to do is encrypt that value again. We're going to encrypt it with a deterministic encryption scheme with a different key, let's say k2, and the resulting cypher text will be a deterministic encryption that allows us to run order queries.

And finally, because we might not want to reveal the deterministic cypher text either, we're going to take that value, and encrypt it again with a randomized semantically secure encryption scheme with yet a different key, k3.

Finally, we have this resulting cypher text that is semantically secure, so we can give it to the database server and not worry about it learning deterministic encryption values or order-preserving encryption values from it.

However, if at a later time a query comes in, and requires us to reveal the deterministic encryption of this value x to the server, what the client can do is to give the server access to this key k3 for the randomized encryption scheme. Once the server has access to this key k3, it can decrypt the semantically secure layer of our onion, and reveal what was encrypted with that randomized encryption scheme.

And what that is, is just the deterministic encryption that we'd had before. And now, the server can run equality queries over our data. Note that because we have a different key for every level of the onion, revealing the key k3 doesn't allow the server to get the plain text data. It simply allows it to strip off this one level of the onion.

And similarly, if we want to now sort on this value, we need to reveal the order-preserving encryption value to the server. And we can do this by revealing key k2, which allows the server to strip away yet another level of the onion for this depth level, revealing the order-preserving encryption of x. So this onion of encryption construction allows us to achieve good confidentiality guarantees without having to know a priori the kinds of encryption schemes that we're going to need, or the kinds of queries that will be executed on the data. And in fact, the way to think of these confidentiality guarantees is that, first and foremost, our system will never reveal the plain text data to the server, because we never strip away the last level of the onion.

But in addition to this, the server might learn some information from the different kinds of encryption schemes that the server has access to. And the way to think of it is the database queries translate into different kinds of encryption schemes that will be revealed to the server, and that translates into the kind of information that's leaked to the server. So for example, if we have a database query that requires us to perform an equality check over the database, this means that we'll have to reveal the deterministic encryption of the data to the server. And this means what we'll reveal is duplicates in data encrypted with that encryption scheme. And the guarantee that our system can provide is that it will only reveal the most secure scheme necessary for supporting a particular query. So if we run an equality query, our system will only reveal the deterministic encryption, and not the order-preserving encryption.

And finally, in practice, what a system administrator might want to do is use thresholds to limit the leakage from such a system. For example, if an administrator knows that a particular value is highly sensitive, it might be better to put a threshold requiring that the data will never been encrypted with order-preserving encryption, and, as a result, return an error if an application ever tries to run sort by a

sensitive field, like a credit card number. So this gives you some kind of a sense about practical techniques that you might use to run encrypted queries over a database without revealing the description key.

And next, we're going to look at some results from our preliminary prototype of this Scribd BB system to see how well it actually works in practice.

6.1.5 CryptDB Results

Now that you've seen how we can execute database queries over encrypted data, let's see how well it works in practice.

So together with several of my colleagues, we have built a prototype implementation of the system called CryptDB that allows us to run database queries of our encrypted data.

The way it works is, as I've been demonstrating in my slide so far, we insert a proxy server between an application and an unmodified database server. And this proxy server, it consists in our prototype of about 26,000 lines of C++ code, so it's moderately complicated.

But the cool thing about it is that the application that is using the proxy server is completely unmodified. It issues standard SQL queries to the proxy server, as if nothing else is going on, as if it's talking to regular database server. And in fact, the proxy server translates these queries into other SQL statements issued to another unmodified database server.

In fact, our database server doesn't really know that it's running over encrypted data. The only change to it is that we had to add a couple of extra functions to the database server, indicated on the right of the slide as user defined functions, or UDFs, that allow the database server to perform some cryptographic operations, like stripping off levels of onions, or performing a couple other cryptographic things that you might need to do in CryptDB. So on top of this product implementation, we wanted to answer three main questions about CryptDB.

First and foremost, we wanted to know, can this approach actually support real queries from real applications?

Second, we wanted to figure out what resulting levels of confidentiality can you actually achieve in such a system?

And finally, what is the cost of CryptDB in terms of performance?

To answer the first question, what we did was we ported several popular applications that are typically database backed to run on top of CryptDB database. These include three popular applications, phpBB, which is a widely used bulletin board, used for many websites. HotCRP, which is a conference reviewing piece of software. And the MIT graduate admission system, grad-apply.

For these applications, we manually examined their database contents, and we chose to encrypt the most sensitive columns, which you can see in this second column of the slide. So there's a number of different columns in these applications storing sensitive data that CryptDB will try to encrypt for us. We also took two somewhat more synthetic applications, or workloads. We took the standard TPC-C database benchmark, for which we tried to encrypt all of the columns, all 92 of them. And we also took a large trace of queries from a MySQL database server at MIT, called sql.mit.edu, and it contains very many, thousands of different databases used by students, comprising over 100,000 different columns.

And we tried to see, what would it take if CryptDB tried to encrypt all of those columns using the techniques you've seen so far? So the results are actually quite promising. So in all of the first four cases, CryptDB is able to properly execute database queries over all of the encrypted columns.

So we can execute the entire application on encrypted data, on the sensitive parts, at least, for the first three applications without revealing too much information to the server. Now, in this trace of queries from the MIT database server, we see that there is about 1,000 different columns for which CryptDB is unable to run queries over encrypted data, and would require that data to be either stored in plain text on the database, or for those queries to be changed to run on the client instead.

Now, if you look at the actual queries that result in that number, you might not be surprised. We saw queries like selecting one over the natural logarithm of some number plus 1.2, or selecting the rows where the sine of some value plus the value pi matches some criteria.

These are relatively complex queries that our simple encryption schemes cannot handle. But you can see that in most cases, CryptDB is able to support important applications over encrypted data. In terms of the resulting confidentiality level, we looked at the onion levels that are available in the database server after our particular database workload runs. So the most interesting column here is the RND level, which means the number of columns that remain encrypted with semantic security.

You can see that actually, most of the columns fall into this category. So for example, 80,000 out of roughly 120,000 columns in our sql.mit.edu trace remain encrypted with Symantec security, providing no information whatsoever to any attacker that compromises the database. Now, a somewhat smaller number of columns are encrypted with deterministic encryption, which, in many cases, still remains pretty secure. And some number, of course, require order preserving encryption, which as we saw, leaks quite a bit of information.

For the first four applications, we manually examined these columns, just to see what's going on. Why do we have to reveal order preserving encryption values? And we saw that in the small number of columns shown on the right, it was mostly some of the less sensitive fields, like time stamps, message IDs, sequence counters, etc., that perhaps are not as sensitive as the raw contents of sensitive messages, salary information, etc.

Now, we weren't able to examine the 13,000 order preserving encrypted columns in the sql.mit.edu trace, purely because of its size, but our hope is that similar conclusions might hold, that they are some of the less sensitive data in those applications as well.

And finally, in terms of performance, we ran the TPC-C database benchmark, one of the five evaluation applications I showed on the previous slide, on a multicore database server that we had in our research group. And we can actually compare the performance of CryptDB to that of an

unmodified MySQL database server. And we can see that, on average, the overhead is actually quite modest, and it amounts to roughly 26% through production at the database server, compared to running the same database queries on a plain text database server. So for many applications that are security sensitive, it might actually make sense to use these techniques to protect against server compromises.

So to sum up, you've seen that CryptDB's approach can actually support a wide range of SQL queries in practice from real applications, that this approach provides a high level of confidentiality for the data encrypts, and the performance overheads are reasonably modest and might make sense for a number of applications in practice.

6.1.6 Summary

Now that you've seen how you can actually run database queries over encrypted data. I just want to wrap-up and summarizes what I've shown you in this module.

I've told you that security is a difficult goal to achieve in practice, in large part because it's a negative goal that requires you, as a system designer, to think of all kinds of attacks that an adversary might try to mount against your system and try to defend it against all these possible attack vectors.

A powerful technique to prevent a broad range of attacks is to encrypt data. Encryption guarantees that unless the adversary can get their hands on the encryption key of your data, they will not be able to decrypt it, and your confidential data will be safe.

Now, one of the challenges in doing this is that many systems require computing on data, and if the data is encrypted, you have to somehow figure out how we can compute on the encrypted data without divulging the decryption key.

I've shown you one practical approach to do this called CryptDB, which allows us to run database queries on encrypted data by using a combination of specialized encryption schemes, like deterministic encryption, order-preserving encryption, together with this construction called onions of encryption, that allows us to dynamically adjust the encryption scheme available to the server, on demand, based on the kinds of queries the application or the user wants to issue.

And in fact, CryptDB is just one of many exciting techniques that you can use to run computations over encrypted data. If you have more complex queries than the simple examples I've shown you in this talk, you can actually use partitioning to split the execution of a query between an untrusted database server and a trusted proxy server that can handle some of the more complex part of your query. More excitingly, you can also try to push all the encryption and decryption all the way into the user's web browser, so that there's no trusted server-side component. Not even the application server itself is ever trusted with the plain text data. And all the plain text data only ever exists in the user's browser.

So together, all of these techniques should help you protect confidential data in a large data system from a wide range of attacks.

6.2 Multi-Core Scalability: Nickolai Zeldovich

You probably know that many computers these days consist of multiple cores running in a single system, all with access to a single shared memory. And in this topic, we'll look at, how do you make sure that this computer system performs well?

So what do I mean by performing well?

In particular, what we're going to talk about is ***scalability***. And by scalability, I mean that if we have a computer that has n cores, ideally, this computers should do n times as much work in the same amount of time as a computer with a single core.

Now, this notion of scalability is often a convenient way to think about achieving good performance on a multi-core system, because as you add more cores, a scalable system will achieve more and more total performance or throughput or work.

Now, they're not exactly the same. It's easy to have a system that scales well but has poor performances. It just does lots of useless work on every core. It's oftentimes that high performance system isn't the most scalable, either.

But there's quite a bit of correlation between these two notions, so we'll stick with the goal of trying to achieve good scalability on a multi-core system for the rest of this topic.

So one reason why a computer system might not achieve good scalability is because of the presence or ***serial sections*** in the overall system. One example of a serial section you might be already familiar with is a typical ***lock*** in an application that ensures that only one piece of code is executing at a time.

Another example of a serial section you might encounter in a computer system is ***accessing shared memory***. So if multiple cores are accessing the same data structure in memory, hardware must ensure that only one core can perform each individual memory access at a time. And thus this serializes the instructions issued by multiple cores, limiting the scalability of the overall application.

And finally, there are hardware resources that might pose a serial section. For example, if all of your programs are accessing a single DRAM interface in your computer system or a single network interface card, then hardware can only allow a single application to access the DRAM or NIC at a given time.

And this again, forms a serial section that bottlenecks your application scalability.

Now, how significant is this notion of a serial section to the overall application performance?

One rough way to think of this problem is to imagine that your application is split into two sections, one that runs in parallel and spends time, p, in the apparel section of your code, and one part of your application is purely serial and spends time, s.

Now, if you roughly imagine how long such a program might run on a machine with n cores, we can take the parallel time, p, and divide it by the number of cores, n. So as we have more cores, the parallel section will run faster and faster. But the serial section will still executive serially, regardless of how many cores we have. And as a result, as we have more and more cores, so as n goes to infinity, what this means is that the total runtime over application is going to approach s, and will not improve beyond the length of our serial section, no matter how many hardware resources we throw at this problem.

So we understand that serial sections are a problem for scalability. But do we really need these serial sections?

One approach to achieving good scalability on a multi-core system is to simply partition your system into n independent cores. Think of them as just n independent computers, working in parallel with one another.

Now, this is a good approach if you can actually partition your data set, your application, your workload across these multiple partitions, because these partitions will not share much with each other and will achieve pretty good scalability. And in fact, this is what you have to do if your problem or data set is so large that it cannot fit into a single computer, no matter how many cores you can buy from Intel. Now, the problem is that partitioning requires you to actually choose a specific partitioning priority.

And it might be hard to break up your single monolithic problem into n independent partitions.

Another disadvantage of this partitioning approach is that once you've broken up your problem into n independent chunks, you might need to load balance these partitions with each other, if it happens that one partition has too much work and another one doesn't have enough.

So for the rest of this topic, we'll avoid this partitioning approach and look at situations where you must have some amount of sharing between the cores and consequently cannot purely partition your data set between the cores of your system.

So for the rest of this topic, I will first tell you about how the hardware works at a low-level, because that's critical to understand how to achieve good performance and what kinds of operations scale well and what kinds of operations do not.

Then we'll look at a couple of case studies to understand what does it take to build a scalable application.

First, we're going to look at how cache coherence impacts scalability by examining a simple implementation of a lock, that is, a basic primitive used by almost every concurrent application. We'll look at how a lock can collapse in terms of performance when you have more and more cores.

And we'll look at techniques that help you avoid performance collapse in a lock implementation.

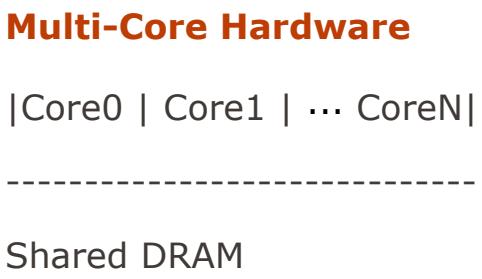
Then we're going to look at techniques to improve scalability of an overall system by avoiding locks altogether.

We'll look at an example application of a stack data structure and look how we can actually perform lock-free reads, reads that don't actually require us to take any lock at all.

And then we'll generalize this approach into a technique widely known as ***Read-Copy-Update***, or RCU, that allows us to perform lock-free reads in almost any application.

6.2.1 Cache Coherence

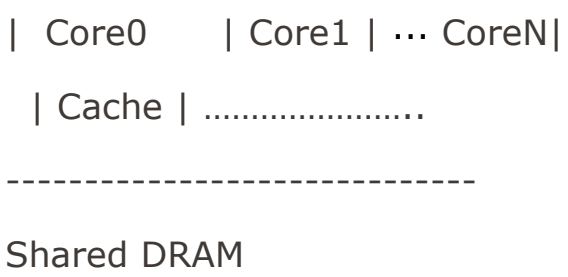
So in this segment, we're going to now start talking about how to achieve scalability for operations in hardware by looking at how hardware actually implements memory operations from different cores.



So at a very high level what a multi-core computer looks like is a bunch of cores, here shown numbered zero through N, accessing a single shared DRAM subsystem.

What this means is data stored in this shared memory can be accessed by any of the cores in our system. And the cores don't have to a priori partition the data to decide which core will be able to access which data in memory.

Now in practice, one problem with accessing DRAM directly is that it's quite slow. In a typical system, accessing DRAM will take easily a couple of hundred clock cycles, which is quite expensive. Instead, what almost every computer system does is introduce what's called a cache in front between the core and the shared DRAM system.



So in a more multi-core machine that you might have today you're going to find a cache sitting in front of every core. And whenever a core needs to access some location in memory it will first check if that location is already stored in its private cache. And only if it's not stored in the private cache will it consult the shared DRAM.

Now, the advantage of this approach is that each of these cores can now independently and very quickly look up things in its cache. It takes an order of a few cycles to access memory locations in the cache in some of the machines you might find today. The problem with caching, of course, is that these independent caches provide a consistency problem for us.

Imagine if core zero first wrote the value five to some memory location, and then core one wrote the value eight to the same memory location. Now, these writes get stored in the private caches of each respective core. So if now core zero decides to go back and read the same memory location it might observe the value five stored in its local cache, even though a different cache has the more recent value eight stored in it.

So this brings up the question of what is even the goal that we would want to achieve in terms of what is the correct answer to give to any particular core performing a memory operation?

Now, the ideal such memory model is typically called ***sequential consistency***. And what this means is that the execution of each individual core and all the cores put together is consistent. It's as if all of the cores executed with direct access to the shared DRAM interface. It's as if the cache is simply not visible.

And we see that in our simplistic version of the cache here, this is not the case, because you can easily get the wrong answer, because the cache remembers your last value and not someone else's last value.

So how do we solve this problem?

How do we achieve sequential consistency in such a multi-core design with many private caches? The typical technique used to achieve sequential consistency is called a ***cache coherence protocol***.

And what a cache coherence protocol does is it stores some additional information in each core's private cache to keep track of what else other cores might be caching or might not be caching about the same memory location.

So the way cache coherence protocol works is that it splits up the total amount of system memory into units called cache lines, which are typically 64 bytes of memory adjacent to one another. And each entry in a cache refers to a particular cache line. So suppose here, core zero accesses memory location 0x1000, as in our previous example, and it wrote the data value 5 to this memory location.

Now, in addition to the address and data of the value being cached, each core's cache is also going to store a state that is going to help us maintain consistency between caches. And these states, for our simplified version of a cache coherence protocol, are going to be one of three things.

It's either going to be modified, meaning that this core's cache contains a different version of the cache line's data than the shared DRAM, and no other core in the system is allowed to have any copy of this data.

Another state might be shared, which means that other cores might have a copy of this data, but they're not allowed to modify it, meaning that all the cores have the same data and it's also consistent with each other.

And finally, a cache line might be in an invalid state, meaning that the data is simply garbage. It should not be used without first somehow validating it.

So imagine that we're somehow in a state where both cores read the same value five from memory. This means that all the cores would be a shared state. So core one would have pretty much the same contents in its cache. The address would be 0x1000.

The data would also be five. And the state of core one's cache line for that address is also going to be shared. Now suppose, as in our previous example, core one tries to write to the address, to that memory location. Well, before it can modify its cache, it must ensure that it's in a modified state.

And in order to do that, it has to make sure that no one else has it in any other state. So what core one must do is it must actually send a message from its cache to the cache or any other core holding this same cache line to invalidate it. So once core zero's cache receives this message, it will change the state of its cache line from a shared state to an invalid state and respond back, acknowledging that it has invalidated its cache line.

At this point, core one can flip its state to modified, because it knows no one else has a shared copy of this cache line. And finally, it can change the data value from five to whatever the application was writing, for example, eight. This ensures that all memory operations remain consistent with one other.

Know that now if core zero tried to access this cache line at address 0x1000, it would not be able to service it directly out of its local cache, and it would have to send another message over the cache

coherence protocol interconnect to core one to request that it switch from a modified state to shared, write the data back to DRAM, so that core zero could read it into its private cache and correctly return the value eight to the application.

So the reason that this cache coherence protocol matters so much for performance on a multi-core system is because the costs involved in accessing cache lines in different states vary dramatically. So as we mentioned before, the cost of accessing memory in the private cache, if the cache line happens to be in the right state-- meaning that you're trying to read a cache line that's already in a shared or

modified state, or you're trying to modify a cache line that's already in a modified state, is going to be quite fast, on the order of one to tens of cycles, which is quite good on modern processors.

Now, if it turns out to be the case that your cache line isn't in the right state in your private cache, then you might have to go to DRAM. And accessing DRAM, as we mentioned before, is on the order of hundreds of cycles on a modern processor.

And this is quite slow. So you'd like to avoid this. But what's even worse is that if it happens to be the case that your cache line is stored in someone else's cache, and in fact, it might be stored in many cores caches and will require many interconnect messages to be sent from your core to other cores and waiting for them to acknowledge your request, this can easily take anywhere from hundreds to even thousands of cycles for cache lines that are being accessed concurrently by many cores in the system.

And this can significantly reduce the overall throughput of your application. So later on we'll look at how this actually shows up in a real application by using a lock data structure as an example. Now before we move on, I want to mention that, of course, in a real system things are much more complicated than the simple picture I drew for you just now.

Just to give you some sense of what a real multi-core machines looks like today, it actually consist of multiple sockets, meaning physical sockets on your motherboard into which processors plug in. Each socket is actually split into different cores.

So there's many cores per socket. The sockets themselves are connected by some point-to-point interconnection protocol, for example, Intel's QuickPath Interconnect QPI, or AMD's HyperTransport.

And the DRAM in the system is actually connected to each individual socket. And each memory access, in fact, takes place through also messages being sent over the same interconnection fabric.

Although the realities of today's multi-core systems are quite a bit more involved than the simple model I showed you before, as we'll see in the next segment it actually suffices for us to understand the performance of simple data structures, such as a spinlock, and to understand why it performs well or doesn't scale well on a multi-core system.

6.2.2 Implementing a lock

All right, so let's now look at how do you actually implement a lock on a cache coherent, shared memory, multi-core system.

So first if all, we have to figure out why do we even lock implementation when we already have the cache coherence protocol that ensures that all memory operations happen in a sequentially consistent manner?

So the reason why you might want to implement a lock is to uphold higher level invariance, such as applications reading data from shared memory, modifying it, and writing it back to shared memory that you want to appear atomic with respect to other cores so that you don't have other cores observing a data structure in an inconsistent state.

So typically, what a lock looks like that allows us to do this is a pair of functions, acquire and release. And we're going to look at how these functions are implemented underneath of this interface.

- **acquire(lck):**

  me = atomic_inc(&lck.last) while lck.current != me:

  pass /* spin */

- **release(lck):**

  lck.current = lck.current + 1

And the particular lock implementation I'm going to present you here is what's called a ticket lock. And this ticket lock, the lock variable itself, consists of two integer values. So this variable lck that's passed to both the acquire and the release function consists of two components, as we'll see in a second.

There is a component called last and a component called current. And they both start out at 0. And the intuition behind how a ticket lock is going to work is sort of like number machine at a bakery. When you come in, you pull a number from this sequence of numbers available in the store. And each customer gets an increasing integer value. And when the Baker is ready to hear what you want, he's going to announce the next integer in turn. And the person with the right ticket is going to walk up to the counter and obtain service.

So this is the algorithm that we are going to implement in this ticket lock. And the way this acquire function is going to work is it'll first increment the last integer value in the lock variable and then wait until it's the turn of that function of that core to continue executing by waiting until the current field reaches the ticket that it pulled out of the last value.

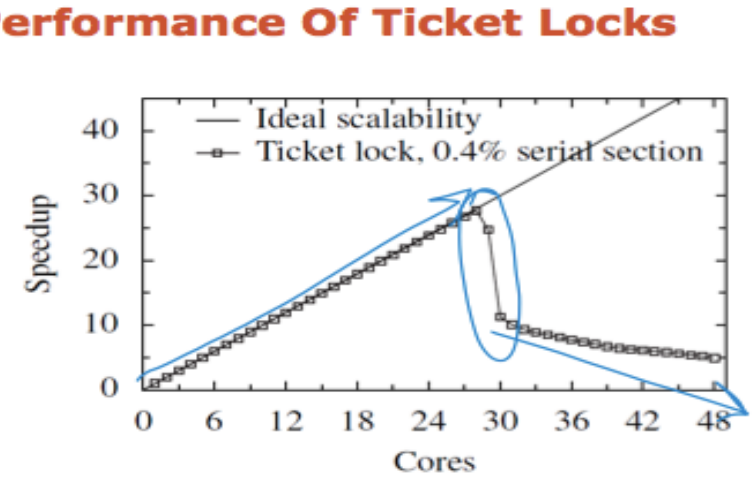
And all this interesting function atomic_inc does is it atomically reads the value of the last field, returns that value to the caller, and also increments that value in memory, all at the same time. Such an atomic operation is critical to implementing anything concurrent a multi-core machine. And the way it actually works is by having the hardware provide a special instruction. The way this instruction works is it gets this cache line into a modified state and then stops responding to any cache coherence messages from other cores while it does its work. It reads the value out, saves a copy locally, increments the value, writes it back to that memory location, and then allows all other cache coherence messages to be processed afterwards. This ensures that this whole atomic increment and read operation happens as a single, indivisible unit.

So building on top of this abstraction, let's figure out how two cores might try to acquire the same shared locked from using this implementation.

So suppose we have two cores, core 0 and core 1. If core 1 starts running this acquire function, it's going to acquire a value of me equal to 0. And atomically, it'll bump this last value up to 1. And then the core will look and say, oh, actually, my value of me is equal to 0. The value of current is equal to 0. I can proceed to execute. And in fact, that acquire will return. And the application can start executing its locked serial section.

Now, if a different core, core 0, comes along tries to acquire the same lock, it'll obtain a different value of the me variable. In this case, it'll be 1 because the last value is equal to 1. And the atomic_inc function will increment that value to 2 now. But now that core 0 cannot precede. It cannot return from the acquire function because it's not yet its turn, according to the current field. In fact, what has to happen now is that core 1 has to release this lock by increment the current field-- namely, the core whose turn it is to now go.

So after core 1 runs the release function shown on the slide, it's going to increment the current value from 0 to 1. And core 0 can now start executing because it's now its turn to proceed. So this is a very simple implementation of a tickets lock. So how does it perform as we add more and more cores to the system? So to give you some sense of how you might measure or think about scalability in a quantifiable way, here's a simple graph. This doesn't show any data yet. It just shows what ideally scalable system might look like.



On the x-axis here, we have the number of cores in the system ranging from 0 to 48 cores in our experimental machine. And on the y-axis is the speed up, namely, how many times faster is an application running at a certain number of cores compared to that application running on a single core.

So maybe the point here, at 30 cores, is equal to speed up of 30, meaning that the application runs 30 times faster if you have 30 cores in an ideal case.

But what happens if we now take a lock, and we try to measure the performance of some application that has a small serial section in it protected by the ticket lock whose implementation we just saw on the previous slide. Well, for a very simple ticket lock that takes up only 0.4% of the application's total runtime, the speed up is shown on this graph with these squares.

This is a very puzzling curve because, for quite a while, the performance goes up quite nicely. And this is roughly what you would expect because the cereal section protected by the ticket lock is less than 1% of the program's totally execution time. And such a small serial section shouldn't impact of the total scalability up to, roughly, 100 cores.

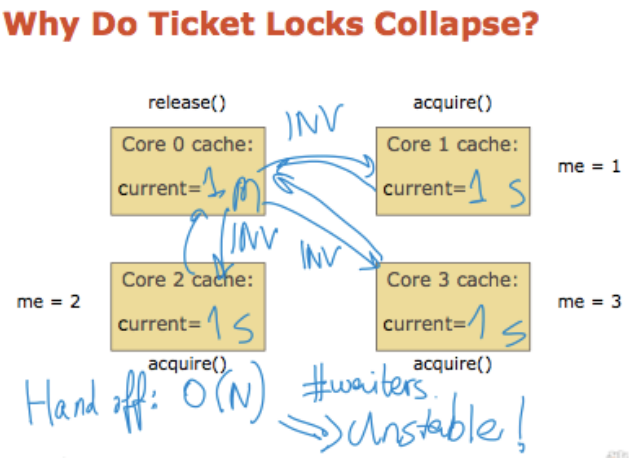
At which point, every core will be running the serial section at one point of time or another. Now something quite drastic happens at around 30 cores, as you can see. The performance suddenly collapses.

And you see much less total performance, with just a few extra cores. And in fact, the total performance of the system keeps going down as you add more cores to your total system. This is disastrous.

What this means is that as you're buying more and more cores to make your system run faster and faster, in fact, it runs slower and slower the more cores you add to your machine. This is what's called a ***collapse of performance***.

And this is something you want to avoid in your system if you want to make sure that it performs properly on a wide variety of hardware and workloads.

So what causes this kind of a scalability collapse for our simple ticket lock implementation? Well, let's take a look at how a ticket lock actually works at the level the cache coherence protocol. Suppose we have four cores now, core 0 is currently holding the lock. So it has a current value of 0.



And cores 1, 2, and 3 are all sitting inside of the acquire function waiting for the current variable to be bumped up to their value. And all of them, because they keep reading the current value, have it in their cache in a shared state. So now, let's consider what happens when core 0 finally releases this lock by calling the release function. If you remember what the release function does, it actually increments the current value. So what has to do is, first, send invalidates to every core to make sure that it can modify this cache line storing the current value in the first place.

After it sends all these invalidates, every other core in the system that's running acquire has to switch its cache line from a shared state to an invalid state. And only then can they send acknowledgements back to core 0, and allow core 0 to switch its cache line into a modified state, and, finally, increment its current value from 0 to 1. At this point, core 0 is done, but every other core is going to try to fetch this value from core 0's cache or from DRAM, again, incurring quite a bit of messages being sent all over the interconnect network and accessing the DRAM just so that all of these other cores can figure out who is going next.

And in fact, for all of these messages being sent through the interconnect, only one core will win in the end. What you can imagine will happen is that every core will end up in a state where it has the current value of 1 in a shared state and only core 1, at the end, will be able to acquire the lock and proceed with execution. So as we can see here, the time it takes for one core to hand off the lock to the next core, in turn, actually takes an amount of time proportional to the number of cores waiting to acquire the lock.

So in asymptotic notation, what this means is that to the time to hand off the lock from one core that's releasing it to another core that's acquiring takes O(N) time, where N is the number of waiters waiting to acquire the lock on the entire system.

As you can see, this is an unstable situation. The reason this is unstable is that the more cores you have waiting to acquire the lock, the longer it takes to hand off the lock from one core to another. And now you can sort of appreciate the intuition behind this collapse. Even if you start out in a situation

where you don't have contention for this lock, as soon as something starts going a little bit slower, more cores start to pile on, the hand off time takes longer and longer.

And because everything slows down, yet more cores pile on. And it's quite quickly degenerates into a situation where every core is waiting for this lock, and it's taking very long to hand off from our core to another. And this is the reason for that very precipitous collapse we saw in the previous slide.

So such performance collapses are quite dangerous in real systems because it means that a simple change in terms of adding the number of cores, or changing the workload slightly, can drastically alter the performance of your overall application.

And that's something you want to avoid in practice. And in the next segment, we're going to look at how to implement a scalable lock that avoids such performance collapse.

6.2.3 Non Collapsing Locks

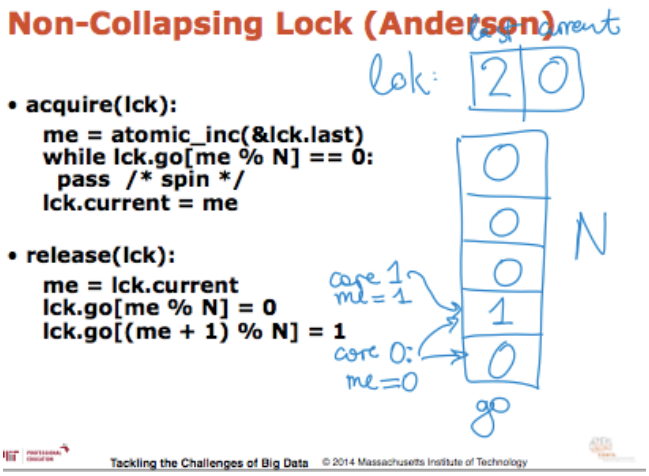
So let's take a look at how we can avoid the scalability performance collapse we saw with our ticket lock, by implementing a slightly more clever lock design. So let's try to understand what is the best that we can hope for from a lock implementation. So of course, we would like to avoid the kind of performance collapse we saw in the previous segment by having a lock that has a constant time for handing off the lock from one cored to another. This will ensure that our total performance doesn't go down as we add more cores.

Of course, one important thing to keep in mind is that, no lock implementation will allow us to improve scalability with the number of cores, because fundamentally, a lock protects a serial section, which is going to dominate to your performance as the number of cores keeps increasing.

So a lock fundamentally is at odds with scalability. But for now, let's try to figure out, how do we design a lock that at least doesn't collapse in terms of overall performance.

One implementation of a non-collapsing lock is called an ***Anderson lock***. And at some level, it looks quite similar to the ticket lock that we saw in the previous segment. Each lock, variable lck, corresponds to a pair of integers variables, last and current, stored in memory. And both of these values initially are zero.

But as we saw in the previous segment, a lock that has a single cache line storing at state will have to transfer this cache line between all of the cores trying to acquire it. So in order to avoid this kind of contention that leads to performance collapse, we're going to allocate a large array, called go, that will allow each core to wait on a different memory location, so they don't contend for the same cache line. So this go array is going to contain a number of different elements, say, n, which is the maximum number of cores or threads you might have in your system. And all of them will initially have the value 0, except for the first one, which has the value 1.



So how do you go about acquiring a lock in this Anderson lock design? So the acquire function starts out similar to a ticket lock, by grabbing a value from the last value in the lock variable.

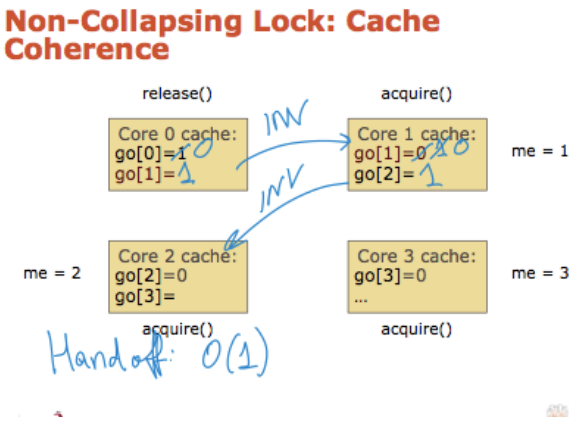
So if, for example, core 0 were to try to acquire this lock variable, it'll get a value of me equal to 0.

And it would atomically increment this last value to 1. Now, what the Anderson lock does is wait for its corresponding memory location the go array to switch to the value 1. So in this case, it's already 1. So core 0 can proceed with the running its serial section, because it's acquired the lock successfully. But if

another core, say, core 1 now tries to acquire this lock, then it will also have to obtain a me value, which in this case will be equal to 1, because that's the value of that last variable and bump it up to two. And then, it's going to wait for a different memory location in the go array to switch to 1, the first location.

In this case, this is what's going to help us avoid the collapse. Each core is going to be spinning, waiting for a different location in the go array to change its value, and thus, avoid the kind of communication between all cores sharing the same cache line.

Now, one practical note to note here is that, the cache coherence protocol keeps track of individual cache lines instead of variables. So in order to make sure these cores can independently access locations in the go array, each location must be a separate cache line. And because each cache line is 64 bites, you'll, in practice, have to pad out elements in this go array to 64 bytes each. But assuming that works out OK, this design will ensure that cores can independently access locations in this large array.



And now, releasing a lock is actually quite straightforward. What a core has to do is simply set its own location in the go array to zero. So for example, core 0 will clear out its 1 value to 0 and bump the next guy in the go array to 1. So in this case, core 0 is going to signal to core 1 that's it OK to proceed with acquiring the lock. And it can do this in a scalable fashion.

We can see why this happens by looking at the interconnect messages that take place when an Anderson lock hands over lock ownership to a different core. As before, suppose we have four cores trying to acquire this lock, and core 0 currently holds it.

As we can see in this diagram, every core is going to have a copy of its own element of the go array in its cache. But when core 0 tries to release this lock, it's going to, of course, write is 0 to its own location and 1 to the next location in the go array. And this will produce exactly one invalidation message on the cache coherence protocol, from core 1 to core 1, because core is the only one that's actually caching go of 1. So once core 1 sees that value, it can acquire its lock, continue executing. And when it goes to release it, it'll similarly switch that back to 0 and write a value 1 to the next guy's go element array, producing exactly one more invalidate message on the Cache Coherency Interconnect Fabric.

So as we can see, this protocol ensures that we send one message on the Interconnect Fabric for every lock handover from one core to another. So in asymptotic terminology, we can say that the hand-off is an O(1) algorithm. It takes a constant amount of time, independent of the number of other cores that might be trying to wait for this lock. And this is what's going to ensure that are lock doesn't collapse under increased loads or increased core counts.

Now, one problem with the Anderson lock in practice is that, it requires preallocating this large array of go elements that's as large as the maximum number of threads or cores you would ever see.

6.2.4 Lock Free Synchronization

There are other designs, such as the MCS lock, that avoid the space overhead. I will not show the pseudocode for this lock because it's a little bit more complicated. But I can at least draw you the diagram for what this lock's data structure memory looks like.

So the lock consists of two pointers, head and tail. And in the base case, when no one is currently waiting for this lock, that's all the memory that we're going to need to store this lock. And as more cores try to acquire this lock and start waiting for it, the cores will themselves form the go array out of a linked list of go elements.

So if, for example three cores are waiting for this array, we will have three elements in a linked list that starts at the head pointer and chains together through all of these elements contributed by individual cores up until the table this linked list.

And the first go element is going to be 1, as in our Anderson lock that we saw before. And everyone else will have a 0 value. So as we can see, we can easily grow this lock to more cores by just growing the linked list. But we do not have to preallocate space for any number of cores or threads ahead of time.

So to summarize, we've seen how to implement a non-collapsing lock, which helps us avoid performance collapse when we have many cores contending to acquire the same spin lock. And one important thing to keep in mind is that even these non-collapsing locks cannot fundamentally improve the scalability of a multicore program, because the serial section protected by the lock still remains and still can ruin the total scalability of your program.

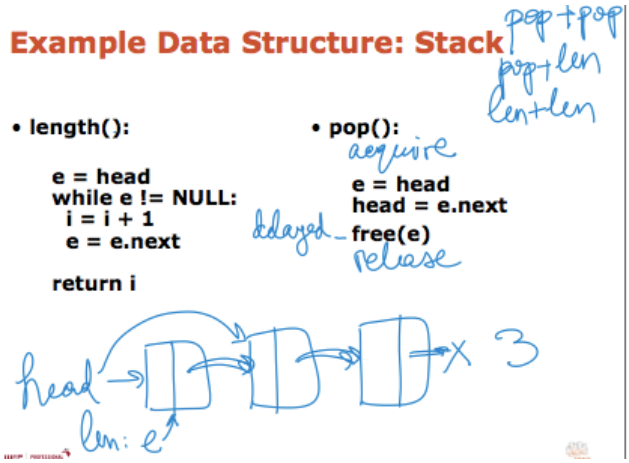
And if you want to improve the scalability you have to eliminate the contention, or the serial section represented by this lock.

One way to do it is to use finer-grained locks that might not be contended.

And another technique that we're going to look at next is to avoid locks altogether for certain kinds of operations.

So let's now take a look at how do we actually improve the scalability of some data structure on many cores.

And let's take a fairly simple example data structure, just a stack.



Our stack is going to have just a head pointer which will point to the first element of the stack. And each element of the stack is going to be like a linked list. It'll point to the next element in the stack. And so on until the end of the list. Probably ends in a null pointer of some sort.

So we're gonna have two operations that we care about on this linked list. It will be a function that returns its length just by starting at the head and counting up every time it takes a hop to the next element, up until it reaches a null pointer.

Another function we're going to have is a function called pop that will remove that first element from the front of the list or the stack.

So how does this data structure run when you run it in a multi-core system?

The thing we have to really worry about are what's called race conditions. What happens if two of these operations run at the same time? They might interfere with one another and produce an undesirable result.

So here, since we have a relatively small number of operations, we need to consider only a small number of pairs and understand what happens if they run concurrently. So let's first see what happens if we have a pop operation running at the same time as another pop operation runs.

So in a simple version shown on this slide, the concurrent pop operations might both execute the first line of code, e = head. And both will have their individual e pointers point into the head of the list. Then both will execute the next line of code. So both will set the head pointer to the next guy in the list. And then both will free their respective e element, which would doubly free that first element in our stack.

This is kind of undesirable because freeing the same element twice will likely result in memory corruption if you're programming in a language like C. And also, running these two pop operations concurrently only removed a single element from the top of our stack, which is kind of unfortunate.

The thing you probably meant to happen was, of course depends on a specification of a data structure, but what you might expect in the case of the stack is that running two pop operations should remove two elements from the stack, regardless of these pop operations running separately or concurrently from one another.

And, of course, the traditional way to solve this problem is to add a lock around the code of the pop function. So we'll call acquire before running pop and we'll run release at the end. This way even if we're on two pop operations at the same time, the lock around the implementation of pop will ensure that there is no race condition like the one we saw. And two pop operations will remove two elements from the front of the list. So that's great.

But now let's see what happens if you run a pop operation concurrently with a length operation that computes the length of the list. And to understand what we even want to happen, we have to think of what should be the allowed result if you run a pop and a concurrent length operation. Because the pop is actually gonna change the length of the list. So in this case, the list use to be three elements long.

After the pop it's gonna be two elements long. Should length return two or three? Well a natural answer that is typically chosen by most data structures is to say if these operations run concurrently, then it's OK to return a value as if length ran first. And it's also OK to return a value as if the length function ran second.

So either two or three would be an acceptable result from the length function if it ran at the same time as the pop. But there is a more subtle problem here. So what happens if we run this pop function at the same time as length? So length initially grabs the pointer to the head of the linked list, e=head, and stores it over here. I'll draw it at the bottom. This is the length function running. And next, the pop function runs, acquires the lock, and switches the head pointer to point there, and frees the first element of the linked list.

Now if the length function continues running, it's going to dereference an element in that freed piece of memory. Which can potentially result in, again, memory corruption, accessing arbitrary memory locations, and your program might crash altogether, or return any value for length at this point. This is rather undesirable.

And one way you could, of course, try to fix this is to put a lock around the entire length function. This is what we've been implementing all this time. So you could call acquire at the beginning of length and release at the end. And this would have solved the problem. At the cost of reducing the concurrency of this program and as a result reducing its scalability.

So what could we do here that would be better? So if we have the acquire and release, of course, we will not have the race condition that I showed on the slide. So we will not inadvertently access the freed memory.

But one thing that we could try to do better is use what's called a ***read-write lock*** instead of a simple spin lock we've been implementing so far. The semantics of a read-write lock is that we have two kinds of lock holders, a reader and a writer. And it's OK to have multiple readers acquiring the lock at the same time because none of them are modifying the data structure.

But only one writer can hold the lock at a time. And if there's a writer no readers are allowed. So the way this would typically look like is that the length function would call the function called read_acquire and read_release instead of the acquire and release functions we saw earlier.

And what this means is that when pop runs length cannot run. But at least if you run two length operations at the same time, they'll be able to execute concurrently with one another. Both computing the same length of this linked list. Now this is kind of nice. Pop is, maybe, not so scalable. But multiple length operations should, sort of, scale better. Multiple length operations can both acquire this read lock at the same time.

The problem lies in the implementation of this read_acquire and read_release. Because, as we saw earlier, somewhere the state of this lock must be stored in memory. And these read_acquire and read_release operations must access that cache line storing the count of readers, for example. And the hardware operations on those shared cache lines will form a serialization point and make the length function not scalable, even though their read lock should, in principle, allow it to scale perfectly.

And this is an unavoidable problem of a read-write lock. Either you're going to have to modify a shared cache line inside the implementation of read_acquire and read_release, or you're going to have to allocate a large amount of memory in these functions to avoid this kind of contention.

So instead, what we're going to look at is a technique to avoid any locking of the length function altogether.

So let's see what goes wrong. Specifically, if we eliminate the read_acquire and read_release from the length function, as we saw before, the problem that shows up is that the pop function will free the element that the length function might be looking at. So the thing I'm going to propose here is to defer the freeing of the element. So instead of freeing it right away, we'll call the magical function called delayed_free.

We'll talk in a second about how we implement this function, delayed free. But what it's gonna do for us is remember that it needs to free this element, e, at some point. But it will not do it quite yet, or at least not until the length function finishes running.

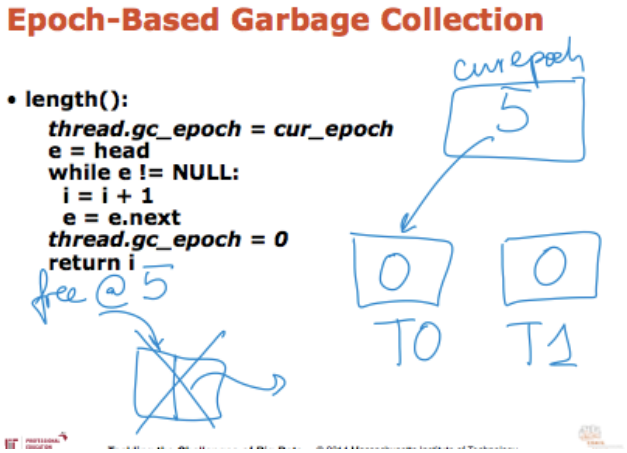
So what this is gonna do is that even if the length function here is running, with the element e pointing at the head of the list, even after pop passes it's element e to the delayed free, length can continue running and continue counting the number of elements in this list. Returning the value three at the end.

And after length is done, then maybe we can garbage collect that first element on the list.

The benefit of this approach is that multiple executions of the functional length on multiple cores do not modify any shared cache lines whatsoever. They can operate purely in their local, private caches, and thus achieve perfect scalability. They do not interfere with each other in any way at all.

But we have to address this one critical problem. How do we implement this magical, delayed free function in a way that doesn't break the scalability of the length function? So typically, this is done by implementing some form of a garbage collection scheme.

And I'm going to present you a simple implementation called an ***epoch-based garbage collection scheme***.



The way it works is it breaks up the total execution of the system into what are called epochs. So each epoch is an integer number. Let's say you're representing the number of seconds on the current clock. And it'll be incremented, let's say, once every second. So the data structures that we'll need to maintain in order implement such an epoch-based garbage collection scheme is the current epoch number.

So let's say it's currently four seconds and some arbitrary starting point. And for each thread running in the system, we'll maintain the epoch in which that thread is currently running the length function. So know that I've modified the implementation of length shown on this slide.

So the first thing it does is it actually reads the current version of the epoch variable and sticks it in a per thread gc_epoch value. So let's see how this helps us figure out when we can finally garbage collect that element from a linked list.

So suppose our pop function decided this was the element that has to be garbage collected. And what it's going to do is it's going to remember that this guy has to be freed at, let's say, the epoch. So the current epoch is 4. Let's free it at the next step epoch, 5. And we're going to decide when it is safe. to do so. So if every thread is currently running and has an epoch value of 0 in its thread, it means that they're not running the length function at all.

But the current epoch is still 4, so it's not OK for us to free that element. So if thread 0 here starts running the length function, it's going to copy the current epoch of 4 into its gc_epoch variable and proceed to execute the length function. At this point, the current epoch might be bumped up to the value of 5. Now, we cannot free that element yet, because the guy that started running in epoch 4-- namely, thread 0-- might still have a reference to that element that's on our "to free" list.

And it's only at the point when that guy finishes running and no other guy can possibly start running in epoch 4 anymore that we can finally free this element, mark to be freed at time equals 5. So by implementing this kind of an epoch-based garbage collection scheme, we can efficiently decide when to free elements, without having to synchronize for every invocation of length.

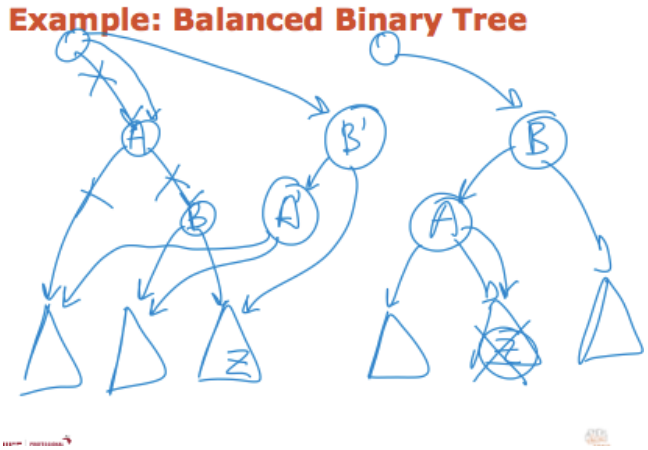
The insight here is that we're able to batch these garbage collection operations by bumping these epochs relatively infrequently, once ever second or so. And as long as you don't care about freeing your memory as soon as possible, then it's perfectly fine for us to batch these kinds of garbage collection decisions.

So I've shown you here are how we can implement a relatively simple data structure-- namely, a stack--

using this kind of a lock-free approach that allows us to scale perfectly for read operations. But you have to think a little bit more carefully for more complex data structures.

And let's look at another example, namely a balanced binary tree. So in a balanced binary tree, you have a data structure that looks something like this.

You have a root of a tree that points to a couple of nodes. And each nodes points to potentially children nodes. And I'm not going to draw the whole tree for you here. I will just say that at the bottom we have a couple of sub-trees, indicated by these triangles.



And the structure of this tree is roughly as shown on this slide. And for clarity, let me just label this node as A, and this other node as B. So one thing that might happen in a balanced binary tree is the tree might require rebalancing if too many elements, for example, were inserted into the rightmost triangle on the diagram.

What a typical balanced binary tree implementation does, in this case, is transform this tree into a different shape. So instead of having node A be on top, it'll actually rotate that portion of the tree so that node B is on top, node A gets moved over to the left, and the subtrees actually stay in exactly the same place as they were before.

So it modifies only the two nodes A and B, and it's effectively raises the rightmost triangle subtree on the diagram, thereby balancing the heights of the subtrees. So balanced binary trees are widely used data structure in database systems and operating system kernels and many kinds of software.

So wouldn't it be great if we're able to perform look up operations on binary trees without having to acquire a lock to serialize us with respect to other modifications of the binary tree, such as rebalancing operations?

The problem is that we can't quite get away with just the delayed free function we saw on the previous slide. And the problem is that a particular look up operation might go down the wrong portion of this tree altogether.

Suppose we have a look up function that's looking for some element, Z, over here in the bottom of that rightmost subtree. When it normally traverses the subtree, it starts from the root. It goes to node A. The element it's looking for is greater than A, so it goes to the right. The element it's looking for is also greater than B, so it keeps going to the right. And eventually it finds the element Z.

Now, if there's a concurrent rotation that happens, then we're in a bit of trouble. Initially, our guy is going to start off from the root, as we saw before. And just as it gets to node A, a rotation happens. So now, our look up operation is sitting in node A, shown on the right half of the slide. And because the node it's looking for is greater than A, it jumps to the subtree shown on the right half of the slide. But there is no Z to be found in that subtree.

So it concludes that node Z is simply not in this tree, because of this race condition that occurred between the lookup operation and the concurrent rotation. Now, the problem is that the look up was not atomic with respect to the modification of the rotation it was performing on this tree.

So in order to solve this problem, we have to use a technique called ***read and copy update*** from the Linux kernel. The way this works is that instead of modifying the data structure in place as we saw here, meaning, we take nodes A and B and update their pointers in place. We're going to create a copy of nodes A and B. We're going to create a node A prime and a node B prime. We'll construct a copy of these nodes as we want them to look on the right hand side of the picture. And we'll splice in pointers to existing subtrees from the left.

So the pointers on A are going to look like this. The pointers on B are going to look like this. And finally, we'll make a single atomic update from the root of the tree to our new node B prime. So now if you ignore all these lines on the left hand side of this diagram namely, this guy, this guy, and this guy-- you can see that the tree formed by B prime and A prime is now the same tree that we wanted on the right hand side of our picture.

But the cool thing that we get by updating the pointer from the root exactly once is that a look up will either see the old pointer and go down the tree with the x'd out edges on the slide, thereby finding Z correctly. Or it will go down the tree with A prime and B prime.

But there, it will also find the correct Z element, because it'll go to the right child of B prime and find Z there. So by performing this one atomic update and bundling up all these little modifications into updates on a copy of the data structure, we're able to atomically update this relatively complex data structure and allow a lock-free lookups to proceed concurrently with modification operations, without having to acquire any locks.

So to summarize, this RCU technique that I've presented to you enables readers to execute concurrently with one another without acquiring any locks, which is great for scalability. And the rules you have to follow are threefold.

First, you're going to have to implement readers without any locks.

Second, you're going to have to make sure that the writers in your data structure update a single pointer for every modification. And if you're writer is current updating multiple values or pointers, you might need to make a copy of that data structure, update the copy, and then make a single pointer swap to the copy.

And finally, whenever you're freeing memory, use the delayed free function that we saw in the previous slides.

If you follow these three rules, we'll ensure that readers do not have to modify any shared cache lines with one another. And as a result, they'll be able to scale perfectly by executing their private caches and avoid generating any cache-coherence traffic at all. So this is great for scaling read-heavy workloads in many practical applications.

And this technique is widely used in both database systems and the Linux kernel to achieve scalability on tens or hundreds of cores.

So in conclusion, in this topic. I've told you about how to achieve good multicore scalability on modern processors. We've looked at how hardware implements a cache-coherence protocol to ensure that caches have consistent values stored in their buffers.

We've also looked at implementations of locks, and we've seen how sharing a single cache line in the cache-coherence protocol can result in dramatic performance collapse for the overall application.

We've looked at techniques to avoid such performance collapse by splitting up our lock into multiple cache lines and creating a non-collapsing lock.

And finally, we've looked at RCU, which is a powerful technique for avoiding locks altogether in the case of read-heavy workload, which ensures that these look up operations can scale perfectly in the presence of other look ups and modifications.

6.3 Big Data Visualizations: David Karger

6.3.1 User Interfaces for Data : The Bug Picture

I'm going to be talking to you today about user interfaces for data. And I'm going to start with the big picture. And in particular, an introduction to myself and my research group. My group is focused on understanding what is it that makes it hard for people to manage data.

And when I started working on this problem, I looked at traditional areas like databases, and information retrieval. But over time, it became more and more clear to me that the real challenges in data management were in the interfaces that people use to interact with their computers. And so that's where the focus of my research group is now. And we sort of have this cycle where we first try to understand what it is that makes it hard to manage data. What are the problems with the user interfaces people have today.

And then we try to create tools that address those problems that we understand. Then we put those tools out and watch people use them. And use that to try to understand still more about what's hard to manage data and create the next generation of tools.

And the theme in all of this work is really the end user and how we can empower them to manage data on their own. So we spend less time thinking about highly specialized tools for power users with specific unusual types of data. And think more about the kind of data management tasks that everybody has.

Now, why is the user interface so important? Well, Hal Varian had a pretty good summary of what's going on nowadays. As data becomes more and more pervasive, the ability to take that data and understand it, process it, visualize it, and communicate with it is becoming more and more important.

And at least three of those tasks, ***understanding***, ***visualization***, and ***communication*** are fundamentally user interface tasks. And it's the user interface that's going to affect how well somebody can do it.

Now, the reason we need those interfaces is that while computers can store and process information far better than a human being, people are still a lot better than computers at a number of information management tasks.

They're better at seeing patterns, noticing oddities, imposing order on some data that doesn't have a model yet, figuring out what a suitable model is. And so if we can somehow create a combination of a computer with its processing capabilities and a human being with their understanding capabilities, putting those together is something more powerful than either one.

And even if the computer is going to be doing in various measures the majority of the work, there still has to be a way for the human to tell it what work to do. Computers are still not very good at taking initiative. I'm going to be talking about interfaces that have value in really two different ways.

One is that they can really help users analyze data in order to reason about it to draw conclusions. And as a step in this understanding, these users need a sort of expanded memory. There's no way that we can think about millions of data points all at once in our heads. The computer can hold those millions of data points and present them to us in a way that we can think about them. Once we see them, we can find patterns, based on that we can develop and assess hypotheses, we can look for errors in the data.

The second major use of interfaces is for communication to others. Even if you see something in the data or you have done some work to understand something in data, a lot of times you can only have an effect with your understanding by communicating what you understand to other people.

And for that, you need ways to present your conclusions to people. And we know from the famous cliche that a picture is worth a thousand words, that visual imagery is very effective for

communicating to other people.

And so using interfaces to create that communication, to share and persuade other people, or to collaborate with them on your understanding and revise your own ideas is equally important in applications of user interfaces.

So just as a quick example of how powerful vision and human understanding is, here we have a well known data set called Anscombe's quartet. It's a collection of four distinct data sets each consisting of pairs of x and y-coordinates.

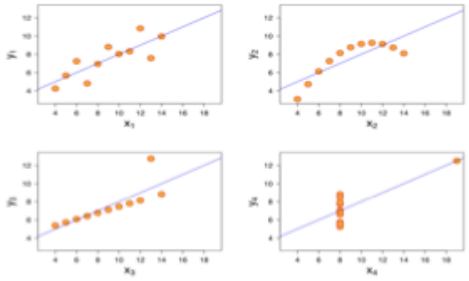
Leveraging Human Insight

I		II		III		IV	
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

- **4 x-y data sets (Anscombe's quartet)**
- **Identical means and standard deviations**
- **What's the difference?**

And if you look at this table, you really can't see anything at all. And if you feed this data into your typical statistical package where it will tell you say the means and standard deviations of these four data sets, you'll find out that all four of these data sets have exactly the same means, and exactly the same standard deviation.

So from a numeric prospective, they're really no different. So what is the difference between these data sets? Well, if we take these x and y-coordinates and treat them as points in a plane and plot them, well then it's completely obvious what the differences between these four data sets.



- We've got one that's almost linear following a diagonal line.
- We've got one that's sort of parabolic.
- We've got one that's truly linear with a single outlier.
- And one that's completely concentrated in the x-coordinate, but has some variability in the y-coordinate that leaps out at us when we look at the pictures.

We didn't see any of that by looking at the numbers.

So user interface design is, of course, a huge sub field of human computer interaction, and I'm not going to be talking about general principles of user interface design. I'm going to be focusing on user interfaces for data.

So let me be clear about what I mean by data. So we're going to be talking about collections of information where I have a large number of items following the same schema.

So for example, I might have a large table, something that came out of a spreadsheet, a table with many rows of data and a certain number of columns. And each of those columns probably has a heading of some sort, which names what kind of information is in that column.

So those column headings form the schema for the data. Or I may have several tables, but not a tremendous number, 10 or 20 different tables of data where again, each of the types of data has a common set of properties.

And the kinds of user interfaces that I'm going to be talking about today, data user interfaces, are ones that really exploit the regularity of that data.

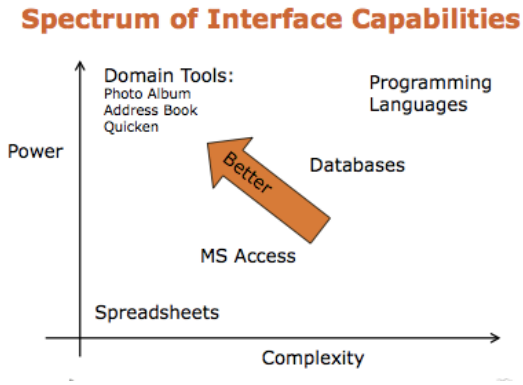
So, for example, the spreadsheet is a really fantastic user interface for tables, right? It shows you the information in nicely organized fashion so that you can, in one glimpse, see all of the values associated with a particular row, see all the values in a column, and so forth.

As another example, a simple scatter plot is something that needs data with two properties, one of which is going to be used as the x-coordinate for your plot and another that's going to be used as the y-coordinate for your plot.

If you look at modern websites, websites use ***templating***, and so all of the restaurant reviews on Yelp look exactly the same in structure, even though their content is different. And that regularity of structure is a help to users who are trying to gather information from that website.

Also on websites we see ***facets***, which are sort of a canonical mechanism for filtering a data set on a particular property. So at Yelp, you can choose to look only at the restaurants that serve Ethiopian food, or only at the restaurants that are cheap.

And so you need this schema, this collection of properties, in order to know what kind of filters to offer.



Now in the spectrum of user interfaces, there's a pretty standard trade off in complexity versus power.

So at the one end, we tend to have tools that pretty much anybody can use, like spreadsheets, which are, I mean, quite complex for pre-computer users, but nowadays are thought of as relatively straightforward, but also are relatively limited in power.

As we go up the power curve, we get more powerful tools like Microsoft Access, which provides a visual interface to databases, then databases themselves, where you can type queries in SQL, and then even higher to general programming languages where you can do pretty much anything you want.

Now, with all the power you get out of programming languages, somebody might ask, why do I want anything simpler? Well, the answer is that that power comes with complexity, so programming languages can only be used by a small number of people.

We need tools that more people can use, and so we try to work our way down this complexity curve. Now at the same time, we'd like to preserve the power of the interfaces and give up as little as possible.

And one way to do that is with domain specific tools.

So if you have a specific data set-- like a collection of photographs with their metadata or an address book or accounting, with each of those data domains and the particular set of tasks, you can build a very specialized user interface that has all the power that you need for that domain without all of the complexity of general purpose programing.

The problem is that you need to do it for every domain. And so given the variety of types of data that people are going to be working with, it's really not possible to do that. And so we make the sacrifice that OK, maybe we can't have all the power.

Let's have as much power as we can have while building tools that are not too complex for regular users.

So as we go on to look at user interfaces for data, I'm going to be ***excluding*** certain things that could be thought of as data. So, for example, image interfaces, things like Adobe Photoshop, that we used to edit images. Well, we could think of this as a giant collection of data, where there's one column for every pixel in the data in the image.

However, that's not the way we work with that data. We have a lot of semantics associated with what an image is and we build tools that really leverage that.

Similarly, for text interfaces, you can think of every document as a giant vector of words and apply database technology to it. But again, we assume that there's certain semantics to what text is and what kinds of interactions people are going to want and so we built specialized interfaces for that.

Similarly, for audio and signal processing and speech interfaces, too much semantics to be thought of in this general data management sense.

Instead, I'm going to be concentrating on tables, rows, and columns, and working with data that generally fits that format. OK.

So to overview what we're going to be talking about today, I'm going to start by introducing you to information visualization, which is a very large area that long predates computer science, just looking at how we can leverage human perceptual abilities to take abstract data and present that data in something that a human can understand at a glance and draw meaning from.

Then, because these information visualizations are so powerful, I'm going to give you some cautionary tales about how one can lie with visualizations and talk about what one should not do in creating visualizations.

Then, moving beyond the kind of stuff that you can do on paper, with the arrival of computers, we're going to look at the opportunities to interact with your data dynamically, changing the visualization on the fly, and the use of that in the growing field of exploratory data analysis.

Once we've looked at that field in general, I'm going to talk about a particular dominant mechanism for making it easy to interact with data, which is called direct manipulation. The idea is to make it feel like you're using your hands to manipulate the data.

Then, once we've got that basic tactic for interaction done, I'm going to talk about a common strategy for data interaction, which is to look at an overview and provide filtering and zooming capabilities and then details on demand.

Finally, I'll take a quick look at the way data interfaces have spread on the web and the sort of standard model of data interfaces that seems to be dominant right now.

Then to finish up, I'll look at two of our own research projects, one, a tool for going beyond the spreadsheet; for taking a bunch of the things that make a spreadsheet good for interacting with data and figuring out how to make them even better.

And then I'll describe another research project we've been doing on letting end users create their own interfaces for data exploration, instead of being limited to those that are created by others for them.

6.3.2 Information Visualization

Our next segment is going to be looking at information visualization. Information visualization is something that we experience every day with instrumentation, with literal heat maps for weather, with maps of transportation going from place to place, social network visualizations.

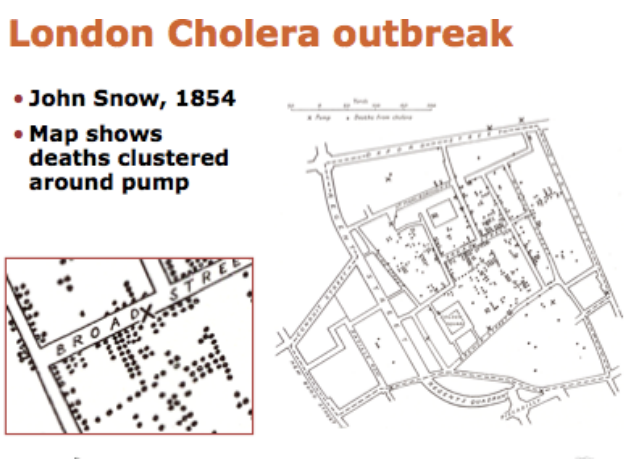
This is a lovely little visualization of the edits that we're done to Wikipedia, a particular page, over time.

We see this rich visualizations everywhere, but they long predate computers.

So here, from the New York Times in 1981, is a beautiful visualization of an entire year's worth of weather in New York City. And so this visualization takes 2,200 numbers, imagine trying to look at those in a spreadsheet, and puts them into a beautiful comprehensible aggregate that shows you the details on every day and the trends heat, humidity, precipitation, all lined up in ways that are easy to understand.

But actually information visualization started far earlier than that. This is a map that was found on a wall in Catalhoyuk in Turkey. At least, most people believe it's a map with abstract representations. There's always debate about what things might mean.

But moving forward from ancient times, a real important moment in modern information visualization was around a cholera outbreak that happened in London in 1854.

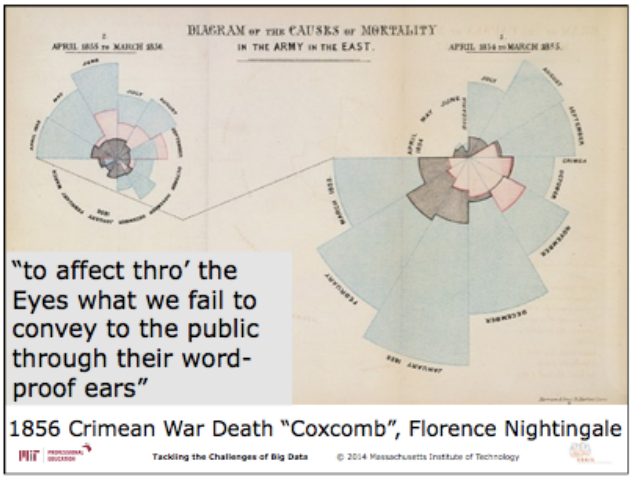


Back then, people believed more generally that disease was caused by some sort of miasma in the air that spread around and infected people. And John Snow had this hypothesis that maybe the disease was coming through the water, and he sat down and gathered information about this epidemic that was spreading through London.

And on this map, he basically put a dot where anybody died of cholera. And he also put an x at the location of every water pump in London. And looking at this map, it's pretty obvious that there's a massive correlation, right?

And that, in fact, the people who are clustered around a specific water pump, are the ones who are suffering from cholera. And John Snow was able to convince the city council to remove the handle from this water pump, and shortly thereafter, the outbreak subsided. Although it's not clear exactly whether there's a causation there or whether he just happened to intervene at the right time. However, this was a very compelling visualization.

Another visualization which really did have clear impact was the so-called Coxcomb is a visualization that was done by Florence Nightingale.



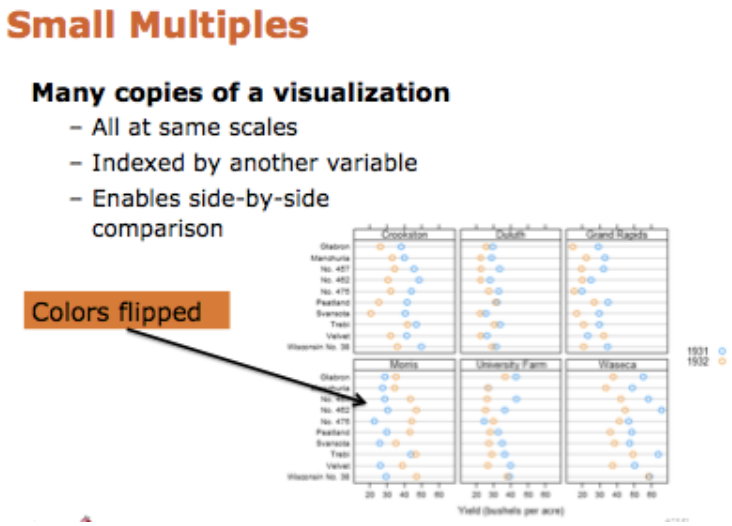
We tend to think of her as sort of the sweet nursy type, but she was actually a very serious campaigner for health, and she created this visualization of the causes of death in the Crimean War. And if you look at this visualization, it stacks together in ways that are not considered quite appropriate in modern information visualization.

The deaths that were caused by battle, the deaths that were caused by other random causes, and the deaths that were caused by disease. Blue being the deaths caused by disease. And just staring at this visualization, you can see that actually, the war itself was no big deal, right?

It was all the disease that was the major source of death in the campaign. And so Nightingale put together this visualization, as and she said it, to effect through the eyes what we fail to convey to the public through their word proof ears.

So again, it's this idea that a picture is worth a thousand words. This visualization and her campaigning were enough to convince the governments of the time to create a commission of inquiry that actually looked into the sources of disease in the Crimean War and tried to do something about them.

Another example of the power of visualization comes from using this technique called small multiples. This was done back in the 1930s during a study of the yields of different kinds of barley, I believe, some kind of grain.



They planted these at a number of different farms, and looked at the amount that was being yielded. In 1931, that's the blue dots, and in 1932, the orange dots, for each of these varieties at each of these farms.

And again, if you had these yields as a collection of numbers, nothing much would have come out of it.

However, by looking at this visualization, you can see that there's something wrong. If you stare at these six multiples, one of these things is not like the others. You can pause the video if you'd like to figure out what that is, but if you'd like me to tell you, take a look at the bottom left corner.

We have a particular, in all of the others, the blue dots are to the right of the orange dots. It looks like 1931 was a better year for all of the varieties. On the other hand, in the bottom left corner at the Morris farm, we see the that blue dots are below the orange dots in the visualization.

What was going on?

Well, somebody created a visualization like this and realized that somebody had actually flipped the year labels on the Morris data. So again, something that is obvious to visualization, which was not at all clear if you were working with the raw numbers.

All right, so we've seen some visualizations, now let's step back and talk about what they are.

What is visualization?

So one way to look at it is it is simply an attempt to transform various kinds of abstract and symbolic information into geometric representations that can be perceived by the eyes.

Bertin, one of the early greats in visualization, talked about finding an artificial memory, so think about having the paper or the record as an artificial memory that supports our ability to think about the information through perception.

Jock Mackinlay, in more modern times, talks about it as the use of computer generated interactive visual representation. So he's already talking about exploratory data analysis, which we will be looking at in a later segment.

Why do we do it?

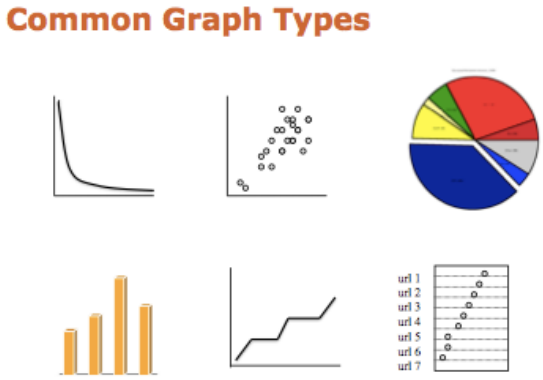
Well, I've said before, a picture's worth a thousand words. Really, we receive information far faster through our senses then we can process it with our thoughts. We just have way more bandwidth.

We can react to an image much faster than we can think about that image. We can perceive and remember details without thinking, and so it's really valuable to transform abstract data that we can only work with by thinking about it, into visual variables, things that we perceive without thinking.

And in doing this, we can leverage what's called pre-attentive processing. This idea that we can actually process information without thinking about it.

So people have done a lot of work to capture the variety of visual representations that people make, and there's hundreds of them.

And I'm not going to try to take you through all of the different details of many different visualizations, that would take an entire information visualization course. Even if we focus just on graphs, we would see that there are many different ways of visualizing information on a graph.

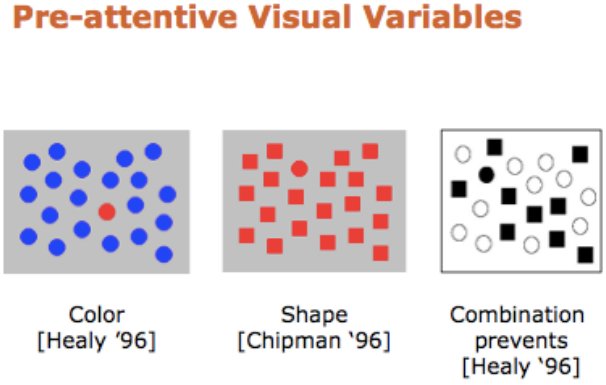


And I'm not going to take you through all of those. But I do want to sort of go lower level to the fact that there are some common classes of visual variables.

And Bertin, one of the people I quoted before, was the sort of person who initiated this sort of study and listed a number of the types of variables that convey information to our senses without us needing to think about them.

Things like the position of something on the screen, the relative sizes of things that we see, their shapes, their color values, whether something is red, green, or blue. The dark or light qualities of the colors, textures that are being shown on the display, or the orientations of lines. All of these things are things that we perceive without having to think about that. And, in fact, this was formalized.

So Treisman, in '85, introduced this idea of pre-attentive processing of visual information, where things like colors, line ends, contrasts, curvatures and sizes, are detected by our eyes, by our visual processing system, before they get to our conscious senses. And we perceive these effortlessly, quickly, and without any particular attention being focused to a specific thing on the display.



And this is backed up with experimentation. So if you look at the picture on the left, you can see right away that one of the dots is not like the others, right?

If you look at the picture in the middle, again, you can see right away, not quite as easily, but right away that one of them is not like the others.

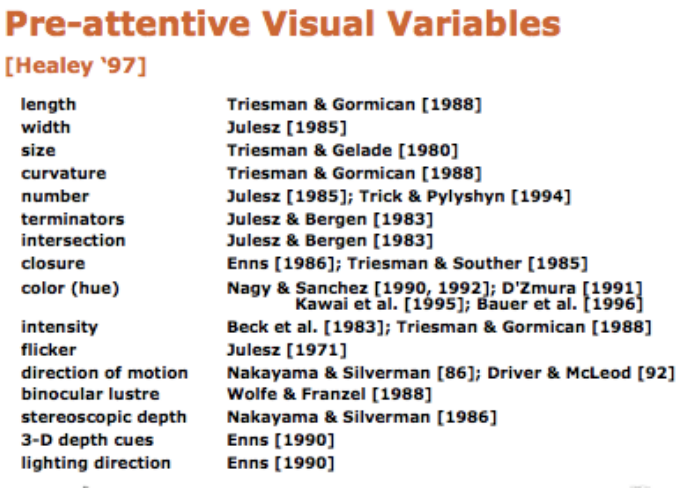
On the other hand, if you look at the picture on the right, it's a lot harder to figure out that there's one thing which is distinct from all of the other things, right?

So in particular, the leftmost one, the only difference is in color, and so you perceive that instantaneously.

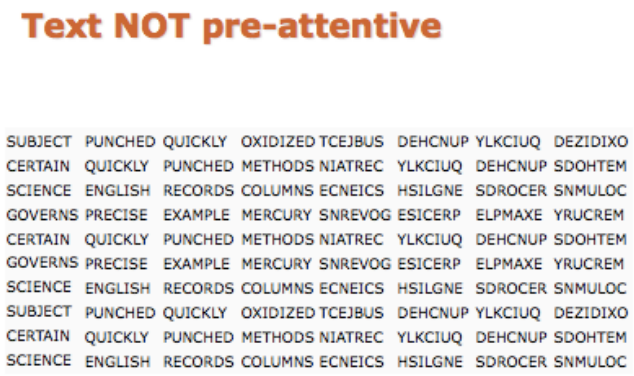
In the middle, the only difference is in shape, and so again, you perceive that quickly.

On the right hand side, you've got this combination of colors and shapes, and that requires higher level processing to figure out what's going on.

So over time, people have done a lot of studies and have identified lots and lots of pre-attentive visual variables, and unsurprisingly, these corresponds to a great extent with the things that Bertin was talking about as the low level visual variables that make up a visualization.



On the other hand, just to contrast, text is not a pre-attentive variable. So if I ask you, what is the structure of this collection of stuff on the screen?



You actually are going to need to stop and read in order to figure out what's going on. You're going to have to engage your higher brain functions in order to recognize that everything on the left is a meaningful word while everything on the right is gibberish.

That's not something that you perceive instantaneously.

Our goal is going to be to transform ***abstract data*** into visual data. And so the first thing is to understand what kinds of abstract data we might be talking about. And here are four general categories.

The first kind of abstract data that we'd work with is what's called ***nominal data***, or what might be called a kind of ***qualitative data***, where there's no inherent relationship between the different data values, things like city names, types of diseases, set of things. And there's no order between the things in the set. There's no relation between them. They're just a set of values.

Then we've got ***ordinal abstract data*** types where there is, in fact, some sort of meaningful order between things, but it's not associated with specific numbers. So notions of cold, warm, and hot. We know that warm goes between cold and hot, but not exactly what value of temperature that's associated with. We might also think about historical eras, right? These are things that we think of in order, even if we don't think about, we know the Renaissance came before the Enlightenment. Historians might have a specific date associated with it, but in general we just think about the order.

Then there's ***quantitative data*** where there are real numbers associated with the values. Sometimes this is absolute quantitative data. So with things like mass and length, there's a clear notion of a 0, and everything is relative to that clear notion of a 0. At other times you might have relative data where the 0 is kind of arbitrary. So dates, where do we set the 0 for date? Well, lots of different calendars argue about that. Where do we set the zero for position? Well, we've now all agreed that it's in Greenwich, but it wasn't always that case, right?

And so latitude and longitude has a certain arbitrariness to it; it still has specific numbers. And so comparisons between the numbers are meaningful, even if the absolute values of the numbers are not meaningful.

Finally, I'm going to mention ***relational data***. And interestingly, a lot of the work that I'm going to be showing you, the historical work, doesn't really touch on relational data. That really has broken out, I think, more recently as we think about social networks and things like that. And there's less in the research record about what works well and what works poorly with relational data. But this covers things like social networks, subway maps, that really what's important is the connections between different items. Interesting.

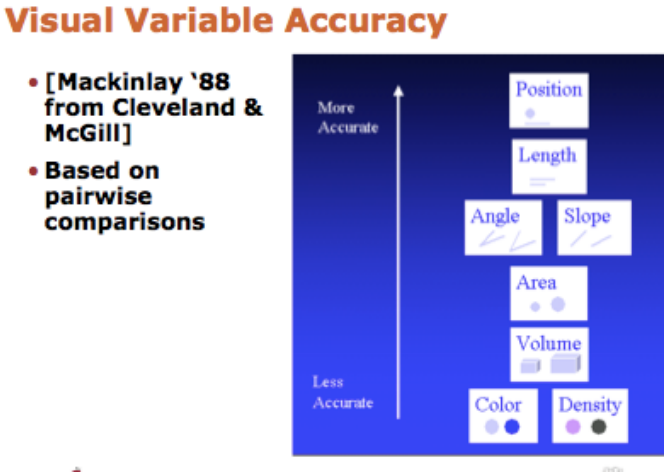
So if we look at the original set of visual variables that I showed you a little while back, we can see that some of them are sort of better arranged for displaying quantitative information, numeric information, and others are really better for qualitative information.

So positions, we can really understand exactly what a position is, the relationships between positions. There are natural orders. There's a natural order and number associated with sizes. This is bigger than that. The area of this is so and so much. So these are great for showing quantitative data types.

On the other hand, if we look at different shapes, well, is a triangle bigger or smaller than a square? Who knows, right? So it doesn't really provide any notion of order or number, but it is good for showing different types of objects. So that works well for nominal information.

Similarly, variations in color are qualitative. Is blue bigger or smaller than green? I don't know. On the other hand, variations in saturation, how intense a color is, there is a quantity or an order associated with this as you go from lighter to darker.

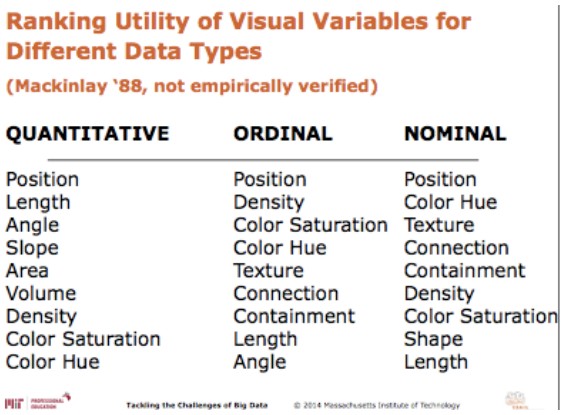
So people have done studies about which kinds of visual variables work better for which kinds of data and have sort of put an order of precision on different types of visual variables.



So they found that, really, people understand or perceive position most precisely. So if you really want to convey a visual variable well, you do that by transforming your abstract variable into a position.

Lengths are also pretty good. Angles, slopes, still can convey numbers, but are less accurate. Working your way down, things like areas, volumes, and worst of all, colors, people don't read a specific number off of those, and so they're not great for conveying specific numerical values.

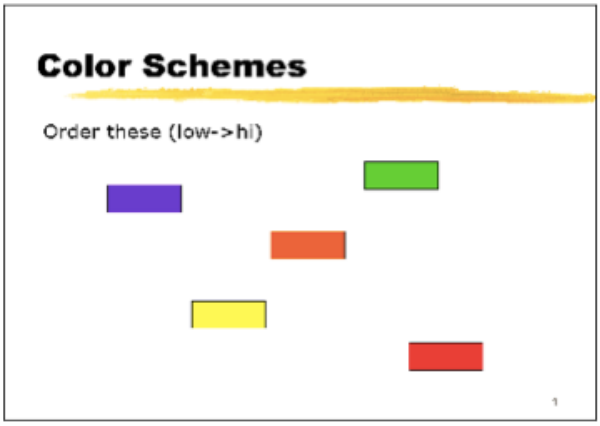
And they've actually done the work of figuring out for these different kinds of information, for nominal information, just plain sets; for ordinal information, which is things that are ordered but not specific numeric values; and quantitative information with specific numeric values.



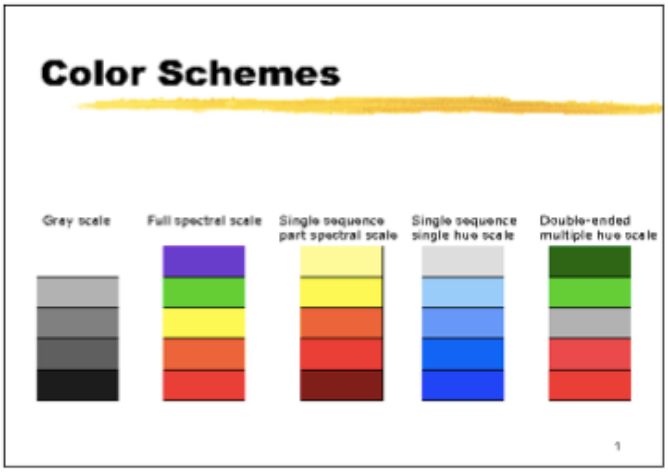
Different visual variables work well. Position wins all the time, but then things tend to separate. So for nominal variables you might want to look at varying colors in order to show different types of things. For ordinal variables, density is very good, darker versus lighter.

For quantitative variables you should look at things like lengths. So this is one thing to think about when you're creating a visualization, is what kind of data you have and what visual variables it should be mapped to.

Just to push on that example, here's a set of colors. This is from one of the experiments that was done to understand this. They asked the users to order these colors from low to high.



And as you might expect they got kind of random results. People who know about the spectrum and different frequencies had this extra knowledge to map it to an order. But in general there's no implicit ordering.



On the other hand, there are other kinds of color schemes where the ordering is very explicit. If we use a gray scale that goes from light to dark, or a single hue color scale that ranges from light to dark, then it's relatively easy to perceive an order.

You still may not receive exact numerical values, though. So you can't ask too much of a color scheme.

So thinking about these visual variables let's you make choices about what kind of visualizations are good. And so, for example, John Tukey, who we'll be hearing more about later, had this famous quote, that there is no data that can be displayed in a pie chart that cannot be displayed better in some other type of chart.

Now he said this before the work on which are better presentations, but we can sort of ground this in the experimentation. Because if you think about it, the pie chart is relying on two things, color to differentiate different types, that works fine, but also on angle.

And we know that length is a far better conveyor of quantity than angle. And so if all you want to do is convey that you have a set of numbers of different magnitudes, using a bar chart is far superior, because people can perceive differences in length far better than they can perceive differences in angle.

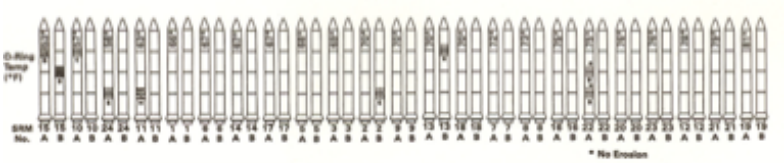
The bar chart also allows you to sort things by length in a perceptible way, which, again, take advantage of what people can perceive quickly.

So just to sort of wrap up this segment with a little case study, back when I was in high school, the Challenger shuttle blew up on launch. And this was a horrible event which led to many investigations, what was going on. What happened was that there was a problem with these O-rings that were separators for certain parts of the boosters. There was unusually cold weather. The engineers of the system actually, it turned out, argued against launching.

They were afraid that something would go wrong. NASA went ahead anyway. And there have been lots and lots of articles written about why. But Tufte wrote one article with his argument for why, which I find reasonably compelling, looking at how was the problem communicated to the people at NASA who were trying to decide what to do about the launch.

Well, this is the information that the engineers had, pages and pages of numbers reporting what had happened historically in launches of these shuttles. And they created a visualization. This was it.

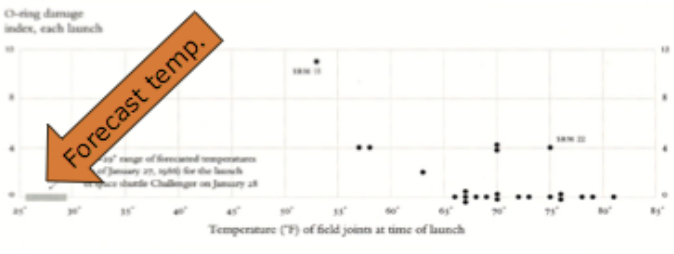
Basically in chronological order they listed each booster from its launch and marked the spots where there had been O-ring problems.



- **Visualization by Morton-Thiokol engineers**
- **Shows problems and non-problems**
- **Temperature not pre-attentive**
  - Need to look close and read

And if you look really closely at the display, you can see at the top of each one a temperature, what was the temperature when the launch happened. So the only way to perceive that temperature is to look really closely and to think about what's going on. You have to think about the pattern in there. You can't just look at it.

Now Tufte created this visualization of the relationship between problems in the O-rings and launch temperature. And here the problem leaps out at you, that launching in cold weather is a terrible idea.



So you've got this giant slope. As the temperature gets colder and colder, you see more and more problems happening with the O-rings. And you see that the forecast temperature is way lower than the lowest temperature at which they ever had done a launch before.

So this is screaming launching would be a terrible idea. You extrapolate that curve and you can predict disaster. Now there's no surety that this would have changed things. And there's been lots of arguments written back and forth about really who was at fault and where was the breakdown in the process that led to the launch.

But I think it seems pretty clear that a visualization like this, if it had been created, would have been pretty convincing to the people who ultimately decided to make the launch.

So that brings us to the end of our first segment where I talked about the field of information visualization, which really leverages the power of pre-attentive processing, this ability of human beings to absorb massive amounts of information with very little effort, with no conscious thought.

I talked about how the different visual variables, length, position, size, angle, color, really have different accuracies and are applicable to different types of information.

Some, like color, are really ideal for nominal information, sets of objects with no particular ordering or relationship. Others work better for ordinal information where we care about order but not specific values.

And still others work best for quantitative information where the exact numbers really matter.

And then, less studied is this whole class of relational information types, where what matters is the relationship between objects.

I've argued that the right sort of visualization can save lives. We saw the cholera map and the Crimean War map of the 1850s, all the way up to the Challenger disaster of 1987. And so what we saw is that the right visualization can save lives.

We also saw that the wrong visualization can be extremely damaging.

And given the power of visualizations, there a lot of things that can go wrong with them. They can really be used to deceive.

And so in the next segment I'm going to talk to you about what can deceive in a visualization and how you can avoid making deceptive visualizations.

Summarizing

- **Power of pre-attentive processing**
  - Absorb mass of information with little effort
- **Different visual variables have different accuracies for different information types**
  - Nominal
  - Ordinal
  - Quantitative
  - Relational
- **The right visualization can save lives**
  - Cholera, Crimean War, Challenger
- **The wrong visualization can deceive....**

6.3.3 Lying with Visualizations

Our next segment is a fun one, Lying with Visualizations. I've just talked to you about how powerful visualizations are. They're kind of like statistics that way.

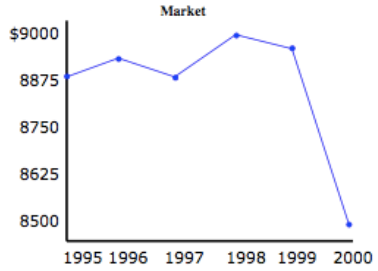
And so to that famous quote about there being lies, damn lies, and statistics, I will add that even worse than statistics are visualizations. They're even more powerful and that means that they're better for conveying truth but they're also better for conveying falsehood. And what one person thinks is a convincing visualization, well, to another person it's just propaganda that's actually hiding the truth.

There are various ways in which visualizations can fail to reflect correct values. And in fact, even when the numbers are right, technically, in the visualization, the eyes can deceive and fool the person who's trying to perceive the visualization.

So there's this famous book, How To Lie with Statistics. Sadly, nobody his yet written the famous book, How To Lie with Visualizations, but I think that they should, because it's just as rich a topic. So I'm just going to show you a few examples.

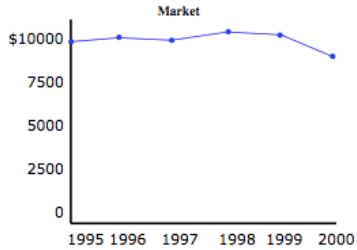
So here's a little plot showing the stock market crash of 2000.

Stock Market Crash?!



Oh my god, what a disaster, right? Well, not exactly.

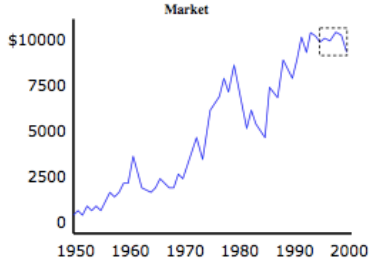
Showing entire scale



This plot, if we show the entire scale of possible dollar values for the market, we see that the crash is not quite as disastrous as it appeared in that first visualization.

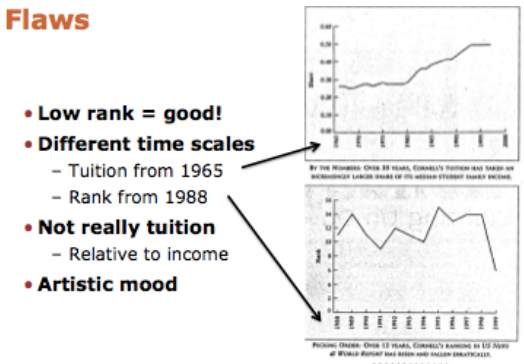
If we back up and show the entire date range over which the stock market existed, we see that it's even less significant in the grand scheme of things.

Shown in context



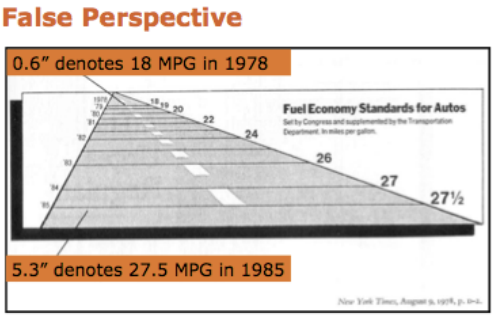
It's not even a particularly big decline. There are many others that predated it and have the same magnitude.

Here's a even worse play on the same thing. This is a magazine cover and it talks about how horrible it is that college prices are shooting up while college rankings are dropping like a stone at Cornell. Well, it makes for a very compelling picture, but let's look at what was actually going on.



First of all, that collapse in ranking. Well, remember, low ranks are good, so that collapse in ranking was actually an improvement for Cornell. They also use completely different time scales. So the tuition data in the first graph was ranging from 1965, while the rank was ranging from 1988. So trying to, sort of, connect them visually and vertically the way that we naturally would doesn't make any sense. There's also problems with the data. It's not actually tuition data that they're reporting but, instead, the amount spent relative to income. So really this picture is more art than visualization. It doesn't actually convey anything meaningful except, oh my god, I've got data and you should read the magazine.

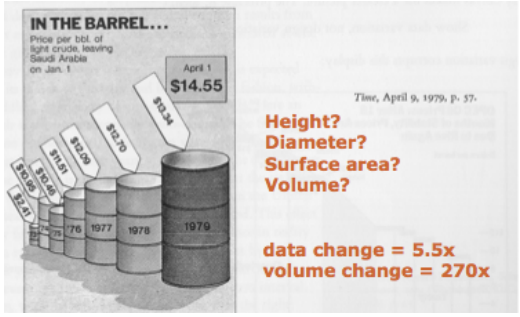
Here's another example. This one from the New York Times. And what this shows is the changes in fuel standards for cars ranging from 1978 all the way up to 1985.



MPG change: +52% Length change: +780%

And there's a problem with this chart, because we've got 0.6 inches of visible region corresponding to 18 miles per gallon in 1978. On the other hand, the 1985 27 miles per gallon takes up 5.3 inches of visual space. So what's going on is, they tried to make this pretty with perspective, but, by doing so, they drastically amplified the change in length. And remember, length is one of those pre-attentive things. Instead of seeing the abstract notion that miles per gallon changed by 52%, we see a 780% length change.

So it's conveying the wrong thing through this perspective. A similar problem emerges here where, even if they weren't using perspective, they're trying to use the size of the barrel of oil to represent increases in cost.



And you can see that the barrel in 1978 is just way, way bigger than the barrel in 1973. This is from Time magazine. Well, it's actually not clear what size matters in this visualization. So the actual change in price was a factor of 5.5, which, of course, is substantial. However, the change in volume from the smallest barrel to the largest is 270 times.

So the visual variable that they used to represent the quantity was just the height of the barrel. So the final barrel is five times higher than the first barrel. But what we see is the area, and what we perceive is the volume, thanks to the perspective and the shading.

So again, even though in this visualization there is the right number, that 5.5 times factor coming from height, we don't read it properly, and so we're deceived by this visualization into perceiving a change of a factor of 270.

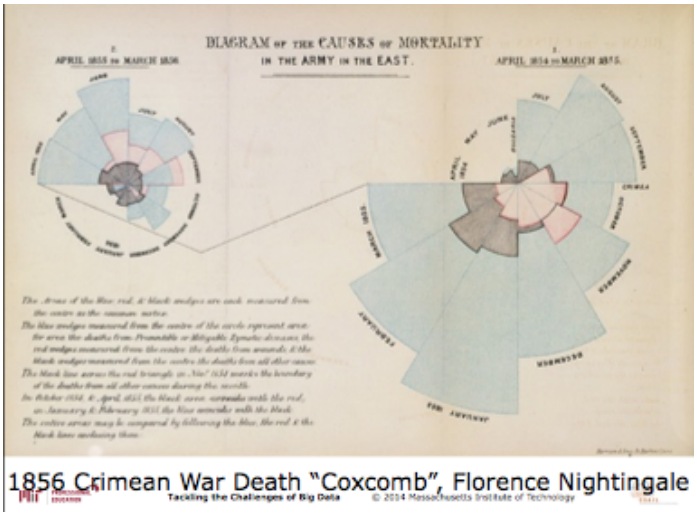
So this issue of, we've got multiple dimensions, we've got height, we've got area, we've got diameter, we've got volume, which one matters? That's a common bugbear in the creation of visualizations and something that you want to be careful about.

And the general rule is that you should try not to use multi-dimensional artifacts to represent quantities, because then there are too many different possible interpretations.

So if you want to represent a number with a visual variable, use a length, use a height, don't use both.

So Florence Nightingale didn't actually have the benefit of all of our decades of research on information visualization, and so she sort of fell into this a little bit.

The visualization that she created for Crimean War deaths used area as the visual variable to represent the number of deaths. And so I think that this actually means that, for us, this visualization loses a little of its power, because we have been, to some extent, trained to pay attention to lengths and not to areas.



1856 Crimean War Death "Coxcomb", Florence Nightingale

And so for example, here in November, we see in the Crimean War map that the radius of blue area that represents deaths from disease is kind of twice the radius of the deaths from other causes. But in fact what Florence Nightingale was saying was that it's a factor of four times as many deaths from disease. She did another thing which we also have learned not to do, which is, she's actually stacked these colors on top of each other.

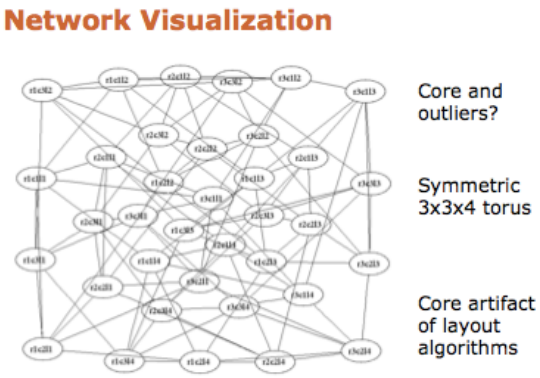
So underneath this pink, there's more blue that she's trying to represent. And so she actually has to explain in these words over here what's going on in this visualization and say that, well, you can't see any red here, because it's underneath the black. So she needs to explain all of that.

Nowadays, we would tend to stack things instead, in order to make it clear what the relationships were. All right, so, these kinds of fallacies that I showed you in print, they show up in our computer visualizations as well So here's a recent compelling picture of what happened when the conservatives came into power in Canada and cut welfare benefits.

And that's the line, and that's mapped against deaths of homeless people, which are represented as a bar chart, where, quite artistically and sadly, the authors actually stack the names of the people who died. And so there's a pretty strong visible correlation to what's going on here. And I don't disagree with their taking this artistic license, but it is worth noticing that there's some playing around with the scale here, in the sense that the top of this curve is at $10,000 and the bottom is at $7000.

So if we had used the full scale, as we did with the stock market, then the size of this decline would be much less substantial.

Here's another example. And I decided to draw something from relational data visualization. So nowadays, people like to take a look at social networks and understand what the clusters are and what the outliers are and so forth.



So this is actually a visualization that I created of a particular fake social network. And looking at this visualization, you can see what look like a core of people in the center and then a bunch of outliers along the edges.

But that is entirely an artifact. In fact, the data here is a symmetric torus, so every item has exactly the same relationships to neighbors as every other item. And this apparent core in the graph is a pure artifact of the fact that you had to take this three dimensional object and squash it onto two dimensions.

And so this is what the software chose to do. So again, the visualization is deceiving.

Here are some general rules for how not to lie.

One thing you should do is ***show the entire scale***, right? We saw this with the stock market example. If you chop the scale, it can make differences look artificially huge.

Show the data in context, show the entire date range instead of just homing in on some weird thing that happened in the middle.

Use a ***consistent scale***. So another thing that those visualizations we're doing, that I didn't talk about, was they actually didn't use an even spacing of dates in what they were showing. If you have things like inflation going on, you might need to use a log scale to get things to become linear, but you should get some sort of linear scale on which to show things.

And most important, perhaps, is to ***use visual variables appropriate to the data***. So don't use something like a length or a hue to falsely imply a quantitative or ordinal relationship when one doesn't exist when the data is nominal.

And finally, be very ***careful with size encoding***. Use height or width and not both. And avoid things like area and volume encoding.

Tufte is one of the great names in information visualization, and he coined the idea of a lie factor, which is to compare the size of an effect, a perceived effect in a graph, versus the size of the difference in the data itself.

Lie factor = size of effect in graph / size of effect in data

And what you want to do is try to keep the lie factor small. You should keep it at one, and have the effect in the data be the same as what is perceived. And he gave some general principles for how to do this.

One is to make sure that the representation of numbers as you perceive them should be directly proportional to the numerical qualities that you're representing. So this says, get away from things like volume, avoid problems like volume encoding.

You should also very clearly label everything. Don't leave people to draw conclusions about what the axes mean. Write out explanations of data on the graph itself, mark important events, show data variation not design variation. So leave the art to the artists. Don't think first about making your visualization pretty. Think about making it communicate what you want to communicate. And that often means stripping out excess detail, which would be pretty, but would be deceptive.

When you're dealing with money, again, it's often a terrible idea to use nominal units because of inflation, money in the past meant much more than money in modern times, and so you should deflate your currency visualization to reflect that.

Also, make sure that you don't add meaningless dimensions to your visual data. If you only have one dimension of numbers, then don't use two dimensions to display them, because, again, it can suggest relationships that don't exist.

And finally, don't quote your data out of context, as we saw from those examples that were looking at very limited date ranges.

So I'm going to finish with another example, which is the stop and frisk controversy that blew up in New York over the past couple, New York City over the past couple of years. There's been a lot of back and forth debate about whether this stop and frisk policy is justified, whether it's reducing crime, so on and so forth.

WNYC came out with this very powerful visualization of the history of stop and frisks to make this argument that it was not a good idea, that it was not effective. They created this plot where they took all the different counties, or sorry, not counties, but tracts, a certain sort of census tract within the city, and for each one, they color coded it on a scale where blue represents very few stops, and it scales up to a pink that represents a very large number of stops, and sort of a purplish shade in between.

Then on top of that, they plotted using green dots every stop and frisk incident where a gun was found on a suspect and confiscated. And looking at this map, it seems that there is basically no correlation whatsoever between the places where stop and frisk was most heavily applied, and the places where guns were found and confiscated, suggesting that this was a terrible idea. OK, it's a very powerful visualization, but it's not clear if it is a truthful visualization.

Here's another one. Somebody saw the WNYC plot and made a very similar kind of color map, but they changed a few things. First of all, they didn't make this artificial break down by census tract, instead they used a sort of density plot. Instead of using a two-color scale, we know this two-color thing can be problematic because colors don't convey quantity so well. So they used a single hue and just varied the saturation from white to dark red, and they did sort of a smooth density plot, which sort of gets darker the more stop and frisks there were in a particular area. And now, strangely enough, it seems likely the gun seizures are exactly correlated with the density of the stop and frisk activities.

So just a change in the way that you visualize, draws you to a completely different conclusion about the effect of stop and frisk. Taking this to even further extremes, here's another stop and frisk map that was created, where they've dotted each stop and frisk incident by the race of the person who was stopped, or the ethnicity of the person who was stopped.

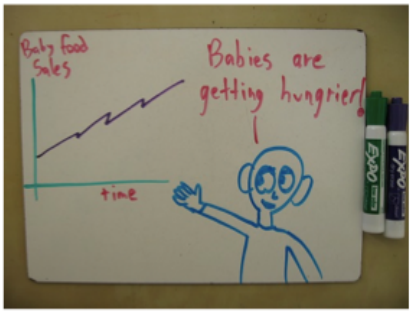
And this can be used to argue that there's a significant racial bias in the stop and frisk. Blue dots in this map represent African Americans, or black ethnicity, and there are certain regions where it seems like only blacks were being the stopped and frisked, so this raises issues of race.

Here's another visualization which seems to take that even further.

It says forget about the geography, let's just look at the race of the people who are being stopped and frisked, and you can see that it's very, very skewed.

So all of these different visualizations, starting from the same data, suggest different things. And it's very important to remember that none of what they suggest is de facto truth. What they show is truth, that more blacks are being stopped than whites, but there's a lot of context that is not conveyed in this visualization. And you have to think about that context when you want to think about what this visualization is saying. Even when the data is exactly correct.

This is from mathwithbaddrawings.com.



http://mathwithbaddrawings.com/

You can look at the visualization, you can see what the numbers tell you, you can yes, these numbers are shooting up, and draw entirely the wrong inference.

So if baby food sales are going up over time, does that prove that babies are getting hungrier? Who knows? It probably depends on how many babies there are, right?

So you have to be careful about the inferences that you draw from your visualizations and understand the context of the data. So summarizing this segment, visualizations are very powerful for communication, and for that reason you have to use that power wisely. You should ask what your visualizations imply that isn't true, and try to keep that from happening.

Ways to do that include showing the entire scale, showing the data in context, and picking the right visual variables for the types of data that you're showing.

Leave art to the artists, and concentrate on conveying just what the data says.

On the flip side, you should be skeptical of visualizations that you see because other people may be making these visualization mistakes.

So looking ahead, in the next segment, I'm going to be talking about interacting with visualizations. A visualization creates lots of questions,

Whether something is true or an artifact ?

Why something is happening?

What if something changes?

If the visualization is static, these things are hard to answer.

But interactivity can be a big help. It provides a way for you to look at data from multiple perspectives, to change the data and see what happens to the visualization, to explore and test hypotheses.

And this makes it better for analysis because you can get more information.

And it actually also makes it better for communication, because if somebody can actually poke at your visualization, it can become more convincing.

So that'll be the subject of our next segment.

6.3.4 Interactivity : Exploring Data

Our next segment is on the idea of exploring data through an interface. So, as I said at the end of the last segment, you can look at a visualization and wonder a bunch of things.

Is this real?

Why is the data this way?

What if the data were changed in the following way?

And if the visualization is static, they're really hard to answer.

Interactivity can be a big help. If you can look at the data from multiple perspectives, make changes to the data to see what happens to the visualization, you can start to test hypotheses and understand the data better. This is certainly useful for analysis, because you can get more information about your data.

And it's also better for communication, because a visualization that can be poked and prodded can be more convincing than a visualization that just sits there.

So another way to look at this is an interaction that means you're not limited to a single data visualization, but instead provides a way for you to navigate a huge space of possible visualizations until you find the one that tells you what you want to know.

Now what I'm going to be talking about over the next few segments is, first, the mindset for this, this idea of exploratory data analysis, the sort of roots of the ideas that have led us to data interaction interfaces. Then I'm going to talk about a very powerful interface tactic called ***direct manipulation***, which is really the right input mechanism for most people for doing interaction with data.

And then I'll show how this direct manipulation technique is used in a higher level strategy, which interfaces should support, for users to get an overview of the data, then be able to zoom in on certain parts of the data, filter it in various ways, and then get details about specific data items on demand.

First, I'll talk about exploratory data analysis.

So this was really pioneered by John Tukey, he of the great quote about pie charts. And he wrote a wonderful book on exploratory data analysis. And he got his start in the '50s, I think, before computers were really dominant.

And so he initially sort of devised this idea of exploratory data analysis with pencils and paper. But he also developed the first exploratory data analysis computer system called PRIM-9, which I'll talk about in a minute.

And he really teased out this idea that there are two different kinds of data analysis (Confirmatory, Exploratory) that you might want to do.

Confirmatory data analysis is taking a hypothesis that you have and testing it against your data.

So testing whether your data fits a certain distribution, testing whether people who have this treatment come out differently from people who have this treatment.

But how do you develop that hypothesis? If you're designing an experiment, you start with the hypothesis and then you design the experiment and see whether your data fits that hypothesis. But at other times, somebody just gives you some data and says, here, what can you tell me about this?

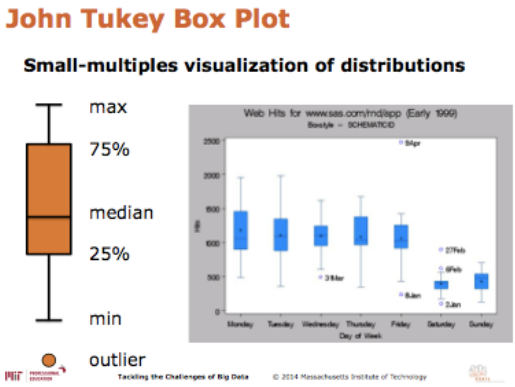
And so you don't have a starting hypothesis. You need to explore that data in order to develop the hypothesis. Now I will say that visualizations are a really bad idea for confirmatory analysis. For the reasons that we saw before, it's too easy for visualizations to deceive.

However, visualizations are great for suggesting hypotheses that you might want to test to give you ideas about what's going on with the data so that you can then go off and try and see whether that data fits a certain model.

So Tukey gave, I think, a beautiful description of this. He said, "If we need a short suggestion of what exploratory data analysis is, I would suggest that it is an attitude and a flexibility and some graph paper or transparencies." Of course, we don't use those anymore.

No catalog of techniques can convey willingness to look for what can be seen, whether or no anticipated. Yet this is at the heart of exploratory data analysis. The graph paper and transparencies are there not as a technique, rather as a recognition that the picture-examining eye is the best finder we have of the wholly unanticipated." So this again is about pre-attentive information.

We can perceive so much by looking at a visualization. It can really give us ideas, show us patterns, tell us about things that we should be looking for using other techniques. So as one contribution to this idea of exploratory analysis, Tukey came up with this lovely visualization call Box Plots.



This is another example of small multiples of this idea of taking several different collections of data and looking at them side by side.

Here, each collection of data is a set of numbers or distribution. And Tukey had this idea of representing each such distribution using this box plot on the left, where you put a box that goes from the 25th percentile to the 75th percentile of the numbers.

You draw a line at the median. Then you draw these whiskers that stick out and go to the minimum and maximum values in the distribution. You might also pull out some dots that represent the most significant outliers from the data.

And so as an example of how to use this, you can take a look at the plot on the right, which shows day by day the hits on a particular website.

And again, through visualization, it's very easy to see that the weekend has just a really different behavior than the rest of the week in terms of accesses.

You can also see, oh my god, something must have happened on April 9th. It's very much an outlier from the general distribution of values on a Friday. But that was only one contribution.

A really major one was the development of this computer tool for exploratory data analysis called **PRIM-9**. And ***PRIM stands for project, rotate, isolate, and mask***, four sorts of operations that people should be able to do with their data in order to explore it.

Projection is just taking high-dimensional data and mapping it down to two dimensions for display on the screen. But the key insight was that you don't just want a static projection, but instead you want to be able to rotate the data along various axes in order to look at it from different directions.

Isolation and masking where the idea that you want to be able to filter down the data in some way. When you see something interesting, you want a home in on a particular set of points and then go back to projecting and rotating in order to understand what's going on with that particular set of points. And, in fact, the system supported these operations through direct manipulation.

You didn't type commands to mask and rotate. But instead you used your keys and things to directly manipulate the data. That'll be the next segment.

So here's a short two minute video of John Tukey demoing the PRIM-9 system. There's a full 20-minute version of this that you could look at. It's kind of amazing how much forethought this system demonstrates that John Tukey had back in 1972. So I encourage you to take a look at the whole thing.

JOHN TUKEY:

This four-dimensional physics data, which you will also see again shortly, seems rather structureless until we rotate it. To take advantage of this kind of structure, a simple thing to do is to mask out some of the points, specifically to mask out all points, one of whose current coordinates fails to satisfy one of a family of simple inequalities. We can see what this does with a new example. This video shows what happens when the masked coordinate is visible and the lower cut off advances from the left steadily and then retreats.

If the masked variable is not one of the two we are looking at, other interesting things can happen, as we will see when we mask the same data on its third coordinate. Again, as the mask retreats, we come back to the original view.

Now we would remember John Tukey for this, except that then he went and invented the Fast Fourier Transform, which is probably one of the most important contributions to modern computer science and engineering. He also came up with the name with the word "bit" to describe the things that we work with, and a variety of lemmas and things.

So I don't know. I think people should know more about what he's done in exploratory data analysis and visualization, but it's just a part of a huge oeuvre of work that he managed to do.

I was kind of amazed to discover this, because actually, by total coincidence, John Tukey was my second co-author ever. And at the time I had no idea who he was. But it's amazing to discover these things about him.

So to summarize what we've talked about here, I've introduced you to this idea of exploratory data analysis, looking at data without preconceptions in order to see what's there and develop some hypothesis, and the important distinction of this from confirmatory data analysis, which should not be done with visualization, but should instead be done with formal models and tools that actually do mathematical tests.

When you're doing exploratory visualization, I talked about the benefits of trying different visualizations, and I've argue that the computer provides great support for this. As you do this, you should remember that visualizations can deceive, and so you should use non-visual tools to confirm what you see in your exploratory analysis.

So in the next segment, I'm going to start talking about the kinds of interfaces that can help you explore data.

6.3.5 Direct Manipulation

Now we're going to talk about a particular interface technique called direct manipulation, which is really effective for data exploration. And just like when we talked about information visualization, we wanted

people to be able to perceive without thinking, when it comes to data exploration, we want people to be able to explore the data without thinking too much about how one runs the interface. So information visualization used pre-attentive visual feature things that came straight into your head perceptually.

What's the analog for interacting with data? So we can start to get a handle on this by looking at the spectrum that I created before of the tools that are used by professionals, things like programming languages and databases, versus the tools that are used by everybody else for interaction.

If we look at the differences, we can understand that advanced users tend to think rather abstractly about the problem, and they have a language for describing the sorts of things that they want to do. And if you tell them that they have to write things down in a description using that language, they can do that.

They can program. They can describe sequences of actions. And if those sequences don't work, they can debug, and they can try things again. And in order to do that, they're willing to read the manual, to be told what the syntax should be for typing those database queries or writing those programs.

Amateurs and novices, on the other hand, tend to think concretely. They've got a specific collection of data and they want to do something with it. They may know what they want to do, but not what to call it in technical terminology. In general, I think they can show what they want to have done, but they can't tell. They can't use any formal language for describing it. If something goes wrong, they don't know what they did wrong. So they are going to have trouble debugging some sort of sequence of operations. And they don't read manuals. They learn by doing.

So if we look at the kinds of tools that amateurs use, they're what we might call direct manipulation tools. And I'll formalize this later, but starting to draw the distinction, while advanced users will work with a command line to manage their files, amateurs and novices like to use a desktop with icons for files and folders that they can drag around.

Advanced users like hot keys while novices need to use menus to find the commands they want to invoke. Advanced users will write source code in HTML or LaTeX or in WikiText, whereas amateurs and novices prefer WYSIWYG editors.

Advanced uses write programs. Amateurs can't do that. They can record macros. They can show what needs to be done, but they can't tell.

Advanced users will write SQL database queries, whereas amateurs do their data management in spreadsheets. Advanced users, like me, like computer games like Zork and Adventure, where you talk to the computer and tell it what to do.

Amateurs tend to use things like Pong and Space Invaders, where you have funny controls that manipulate things on the screen.

So the direct manipulation paradigm and this was put forth, I think, first by Ben Shneiderman in 1983, is a number of ideas about making interaction with the computer feel more like manipulating physical things. So you want to have a continuous representation of the objects and actions of interest. You don't want somebody to go and write some program and then come back and magically have things happen.

You want to have the data sitting there, and as the user acts on the data, you want the data to change based on their actions. You want those actions to be physical, scrolling, pressing buttons,

not writing some complex syntax. And as they do these physical actions, you want the effect on the display to be rapid and to be incremental and reversible, so that users can see what happens when they do something. And if they don't like what happens, they can undo it quickly and try something else.

So if we go back to this sort of differentiation between advanced users and amateurs, you can see that this direct manipulation interface paradigm really fits well with what amateurs want to do.

Amateurs think concretely. Direct manipulation interfaces just show them the data that they want to explore. They know what they want to do, but not want to call it, so you should show them the available actions, not require them to know some magic syntax for describing what should be done.

And you should give them immediate feedback when they try something. Again, these physical interactions direct manipulations of the data, are a good way to show what should happen instead of telling.

Having it be reversible is a great way with dealing with the fact that they don't know what they did wrong. They can just undo it and it doesn't matter what they did wrong. And again, the sort of explicit presentation of buttons and things that you can move around, those allow end users to learn by doing because they can discover what can be done.

Now, there's always a trade off, and direct manipulation has that trade off. In general, programming languages and command lines are usually more powerful than direct manipulation interfaces.

You have to usually sacrifice some power in order to make things direct manipulation. But direct manipulation is always easier to learn and use, and even experienced users prefer direct manipulation when they can get it.

We all do like to use a WYSIWYG editor when we can. But they resort to programming when the direct manipulation tools aren't powerful enough. Now if you don't believe me, let me suggest to you that the web is perhaps the triumphant example of direct manipulation.

Because what could you do on the web? Well, the web is a tool that lets you look at a document that describes access to other resources and allows you go off and fetch those resources from somewhere else. Well, by that description, we certainly had the web back when I was in college in 1987.

I could look at a document which described an FTP site. I would use the command line to launch my FTP program. I would type in the address of the site I want to fetch from. I would type in the name of the content that I wanted to fetch. I would then download it and I would then feed that content into a program that would display it for me. So I could do everything that the web provided.

What did the web do? Well, the web just made this direct manipulation. Now, instead of having to type all of these commands, I look at a document and I click.

So there's no typing. Instead, I click on what I want. I show and I don't tell. So the web didn't actually make anything new possible. All it did was make something much simpler that was already pretty simple.

I mean, it was just three FTP commands, but changing it from three commands to one click was this total revolution, and that was just an application of direct manipulation.

Let's look at direct manipulation interfaces for data. Here's an early example by Shneiderman of one of these direct manipulation interfaces.

This is one called Filmfinder. It shows a scatterplot of movies ranked on the x-axis by date, and on the y-axis by some measure of popularity. It uses color coding to display different types. And this is an appropriate choice, types of movies are a nominal quantity, different colors are great for presenting nominal quantities.

On the other hand, dates are a quantity, a quantitative value, and so they make good sense to use as something to plot on a number line. You'll notice that for popularity, they've numbered the popularities from zero to nine.

So they've made an ordinal, not a quantitative. But it still is something that you can put onto a scatterplot. So it's fine as information visualization, but they then added interactivity in this panel over here.

And what they did was they created sliders. So you can, for example, take this slider and slide, and this is an alphabetical slider, you can slide to a particular actor. OK, and I think I show that on the next slide.

Yes, so here, they slid the slider to select Sean Connery as an actor. And they've also used a range slider, which limits the length of movies that should be displayed, to be movies that are between 50 and

now, I've just covered it up-- but between 50 and 276 minutes. And this has filtered down the scatterplot to a smaller set of movies. And so that's some simple direct manipulation interactivity for filtering the data.

So I've already discussed this, but we had these two quantitative feature axis and one set of hues, colors, for the nominal movie types. And we had a number of different kinds of direct manipulation

buttons for selecting these categories, sliders for working with these ordinal names, a range slider for the quantitative length.

We also in this slide actually have the ability, although they're not leveraging it here, to zoom in on a smaller date rate, oh sorry, it is zoomed. So this is zoomed to range from 1960 to 1995, as compared to the original scatterplot, which started all the way back at 1920.

So you can zoom in on the data and filter the data in a variety of ways. Just to look ahead, I'll also mention that the initial view gives the complete overview of the data. So this is, again, you don't want to start users with a blank screen. You want to start them with a decent starting point, an overview of what's there, so that they can get a sense of where they can go.

And also, they leverage the fact that their selection happens in context. So you start with the whole scatterplot. When you filter down, you still see things in their relationship to where they were in the whole giant scatterplot. So you can remember that oh, these ones were in this highly popular section of the movies, and now I can see just which of Sean Connery's movies were highly popular.

So this, again, leverages in the visualization pre-attentive interaction, the idea that you can acquire information by site. The goal of direct regulation is input via, I'm putting in quotes "pre-attentive", because it hasn't been formalized this way.

By manipulating the data through the same sort of instinctual activities, the user obviously has to make a decision to do something, but they don't want to have to think about complex commands to type in order to do it. In fact, video games are probably the pinnacle of this. We know that professionals playing video games are reacting without thinking to what they see.

And their hands are doing things without their minds telling them to do those things. That is truly pre-attentive reaction. And maybe some day, our data interfaces will allow us to do that as well.

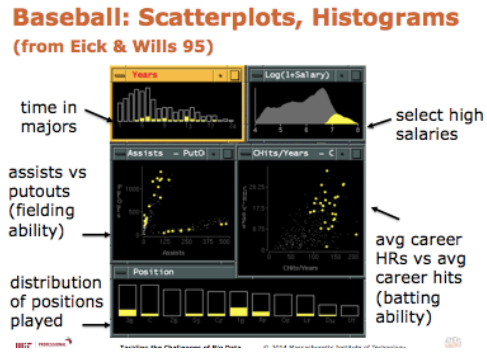
So I think that direct manipulation tools for data are as really as important as the desktop metaphor and WYSIWYG editors were for general user interfaces.

Let me show you one other example of direct manipulation, which is an idea called linking and brushing, which is a powerful technique for doing data filtering.

The idea is to take two different information visualizations and then allow you to filter by actually brushing over some section of one of the visualizations. This will become clear in a second.

So you select a subset of the data points in one visualization, and because the visualizations are linked, that filters down the data in the other visualization as well. And so this really means that your visualization is not only an output, not only something that's being presented to you, but also an input tool, where you can point at certain parts of the visualization and say, I want to filter down to just the stuff that I see here.

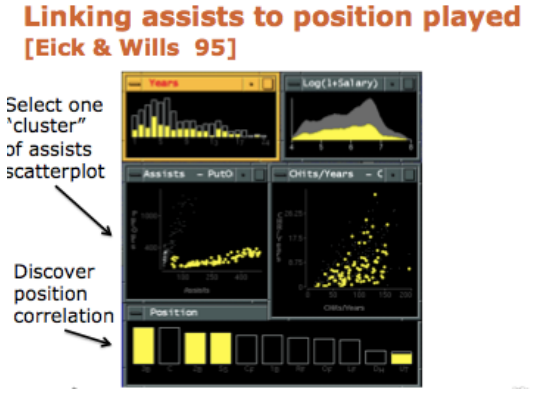
And so here are a few examples of systems that have used this brushing and linking concept. So here's one example of such a linking and brushing interface due to Eick and Wills. So this is a display of baseball players statistics.



We have a number of different plots showing different pieces of information. So over here, we've got the salaries of the baseball players. We've also got a bar graph histogram showing how many years they've been in the majors, scatterplot of assist versus putouts, a scatterplot of career hits per year, their batting ability versus their career home runs, and another histogram of their positions on the team.

And it's a little scary that we have to actually plot the salaries on a log scale, but that's the way professional sports are. And one thing we might want to do is understand how people with high salaries differ from other people in the game.

So using this linking and brushing technique, since all of these different plots are linked together, we can select just the high salary players. And when we do that, we're able to see their statistics in all of the other plots in context.



So for example, we can see that there's bimodal distribution on how long they've been in the major peaking at six years and at 13 years. We can also see, perhaps unsurprisingly, that the high-paid players are good hitters. They have high career hits and high career home run counts. But we can see that there's not so much correlation with their positions on the team.

So just to summarize these conclusions, if you look at this data, you really can't earn a high salary in the first two years, again, unsurprising. There's this bimodal distribution on salaries. If you look at the scatterplot, it seems that hits per year are a better predictor of high salaries than home runs per year right.

We've got a few dots here for people who have very high home runs, but didn't do particularly well on salary. But you might also see an outlier over here, somebody with a few home runs and only a medium number of hits per year, and ask why that is?

Why is this guy getting paid so much? And that's sort of is an exploratory moment-- you see an outlier and you ask what's going on.

And the answer is that this player is a player-coach. And so they're being paid well for other reasons. But another thing that you can see in this plot is this odd sort of clustering of assists versus putouts into two groups. And not being a baseball aficionado, I would've never expected to see this in a scatterplot and have no idea why it is.

Now, I'm going to let some of you who have more experience with baseball take a minute to try and figure out why there should be this partition of the players into two separate groups.

Now, if like me you don't know anything about baseball, you can't figure this out by thinking, but you can figure it out by linking and brushing. So the natural thing to do is to select one of these clusters and look at the data in context. And if we do that, it becomes completely clear what's going on. There's really no correlation to years in play or salary. There does seem to be a little bit of correlation in terms of hits.

But really, what's become clear is that third-base, second-base shortstop, they have a different profile than the outfielders and catchers.

So we have the defensive versus the offensive players, and they have a different spectrum of assists versus putouts. So that's what's going on with the scatterplot. So that's it a quick introduction to the idea of direct manipulation. And now that we've seen that, I will start in the next segment to tell you how we can take that idea of direct manipulation and use it as part of a strategy for doing a more broad scale interaction with data from start to finish.

6.3.6 Interaction Strategy: Overview, Zoom, Filter, Details

Our next segment is going to look at general workflows for interacting with data. And a mantra that has been popularized by Ben Schneiderman again with the idea that you start with an overview, you then provide tools for zooming and filtering. And then, finally, offer the ability to get details on demand.

So Schneiderman wrote a nice little article about this in '96. I'll provide you with a link to that. And it looks at these sort of four distinct concepts.

Overview: The first is the importance of giving the user an ***overview*** of all of the data as a starting point, not starting with a blank screen and asking them to do something.

Filter: Then providing them with mechanisms for ***filtering***, ways to restrict to interesting set of points.

Pan/Zoom: Then giving them ability to ***pan*** and ***zoom***, to hone in on an interesting area, and get more details about it.

Details: And finally, the ability to get details on demand. You can think of this as an extreme zoom or you're looking at one point, get everything, every detail about that point or a few items.

Now, I'll quickly go through the idea of an overview, the fact that you don't want to start with a blank screen. You want to give people an idea of what the data is and what they can do. And so it's worth thinking about what a user's first impression should be, when they launch up your tool.

We've also already seen quite a bit, when it comes to filtering. In the film finder, we saw buttons for filtering on type, sliders for choosing ranges of values. And this idea of linking in brushing is a powerful way to do filtering of the data.

We haven't talked so much yet about panning and zooming. This is really a powerful kind of direct manipulation filtering that leverages a spatial metaphor. So again, we have a strong sense of position, and so the idea of moving through a data set is very natural.

We all understand scrolling as a way to bring new data into view and move the old data out of the way, so that we have room for the new data. In multiple dimensions, this is referred to as panning.

And so the idea here is to apply the features that have been mapped to spatial coordinates. And so now a shift in spatial coordinates becomes a shift in the parameters that you're filtering.

Another benefit of panning and zooming is that it provides a sort of an animated transmission. So as you pan, you see all of the points moving from their old place to their new place. And this helps the user preserve the context of what they're looking at. It's far better than having a sudden flash where the points all rearrange into a completely different arrangement.

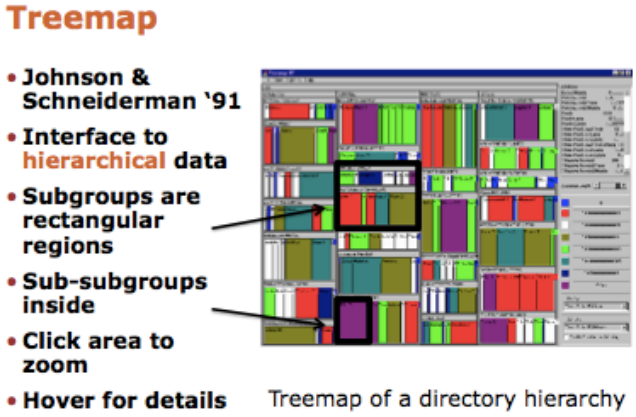
So this has obvious applications to spatial data, to things like maps, but also applies to more abstract visualization, to visualizations of more abstract features. So most obvious example, as I said, is spatial.

We have all of these capabilities in Google Maps. After I open up a map where I've done a particular search, Google will provide me with a presentation of all the points where there is something that matches the search. I can now pan, I can slide the map around in order to look a regions that are not currently on the screen.

So what I'm doing is changing the spatial filtering of the latitude and longitude that I want to observe. I can also zoom in, getting more details about a smaller spatial area. And finally, if I click on a particular dot then that pops open something and gives me more details on demand about that particular dot. Obviously, you can't have all of those details open at the same time. They would completely obscure the map. So that's something that you provide on demand.

So as I said, the map is the most obvious spatial thing that you might want to pan and zoom. But this can be applied to more abstract concepts as well.

And so here, is an early example of Johnson and Schneiderman's tree map, which is a way of navigating hierarchical data.



So this is another type of data that I haven't really focused on yet, but that we all experience in our directory hierarchies. We've got a folder, and inside that folder, there are more folders. And inside each of those folders, there are more folders.

So how might you get an overview of everything in all of your folders? OK, well, Johnson and Schneiderman came up with one way to do that. And so they created a tree map of a directory hierarchy here. The idea is that you have your entire collection, your top level folder. And then you create these boxes representing the subfolders of the top level folder.

Then, inside of each of the subfolders, you create a box for the sub-subfolders. And you get this kind of mosaic. And the regular even layout of the top level boxes in the containment of all of the subfolders inside provides you with an understanding of the containment relationships in the hierarchy.

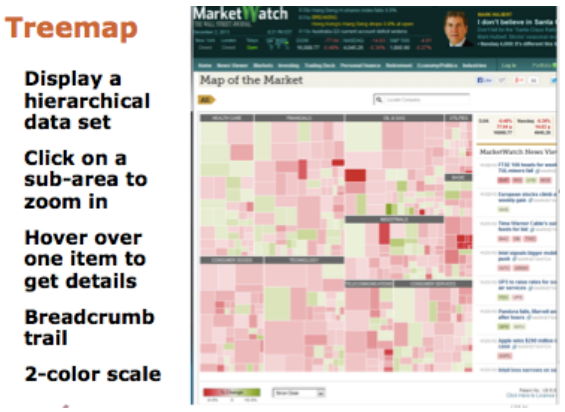
But notice that here you are looking at an entire directory hierarchy, and not at just one folder.

Furthermore, you can take a specific subfolder, like this one. And if you click on the heading of the subfolder, you will zoom in. You'll get this animated expansion, and you will see more details about the subfolder that you're looking at. So this was the early 1991 version.

This is actually a lovely example of a transition of research into practice, because if you now go to the market watch website, you'll discover a map of the market.

And so what we have here is a visualization of the stock market by sector and by individual companies within each sector. And we've got a color coding here. This is actually an improvement on the original Schneiderman example. So here the colors didn't actually have a lot of meaning and were somewhat confusing.

But here, we've got a nice and natural two-color scale, shading from red to neutral to green to represent stocks that have gone up in value, and stocks that have gone down in value. Now, before I said that multiple colors are better for nominal values, but here is one exception.



When you have a clear zero, then it's very natural to have one color to represent increases, and a different color to represent decreases. And to then use saturation within each of those colors to represent the amount of increase or decrease.

So that's what's going on here. And we can see that this is a relatively bad day for the market, because most of the sectors are pink, and most of the individual companies are pink.

So I have sectors like health care and financials, and each of these gets one big box in the visualization. But I can go and click on one of these headers, like financials, no, I think actually picked technology. And if I click on that sector, so now within that sector, it now subdivides into subsectors. So technology is now divided into software and hardware. I guess all technologies computers now, so there's nothing but computers and technology.

But again, now I have two separate sectors or two separate subsectors, each displayed in its own tree map. If I now hover over one of the items, I get details on demand about that particular item. Again, I can't show you the details about every single item at the same time, but there's room to show details about one, if I ask. Another useful trick to look at here is what's called the breadcrumb trail.

So here, there's a little sequence of what were the things that you did. This is quite useful to help me remember what I did to understand what my current state is. It also provides an easy way for me to reverse my operation. If I click on all, it takes me out of the particular subsector and takes me back to the original visualization.

And we've already talked about this two-color scale and its benefits. That's a quick overview of this general strategy for interacting with data, where you start with an overview, then you provide tools for zooming and filtering, and then, finally, offer details on demand.

And now, that we've seen that in sort of the abstract, and a particular example of tree maps, I'll now show you how these ideas are explored in the context of who did the web and the data websites that we visit every day.

6.3.7 Data Interfaces on the web

Now that we've talked abstractly about some of the principles behind the design of data interfaces, I want to look at the practice of how data interfaces are designed on the web.

A lot of the websites that we visit every day are basically front ends for giant data sets. And a common paradigm has emerged for presenting these data sets and letting users explore and interact with them.

It's useful to have this paradigm, not just because it's a good paradigm, but because users have come to expect it. And so now they know what to do and what to look for when they visit a brand new website.

And this paradigm really nicely matches Schneiderman's workflow that I described in the previous segment.

Pretty much every website has developed certain kinds of ***views***. An overview of all of the items in their collection for a user to look at.

They've also offered a technique called ***faceted browsing***, which is a very natural, direct manipulation technique for filtering the data that the user is looking at.

And finally, they leverage the uniform structure of the data in ***templates*** that provide a uniform presentation of the individual details of items that are on the websites.

Views

So starting with the views, pretty much every website that we go to has devised one or more aggregate views of the information on the site.

So with Flickr, we've got a grid of items. The Boston Logan Airport has a plain old table of arrival information. On Yelp, you've actually got a pair of views, one of which is a list, and another of which is a map of restaurants, resulting from a given query.

So these are aggregate views of the information that the user can skim in order to get an overview of what's going on.

Looking at that Yelp page, you can also see the use of templates. So this is a sort of view of the details of any particular item, and again, it leverages the uniformity of the data. So at Yelp, we know that every restaurant has a name. And we know that that name is always going to be at the top left of the index card for that restaurant. Similarly, every restaurant has a little picture associated with it. Every restaurant has an address. Every restaurant has a rating.

And these are always put in the same place in a regular arrangement, which makes it really easy for us to scan, and for our eyes to naturally home in on the information that we want. In the map view, the default display is purely an overview display, but again, if you click on a particular point, you get another detailed view, which again, is generated by a template.

And you'll notice that it's not quite the same template as is used in the list. In this template, the photograph is on the right, and there's less information than there is in the list. But every single item that you look at in the map view is going to be presented the same way, with the same benefits.

So templates are great, because they give this uniform presentation. They're easier for us to parse.

We quickly get used to what attribute goes where, and so we're, again, leveraging the power of position in being able to receive information quickly. It also just feels pleasant for all of the items to be presented the same way. It's elegant. It's uniform. We could show the same information in the table, like Boston Logan Airport did, but in general that's considered kind of ugly and difficult to deal with.

These templates are generally generated by a templating engine, and templating engines, we haven't talked much about architecture in this unit, but, templates make it really easy to maintain your website.

If there's a change to the data, you just change the data. You don't have to write any new HTML code. The templating engine will present the new data same way as it presented the old data.

But the thing I really want to focus on is this idea of ***faceted browsing***, which really has become dominant on the web.

And we have an example here, again, on this Yelp website, of letting you check off, you can filter on which features are available at the restaurant, or which neighborhood the restaurant is in, or which category of food the restaurant offers. And all you have to do is check off a few of the values, and the site will filter down to only present the items that have those values.

And so similarly, we can look at Amazon, and they put their facets on the left instead of on the top. But you can filter on format. You can filter on author. You can filter on book series. You can filter on when the release was. And again, the view, the list view that Amazon is offering, will update to show only the items that match the values that you filtered.

So this is really the dominant filtering paradigm, where you give users a collection of these facets. Within each facet, you have a set of values associated with the particular attribute. They pick some, and the collection restricts to the items that match those values.

Now, this is actually a change from the earliest days of the web. If you go back to the good old days of the original Yahoo, how it got its start was as a giant web directory. And that web directory was a hierarchy. You had the top level hierarchy, and then inside of that hierarchy, you could navigate to a sub area, like arts. And then within arts, you could navigate to a sub area, like literature. And the underneath, you have all of these subcategories, like American literature, British literature, children's literature.

And so this was, again, sort of coming off of the files and folders notion of our desktops. Made a lot of sense to try and categorize the web the same way. But this has some real drawbacks.

This forces the user to obey your decision about what the hierarchy is. They can go up and down.

But if they want to filter, if I just want children's stuff, so that means children's literature, children's art, children's clothes, there's no way for me to filter that. I'd have to descend once to find children's literature. I'd have to go up and descend again to find children's art. I'd have to go up and down again to find children's clothes.

I can't use filters that don't match the hierarchy. So in a sense, these hierarchies provided a way to zoom, to go down into more detail on a specific area, but no real general notion of filtering.

So faceted browsing, and this idea of facets really gets us past this, right? It provides a direct manipulation, where you can see what all the different axes of filtering are and make your choices within each of them independently.

Another benefit of these facets is their direct manipulation nature. If you don't like the results that arose, you can undo whatever it was that you did with the facet. So it meets the direct manipulation paradigm.

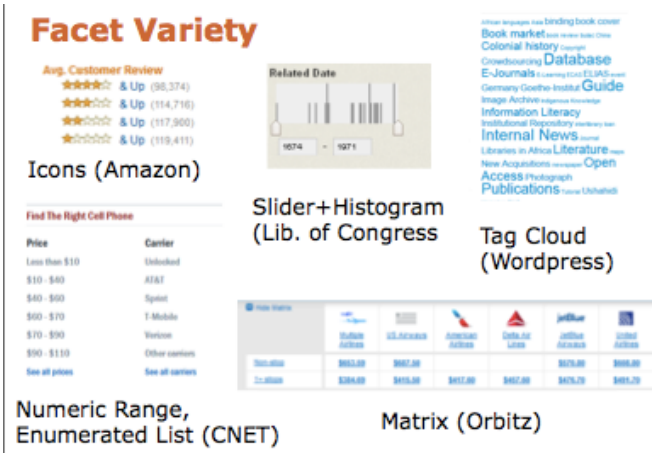
But unlike a hierarchy, you can make your selections within each facet independently. So you're not restricted to what the creator of the site decided was the right order of application of restrictions.

And you can apply the facets in any order. The result is just an intersection of all the facet constraints. Again, this is something we're used to now that if I use this facet and this facet, it combines the restrictions. But the final result is independent of the order.

There's no memory of how the user got there.It's just based on the current facet values. So again, it's a very simple mental model of how you're filtering the data. And so that works well for end users. So there are a lot of different kinds the facets you'll see on the web.

So here at CNET, for nominal data types that are just lists of values, you can just make a list of values. We also saw that on Yelp for the different types of restaurant.

Now, small details, CNET allows you to just click on the thing, whereas yellow pad check boxes that you would fill out minor stylistic change and here's another type of asset when your values are numbers you might not want to list all of the numbers, especially if there's too many of them, if it's sort of a continuous range, but instead, you can give numeric ranges for people to select, right?



Another way to deal with numbers is to create a slider facet. We saw this in the film finder. This particular slider facet has a nice added feature that actually gives you a histogram of the values, so that you get a prediction of how many items you're going to get if you move the slider facets around. That's quite useful.

Amazon turned their facets iconic. So this is just a facet for selecting between one and five stars, but they made it pretty with pictures.

At Orbitz, you have this interesting matrix facet, all right? So when you're searching for flights, you can choose to filter on the number of stops by clicking on one of these facets. Or you can choose to filter on the airline by clicking on one of the columns. Or you can specifically choose an airline and a number of stops by clicking within the matrix.

And then, of course, there are the popular tag clouds, right? So this is a facet that shows the number of keywords associated with different items. And by clicking on a few of the keywords, you can filter down to items that match those key words.

So these are all different presentations of the same idea of a facet.

There are some common practices that have emerged, like ***hiding values that yield no results***. So you don't show all the values of a facet, you only show the values that still exist in the collection as it's currently filtered. That helps the user avoid running into dead ends.

Even better than just hiding the facets that shows your results, you can do things like I showed here with a histogram, give the user a sense of how many results you're going to get if they click on a facet.

So if you have a list, you can put some counts of how many items there are associated with each value. And that, again, is useful sort of forewarning for the user of what's going to happen if I use this facet.

And we also already saw the ***breadcrumb*** shell idea, right? You can show user what facet selection they made and let them undo those facets selections as a way to reverse their practice.

So this kind of summarizes, I think, the design of almost all data interactive websites, right facets, views, and templates. It's nice to have this paradigm. Users know what to expect, so that they know how to interact with new sites. Now, you may ask is it really true that all data websites are done this way?

So here's a quick tour.

Here is Epicurious, a recipe website. What do they provide? Well, they provide a list view with templated items. The list is sortable, that's a common thing that you offer for lists. They have a tech search box, which I haven't talked about, which is pretty obvious to add. And they have faceted browsing. So here I'm selecting a season for my recipe.

Here's eBay. What do they offer? Well, they offer a list of items, they offer sorting of that list-- each of the items is templated. And there's facets for filtering the items in various ways along with tech search.

Flickr. So this is one Flickr site design to go. But they use view of items as thumbnails, but there's a template for each item, where you show the picture and the title. There are details on demand using a template. You can sort by various properties. And there's faceted browsing on things like who took the photograph along with a tag cluster.

CNET Reviews. Well, we've got a list of items. They provide details of using a template. Sorting of the items by various properties, faceted browsing is over here, and tech search at the top.

Housingmaps.com, a way to find rental properties. Here, you've got two views, one is a tabular view and another is a map view of the information, but both showing aggregates. In the tabular view, you've got templates. If you click on one of these things, you'd get details on demand. Again, they allow you to sort, in this case, by clicking on the headers of the tags. And up here, you've got your faceted browsing.

Here's the Museum of Fine Arts in Boston. Again, templated items, facet browsing on things like artist, sorting on various properties, and tech search box.

Here's one more LinkedIn. Again, templated items over here, faceted browsing over here, sorting by various properties, and the tech search box. So can't show you all the places on the web, but hopefully, I've convinced you that this is a pretty common paradigm.

To summarize, what we provide is lists, maps, grids to provide an overview of all the information; facets for filtering all that information; and then templates for showing item details uniformly.

And again, this fits very well with the Schneiderman mantra of overview first, zoom in filter, and then details on demand.

You might wonder, is the sameness of all these sites a good thing? Well, most information presenters are not ambitious, they're just trying to get the information out there in a sort of normal way. It is certainly true that very carefully designed domain that has specific information interactions will always be better.

And if you want to invest a lot of money and effort, you can make something better than this generic interaction. But it's great to have a powerful lowest common denominator, and people's experience of it makes it even more powerful, because you can leverage their expectations. They don't have to learn to interact with a new site, they already know how to do.

6.3.8 Can End Users create Data Interfaces

So I'm going to finish up with a pair of segments describing a couple of the research projects that we've been working on in this area of end user interaction with data.

The first addresses the question of whether end users can create data interfaces themselves. So far I've showed you a lot of data interfaces that were created by professionals, but if a user has a new type of data, maybe the interface that they need for that data doesn't exist yet.

And what are they going to do in that case? Well, one thing they can do is hire a professional, but that's expensive. If we look at the way the data web exists now, the sites that we visit, they're generally created by professionals. You have back-end database engineers and front-end designers.

Regular users really can't do this. And so your typical user on the web today authors static web pages, static text pages, or puts data on sites that other people build.

But what if you want to present data your way, either because it's a completely new kind of data, you know you're really interested in ocarinas and want to have an entire website devoted to ocarinas, there isn't a place where you can put your ocarina collection, right?

Or if you just have data that is like what's done on other websites, but you want to present it in a different way.

So there's Flickr for photos, there's Epicurious for food. But what if I wanted to connect those two things and have food photos and associate them, and be able to organize my photos by food, and link them to recipes.

How can I do that?

So the idea that we've been pursuing is that, as I argued in the previous segment, all of these data websites are pretty much the same. And if they're all the same, maybe you don't need to program them. Our approach has been to explore extending the HTML standard, the way you describe documents on the web, so that you can also describe the standard visualization elements that I showed you in the previous segment.

And so the ideas to use HTML itself to describe the templates that you want on your web page, to describe the views that you want on your web page, and describe the facets that you want on your web page just by writing some additional HTML tags into your document.

Those tags, of course, need to refer to your data to your columns, but it's certainly possible to do that. And end users are very comfortable with working with spreadsheets. They understand this notion of having data in rows and columns. And so if you could just use the tags to talk about specific columns of a spreadsheet, that provides a way to describe the linkage between your visualization elements and the data.

And if we can do that, then creating a data visualization is just going to be HTML authoring. So a lot of people who can't program or say they can't program are still pretty comfortable editing the source code of an HTML document or copying and modifying somebody else's.

That's how most of the web was originally created. And if they can't do that, they can still use a WYSIWYG editor for HTML. And so if we've created HTML tags for these things, we can add them to a

WYSIWYG editor. And at this point, anybody can create their own visualizations. So here's an example of the kind of tags that we imagine wanting to have. So you would put some data into a spreadsheet, say a sequence of people with first name, last name, age, and a latitude-longitude column for describing where they live, what their home address is.

And then you would start to create, for example, a template. So this is just a fragment of HTML.

Details

- **Data in a spreadsheet**
  - [first-name, last-name, age, home-latlng]
- **Templates**
  -
    ,

- **Views**
  -
  -
- **Facets**
  -

And this fragment of HTML says, I am a template. And it is for items of type Person,x things that are in the spreadsheet. And whenever you want to render an item of type Person, what you should do is you should first make an HTML span.

And into that span, you should fill in the field from the last name column of the data. Then you should put in a comma and then you should fill in the data from the first name field of the data.

So this is just a demonstration, an example of how a person should look on my web page.

Similarly, I could create views, right? I could imagine just dropping a tag into my HTML document and saying, I want there to be a map here. And this map should plot my data using this stuff in the home

latitude-longitude column of the spreadsheet. I might want to say, make a list of the data, right? Present sort of a list.

And of course, the list is going to use this template to show each of the items in the list. And maybe I'll specify that you should use the age as a sort feature in that list.

And finally, for facets, I might say, I want to make a facet that allows people to filter. And the expression that I want them to be able to filter on is the age of the people in my data set. So again, that's referring to a particular column of the data. So hopefully, this makes sense as an abstract concept of creating tagged for the visualization elements.

So we created a prototype that actually attempts to do this. It simulates this extension to the HTML vocabulary. It's a JavaScript library. And when you include this JavaScripted library in your web page, it looks for these tags. And whenever it sees these tags, it does the computation and rendering that is needed to create the views and templates, and facets that you specified in your HTML document.

It's independent of any server. It's really creating this illusion that there's just more HTML tags that you can use. And all that happens is at the final moment, when the page is on the client browser, this JavaScript runs and enriches the page with these visualizations. And so as a result, you can completely interleave it with any other HTML.

You style the page however you like, you put the facets wherever you want, you put the views wherever you want. And it is completely under your control, like any other kind of HTML document. You're not limited to the ability to embed a particular data interaction that's coming from some third-party site.

So you author one HTML document and you get an interactive data visualization as a result. So this is a particular example that was generated using our exhibit framework.

The author this is one of our own examples that we created, but we created this page. And into the page, we linked to a spreadsheet that contains information about US presidents.

And then we dropped in some tags saying make a timeline that shows when the presidents were office. And color code that timeline according to the party that the president was in. Then make a map of where the presidents were born. And inside the map, we specify these things by specifying attributes on the tags that we're writing. Make a photograph of each president using the photographs that are linked from another column of the data set. Over here on the left, we created facets.

Here's a facet that allows you to filter on the religion, which is another column in the presidents' data. Here's a facet that lets you filter on the political party. Here's a facet that lets you filter on whether or not they died in office.

So this is just one HTML document, but by virtue of our JavaScript library, which understands these new tags, it becomes an entire interactive data visualization using the same elements as people are used to on any website.

So nobody who's ever visited this page has ever had any trouble interacting with the information. In particular, they know that if they click on one of these icons, that get details. And of course, these details are generated by a template, which is another piece of the HTML that we've drawn into the page.

So exhibit is this JavaScript library, it's open source. You can go here and play with it, if you want to. It currently handles visualizations of up to about 100,000 items. It was deployed in 2007. And since then, we're inching up on 1,000 different domains that are using the exhibit framework to display a variety of visualization that have in aggregate seen millions of views. Just to take you on a quick tour of some of them. These are all the visualizations that our group didn't create, but that were just created by random people who found the exhibit framework. Here's one of our artists and albums on a timeline with filtering on artist and on year.

Here's a car finder, where you can filter on manufacturer or rating, or price range.

Here we've got the art exhibit, primitive art associated with rites of spring. I mentioned ocarinas before. Here's an ocarina catalog. You want to buy an ocarina? You go to ocarinacatalog.com. And the entire site is just this one page, which shows the details using a template.

Here are the facets to filter on things like color or tuning, or who the manufacturer is. Here it's being used for biology. This is a gene map. And you can filter on all sorts of strange biological concepts that I don't know. So this highlights the importance of letting people create their own visualizations. I can't make this visualization, I don't know biology. A biologist couldn't previously make this visualization, because they didn't know computer science. But if we make the creation of visualization just an authoring task, then the biologists can do this themselves.

Here's a PhD thesis on language acquisition, where these are all the different interviews, the different subjects that were studied as part of the process. And you can filter on, well, I can't tell you what the values are, because I don't read Japanese, but you can filter on questions and genders of interviewees, and so forth.

Here's an art history timeline using a rather dramatic color scheme.

Here is a way to hunt for rental apartments, where you can scatterplot them. The scatterplot is just another kind of view. Here you're scatterplotting distance versus price, so you can filter on whether there's a double room available. It can be used to visualize the data coming out of US government. So this is on the data.gov website, where the US government is publishing lots of data sets for use.

And here is a map of clean air status. And you can filter on things like terrain in order to understand the effect of terrain on pollution. Here's, if you want to buy a ticket at the Prairie Home Companion. They don't even use any facets, but it's just convenient to have a simple templating engine and the ability to switch between a view that's a list of shows and a view that's a map of shows.

So here's a list of court cases being seen by the European Court for Human Rights that the Russian Justice Initiative wants to track.

Newspapers can use it. So here's a map of all the unsafe bridges in Minneapolis, color coded. Red means super dangerous, and green means OK. And basically, it looks like Minnesota is way behind on their bridge maintenance.

Libraries can use it to index periodicals and law resources.

Sports fans, here's all the World Cup games, and here's a map of where the winners are, a map of who the hosts are, a timeline of when they were played, and so forth. You can filter on years or on which tournament the game is involved with.

Really esoteric stuff. If somebody wanted to make a collection of breweries and distilleries in Ontario during the First World War. I don't know why, but they were able to do it using exhibit.

Here is the personal version of Yelp. Somebody made a vegetarian guide to Glasgow with filtering on the same sort of things as Yelp filters on, like type of food and details about each one, and a map of where there.

All right, so in summary, I believe that the sameness of data websites is really an opportunity for simplification. And instead of requiring programmers to create these websites, we can now standardize around a few basic elements, the aggregate views, templates for details, and facets are filtering.

And by turning these things into simple tags, we make it possible for any user to create their own visualizations just by authoring an HTML document and a spreadsheet, ultimately doing all this through a WYSIWYG interface and allowing every user to create their own data visualizations to move beyond just text.

Well, given the power of visualization and even more of interactive visualization, I believe that this will permit individuals to communicate their ideas far more effectively to others on the web.

6.3.9 Beyond the spread sheets

So in this segment I'd like to go back to the lowly spreadsheet and ask how we can make it better without losing the things that make it so popular.

Databases

So for starters, we all know the databases are the dominant paradigm for data management. Databases operate on tables. They have columns. Each column is a particular property, a particular attribute of interest. We have one row per item. So we might have a database table where the rows are people, and for each person we have a name and age and an employer. Or we might have a database where each row is a company, and for each company we have a name, a sector, and some earnings.

And databases support all sorts of natural basic operations on this data. So we might select certain rows by filtering on values in a particular column, so select all rows where the age is 43. There's also this very powerful ability to do joins, where we might actually connect between multiple tables and talk about constraints on one table and how that impacts another table.

So if I want to, say, find all the employees of companies earning more than a million dollars, this is going to require me to connect the Person table to the Company table by what the employer property of each person is, and then use that connected table to filter on earnings of companies and see what people fall out of that filter.

And for databases, SQL is by far the dominant programming language. But it is a programming language, and that means that most people can't use it.

So instead, most people turn to spreadsheets. And so here is a typical use of a spreadsheet where you've got a whole bunch of people and you're keeping track of things like their occupation and their gender and their age in one giant spreadsheet table.

So spreadsheets are really the dominant database for end users. And in fact, this is because a direct manipulation tool. They don't require programming. Instead you just do things to the data that's sitting in the rows.

In fact, there have been studies that have found that most spreadsheets have no formulas in them. So although spreadsheets were originally created as accounting tools, where the goal was to do formulas, to do sums and differences and things like that, people have really broadened out there use and they use them as a database now where what they care about is looking at the tables, sorting the rows.

And in fact, spreadsheets have moved in this direction. So newer versions of spreadsheets now give you capabilities like filtering the rows based on certain values, which didn't actually exist in the earliest renditions when they were being used for calculation.

But despite their power and their broad use, they are quite limited, right? So in general, a spreadsheet only lets you view one table at a time. You have to switch between different worksheets in order to see multiple tables. And this makes them really unable to support joins, this ability to connect data across multiple tables, which is perhaps the most powerful single operation in databases. So it's hard to represent these connections between tables. It's also very difficult to deal with many-to-many relationships.

So if you have a group of companies and you have a collection of sectors, and each company is in many sectors and each sector is occupied by many companies, it's quite difficult to represent that in a spreadsheet.

So the goal of this research project is to add power to deal with this richer kind of data and support joins while preserving the look and feel of a spreadsheet. And we do this with two interesting ideas.

One is the idea of ***nested views***.

So instead of having flat tables, we're actually going to unpack each cell into being something richer. And the other is linkage between multiple table so that you can easily navigate from the data in one to the data in another.

So I've already talked about these sort of many-to-one and many-to-many relationships, but the example I'm going to work on is a course database. So you've got a collection of courses, and each course may have many readings associated with it.

Each course may also have many sections. And the sections may have several different instructors and, conversely, each instructor may be associated with many different sections. So this is a kind of data set that would be pretty hard to represent in a spreadsheet.

If you had to do it, you would probably use multiple worksheets: one worksheet for the courses, which for each course may be list the section names and then a separate spreadsheet for sections that for each section would list the meeting times and the instructors, and then a separate spreadsheet for instructors which would list the sections.

But now imagine what happens if you want to add a new instructor to a section. You have to go to the section sheet and add the instructor to the section. But you also have to remember to go to the Instructor sheet and add the section to the instructor. It becomes quite painful.

One approach that people will use for one-to-many data is comma separated lists. If I have many instructors in a section, what I'll do is I'll have a section column and I'll have a list of instructors in that column.

Now that lets me represent that information, but it breaks things. So for example, I can no longer filter on that column. So spreadsheet filtering right now says you pick which values you want to have up here. If you're looking for a particular instructor, the spreadsheet isn't smart enough to say, if the instructor is in this list of instructors in the column, that's OK.

So putting in comma-separated lists sort of breaks the structure of the table.

Related Worksheets

We take an alternative approach using a tool called Related Worksheets. And this is the research of a student of mine named Eirik Bakke. And Related Worksheets look a lot like spreadsheets, but you can see that they nest.

So here we have this collection of courses. And as I said, each course has a reading list. But we don't present that reading list as a set of comma-separated values. Instead, the cell contains a sub-table of readings. And that sub-table actually has two columns, one column for the author, and one column for the title. Similarly, in the sections, we have a sub-table of sections. And in fact, each section has its own sub-table of meeting times as well as an instructor.

So this kind of nested presentation really addresses the problem of working with one too many relationships and connections between different parts of the data.

So how is this kind of visualization created? Well, you start just like with a spreadsheet, with a blank table. And you start creating columns in that table. So here, I'm creating a Courses worksheet.

And now I could start entering columns of data into the Courses spreadsheet in a perfectly normal way. So here's a course code, first column. And here's a course title, second column. But now things get more interesting.

So one of the things that I can do is specify types on the data. So this column, maximum enrollment, that's got to be a number. And so this spreadsheet will tell you if you're doing something wrong, by entering text where there should be a number. But that's not the interesting type. The really interesting type is that I can specify whether a particular cell contains one value or multiple values.

So are there are multiple readings for the course? Are there multiple instructors for the section? If I just tick off the Multiple box, then the spreadsheet knows that there can be multiple values in that cell. So here I've actually said that there could be multiple text values in the course code, because a course can be cross listed under several different department codes.

The next idea that we add in is the idea that some of your columns may actually be references to information in other spreadsheets, in other cells that you've got.

So here, I'm creating an Authors table. So this is the Readings spreadsheet. But what I'm saying is that this particular column called Course, it's not just a piece of text. But instead, each cell in this readings list contains an item from the Courses spreadsheet. In other words, it refers to some row in the Courses spreadsheet. And once I make that specification, I can no longer really take arbitrary text into this field. Instead, I get a drop down menu for selecting a row from the Courses spreadsheet to populate this column.

Now, once I've done that, once I've started talking about referring to something else, well, now if I know that this column of reading list is actually not text but is a reference to the Readings spreadsheet, then I can actually use that reference to get more data about each of the readings. So here, this title is actually pulled out of the Readings spreadsheet. So if I decide to change the title, if I got the title of the reading wrong and I need to change it, I just go change that title in the Reading spreadsheet.

And its presentation is updated right here, without me having to remember to do it. The other important idea about these reference types is that we know that they are connections between two spreadsheets.

And so we provide a shortcut. If you hit Control Space, you'll actually jump from the view of this item in this spreadsheet to the view of the same item in the other spreadsheet.

So here I'm sitting on a course code. I'm in the Readings table and sitting on a course code. If I hit Control Space, I'll actually jump to that course in the Courses table.

And I'll be sitting on that reading within the Courses spreadsheet. So this makes it very easy to navigate between the different connected parts of data. So the nesting lets you show the connections in place.

And we also provide this way to teleport between them. We believe that this related worksheets model provides a richer database-like interaction than normal spreadsheets, without sacrificing the standard concepts of spreadsheets, of rows, columns, and cursors, and being able to just directly move around in the data.

We did a quick user study to try and verify this. We basically took a spreadsheet representation of the course information and our related worksheet representation of the course information. And we put those in front of users and asked them to answer questions, like, who taught such and such course? Or what course is taught by such and such person? What percentage of the grade in the course is derived from the following component? What is the email address of the teacher who teaches the following course, and so forth?

And if you look at the data, what we find is that for information that is in one table, directly in a single table, there's really no difference between the two tools, because in either case, you just need to find the right row and then look at the proper column of the row.

But as soon as your questions involve connections between different tables, then unsurprisingly, our related worksheets tool allows people to find that information much quicker by looking at either what's presented directly in the nested cells of the spreadsheet or by using this

Teleportation feature to move to the connected data worksheets without losing their place. We also did some studies of correctness. And so for example, a lot more people were wrong in answering question five than in using related worksheets, not only in terms of speed, but also in terms of correctness.

The Related Worksheets tool allows people to do a better job. To conclude this segment, spreadsheets are the dominant database tool, and that's not going to change. So we should be thinking about how to make spreadsheets work better.

Do we have to sacrifice the key capabilities of databases to handle multiple tables, multiple values, and joins? Or can we do that in a spreadsheet?

We think the answer is yes, and that if you enhance the spreadsheet paradigm, with multi-value cells, with nested information from other cells, and with the ability to teleport to related worksheets, you can get back a bunch of the power of databases without sacrificing the understandability and usability of spreadsheets.

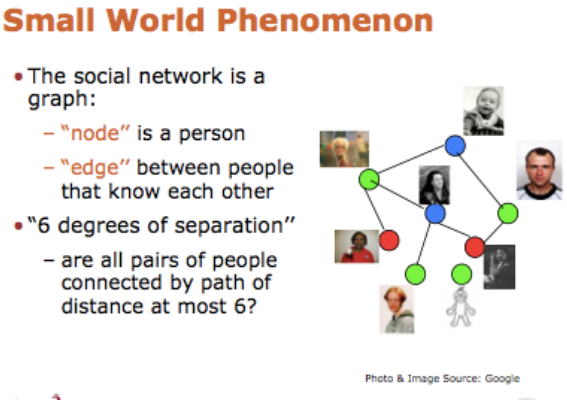
We validated with a user study that shows that people can actually use Related Worksheets faster and with fewer errors than they can using Excel.

7 Big Data Analytics

7.1 Fast Algorithms I: Ronnit Rubenfeld

There's been a huge hubbub about big data.But what I want to talk to you about is what you do when you have really big data, so big that there's no time to look at it all. What can you possibly hope to say about data that you can't even view?

Let's take an example to be concrete. Let's take the famous Small World Phenomenon. We'll model the social network as a graph. We'll associate with each person a node. And we'll place an edge between pairs of people that actually know each other.



And the six degrees of separation property, that Millman first studied, was later a subject of a Broadway play and then became even a movie, asks if all pairs of people are connected by a path of distance, at most six. So you see in this graph, this guy is not connected to anybody. So this graph does not have the six degrees of separation property.

So this illustrates the problem we have with really big data. It may be impossible to access some of the data. And even what is accessible is so enormous that no single individual or even reasonably sized group of individuals could hope to access all of this data. And even what you can access as a group is so large that by the time you access it all, the data might actually change.

So in our small world example, somebody could die, somebody could be born, things could change. So this illustrates the problem we might have with the standard notions we teach in undergraduate algorithms courses.

We teach that the goal is to get a linear time algorithm. Once you get a linear time algorithm, that is an algorithm which, on inputs encoded by n bits or words, takes at most a constant times n many time steps. We teach you that when you get this, you're done. You get a gold star, A plus. And you can move on to the next problem on the exam.

But maybe this notion is inadequate for big data. But what could you possibly hope to do if you can't even see all of the input data. You certainly can't answer the same types of questions that we'd hoped to answer before.

Any question that has a "for all" or an "exactly" type statement in it is going to be hard for us to solve. We're not going to be able the answer exactly how many individuals on earth are left handed. We're not going to be able to answer whether all individuals are connected by at most six degrees of separation. So we're going to have to make some compromise.

So the compromise that's traditionally taken by statisticians is to ***approximate***. Let's approximately determine how many individuals on earth are left handed. And this is well understood. All the sampling bounds and polling bounds, we use this to determine who's going to vote for which political candidate. We use this to determine if a certain type of drug is working on a certain disease.

So this is something that's classically studied and used all over. But because of the types of data they have now appeared are much more interesting and have certain kinds of structure that we're looking for, the questions that are now arising are quite a bit different than those that have been classically studied.

So in particular, for the six degrees of separation property, we might ask, is there a large group of individuals that are connected by at most six degrees of separation? You might ask, are at least 99% of the individuals on this earth connected by at most six degrees of separation?

So what types of approximation or compromises are we looking for? We're going to talk about two in this module. We're going to talk about property testing. We're going to try to distinguish data that has a certain property, such as networks that have six degrees of separation, from data that is far from having the property, networks where not even 50% of the population has the six degrees of separation property. And we'll define what that means.

We'll also talk about classical approximation problems. We're going to try to approximate the correct output of a computational problem. The computational problem might be the classical statistics problem of estimating how many people are left handed on this earth or it might be something more interesting, like approximating some solution to a combinatorial optimization problem.

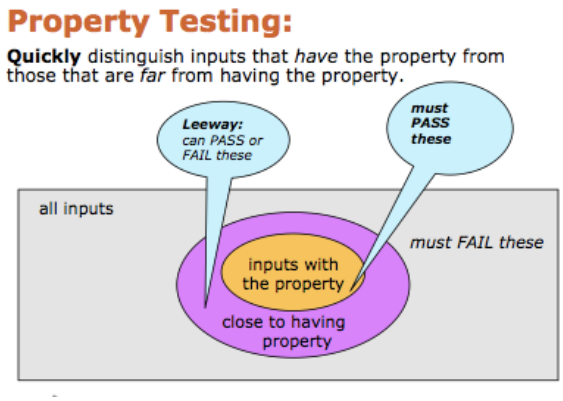
So we've seen that we're going to have to make some sort of compromise. We've talked about two types of compromises that we might make.

And in the following segments, we're going to see how and when we might apply these types of compromises to specific questions.

7.1.1 Property Testing Algorithms

So in the past segment, we saw that we're going to have to make some compromise. And in this segment, what we're going to look at is the ***property testing*** compromise. We'd like to ***distinguish data that has a certain property from data that is far from having the property***.

So it's a little bit different than the way we usually ask questions. Normally, we ask if it has a certain property or not. But here we're going to give a little leeway.



So if you let this big gray rectangle be the set of all inputs. And inside, we'll say that this yellow oval is inputs that have the property. Here, you want to say, yes, they have the property.

Our algorithm has to pass these inputs. It has to say, yes, these guys have the property. Here, we want to say no, they do not have the property. But these purple guys, yes or no is a perfectly fine answer because they don't have the property, so it's perfectly fine to say no. But on the other hand, they're pretty close to having the property, so it's not the worst thing in the world if we said yes, they do have the property.

So we have these purple inputs, where we don't care what answer the algorithm gives. And this leeway allows us to design much faster algorithms. So that's why we might want to look at property testing. You can do it so cheaply.

And a lot of times, that's the natural question to ask. For example, if you know there's noise in the problem anyway, why would you want an exact answer? And if the data is constantly changing, it's never exactly going to have the property or not have the property. It's always going to be either close or far. And it can give you a fast sanity check.

If you have to perform lots of these tests, at least the really bad inputs you can get rid of quickly and spend the rest of your processing time on the finer distinctions. And finally, it's related to the model selection problem in machine learning. So what types of properties might we look at?

Well, we could look at any type of property of any type of object. We could look at properties of functions. Is our function a linear function? Is it a convex function?

So what properties are we interested in? We could look at properties of any object and any type of properties of these objects. So we might look at whether a point set is clusterable. We might look at whether our graph has small diameter or six degrees of separation property, or we might ask, is the graph even connected?

Is a list of numbers in increasing order? Does it have small edit distance to another string? Are we close to a code word? And lots and lots more properties.

Whenever we do solve a property testing problem, our model has to specify, how do we represent the object and what are the allowable types of queries? Then we must define what we mean as a notion of close and far.

For example, we might say that two graphs are close if you can delete a small fraction of the nodes in one graph to make it look just like another graph.

Or you might say that a string is close to another string if you can delete a small fraction of the entries, and then they become identical. OK so the notion of close and far is specific to the object that we're looking at, and you have to define it each time you define one of these testers.

So what are examples of things we can test? Well, in or six degrees of separation property, we can test if a social network has six degrees of separation or is far from having six degrees of separation in constant time independent of the number of people in the network. So that's something we can do quickly.

We can also test if a large, large set of data points are clusterable in constant time, again independent of the number of points in the data set. So this weakening of what we're asking of the algorithm is really helping us to get much faster algorithms.

How do we construct a property tester? We need to find a characterization of the property that is both very efficient to test and robust in the following sense. We want the objects that have the property satisfy the characterization that we're going to be testing. And we want that this characterization doesn't allow bad objects to pass. In other words, objects that are far from having the property, they should be unlikely to pass.

And our tests are going to be randomized. So we're never going to have perfect probabilities of air. But we can make our probability of seeing the wrong answer as small as we like. Showing that objects are far from having the property are unlikely to pass is usually the bigger challenge.

So how would we go about constructing a good characterization? Let's start with a bad testing characterization for six degrees of separation. Here we have, for every node all other nodes are within distance six. So what's bad about that?

That's a perfectly good characterization of six degrees of separation. But this is a really bad characterization, because how are we going to test that for every node all other nodes are within distance six? We're going to have to run through every single node in the graph and then compare it to all other nodes in the graph. That takes forever.

Here's another bad one. It's going to fix the first problem, but not the second. We can just test it for most nodes. Now, we can just do random sampling and make sure that most nodes have the following property, that all other nodes are within distance six. So this part's not a problem. But we're running into a problem here, because how are we going to test that all other nodes are within distance six? There's too many.

But here's a good characterization. How about testing that for most nodes, say 99% of the nodes, there are many other nodes within distance six. And it turns out many is small enough that it's very doable. Many is actually something on the order of 10 or 20. It's not a big deal. And most is, we have to check that 99% of the nodes have this property.

So a few hundred samples would be good enough for us to do this. This is something we can do much faster than looking at all the nodes in the graph. So we're talking about doing a number of tests that's on the order of a few hundred, instead of having to look at a network that contains as many nodes as the people in this world.

Well, many other properties have been studied on graphs, functions, point sets, strings. And there's been a lot of exciting work in this area in recent years. There are amazing characterizations of the problems they can be tested, both in graphs and functions, that can be tested in constant time independent of the size of the input.

In the next segment, I'm going to talk about more traditional approximation problems that can be solved in sub linear time.

7.1.2 Sublinear Time Approximation Algorithms

So in this segment, I'd like to talk about sublinear time algorithms that perform classical approximations. Here, the goal is to output a number that's close to the value of the optimal solution. But since we're running in sublinear time, there's not going to be enough time to construct a solution.

So some examples of problems that have been considered are sublinear time algorithms that approximate minimum spanning tree, vertex cover, max cut, positive linear programming, edit distance. These are just classical problems have been well-studied in our community, but now we're approaching them from the sublinear time perspective.

OK, so before I get into it any further, we'd like to talk about what we mean by an approximation. So there's two notions of close that we're going to discuss. We could talk about an approximation being a ***c-multiplicative approximation*** if we output a number that's within a multiplicative factor of c of the best solution.

So we might want to output something that's at most c times the optimal solution. We might also discuss a c-additive approximation. Here, we'd put a number that's additively within c of the best solution. So here, our output should be at most the optimal solution plus c.

So here's an example, the vertex cover problem. Here, we're given a graph, and let's say let's take this graph. A vertex cover is a subset of the nodes that touches every edge. So we have to touch all four of these edges. So here's an example of a vertex cover. This is a vertex cover of size 2. If you look at this graph, you'll see that there is no such thing as a vertex cover that has only one node in it.

And the question we're going to ask is, what's the minimum size of a vertex cover? So this is one of the classical NP-complete problems. There is, however, a polynomial time multiplicative 2-approximation based on the relationship of vertex cover and maximal matching. So I'll say more about that in a minute, but what this means is even though we can't find the minimum size of a vertex cover, we'll never output a number that's more than twice as big as the minimum.

OK, so it's a pretty good approximation. What we're going to see here is that we can get a really good approximation for vertex cover on sparse graphs, and the running time of our algorithm is going to just be a constant time completely independent of the number of nodes in the graph.

So we're going to output y, which is, at most, 2 times the optimal answer. But we're going to have a little bit extra additive error. So we're going to have this extra epsilon n additive error, and that's because we're going to have to do some sampling. So how are we going to do this?

We're going to use the oracle reduction framework, and we're going to assume there's some sort of oracle that has in its head a really good vertex cover. So it's no more than twice as big as the optimal vertex cover.

OK, so this thing is going to be able to answer us queries of the following form. We're going to ask it, is node u in your vertex cover? And it's going to say, yeah, node u is in my vertex cover. Then we'll ask, is node v in your vertex cover? And it's going to say, no, it's not in my vertex cover.

So we can ask node by node questions. And we're going to use this oracle plus standard sampling to estimate the size of the cover. So what we're going to do is just pick random nodes, and ask the oracle whether these random nodes are in the vertex cover or not. And by estimating the fraction of times that the oracle says yes, we're going to have a good idea of how many nodes are in the vertex cover. But that is why we need this epsilon and additive term, because we are using sampling at this point. So that's where that comes in. And this factor of 2 comes in because the oracle doesn't actually have the best vertex cover. We can only expect that it has a factor 2-approximation to the vertex cover.

OK, now this is all great, but how do we get this oracle? I mean, that's sort of magic. And it turns out that we can actually get this magic in constant time. So there's actually two approaches to this. So one approach would be to sequentially simulate the computations of a fast distributed algorithm.

And this is an approach that's taken in constructing many sublinear time approximation schemes. But I'm not going to actually talk about that today. What I'm going to talk about is an approach that tries to figure out what the greedy maximal matching algorithm would do on the node u, because we're using the greedy maximal matching to construct a good vertex cover. And if we can figure out what the greedy maximal matching algorithm does on u, then we're done.

So to do that, I have to explain to you how we use maximal matching to get a good vertex cover. So what is maximal matching? Well, a maximal matching is a subset of the edges, such that no node is in more than one edge. So you could think of people, and you could put edges between pairs of people that like each other and are willing to get married. And then, you put edges in the matching if those people agree to get married, but you want to make sure nobody gets married more than once, OK?

It's OK to have people that didn't get married at all, but it's not OK to have people married more than once. OK, so think of that as your matching. And M is maximal if nobody else can get married. OK, if anybody else gets married, then you've broken the legality issue. OK, so a well-known fact is that nodes in any maximal matching would give you a pretty good vertex cover.

So all you need to do is find a maximal matching. And it turns out that what you do is you put all nodes matched by the maximal matching into your vertex cover, and that it's going to give you a factor 2-approximation to the best vertex cover. So the beauty of maximal matching is a very simple greedy algorithm can be used to find one.

OK, so all you need to do is, and now I'm not talking about a sublinear time algorithm, just a standard algorithm. This is the standard greedy algorithm. Just look at the edges in any arbitrary order. It doesn't matter. OK, so go through the edges one by one. And when you go through edge u, v, you look and see, have u and v previously been put in the matching? If they have not, then they can legally be

married, so add this edge u, v to your matching. If either of them was already previously married, then you can't put edge u, v, into the matching, so don't. OK, so you run through every single edge, put the ones in that you can, and output this matching.

Why is it maximal? Well, if you decided not to put the edge u, v, into the matching, then either u or v had to already have been matched. OK, so that gives you a nice maximal matching. It's a very simple algorithm.

So now what we want to do is figure out whether a specific edge e was put into the matching by the greedy algorithm. But we don't want to run through all the edges in order to figure out if edge e was put in the matching. So good news, you only have to look at edges that came before edge e in the matching to figure out if e is placed in the matching.

OK, I want to know if e is placed in the matching. I need to look at anything adjacent to it, any edge that shares a vertex with it, and see were they put in the matching? And if none of them were placed in the matching, then neither of u's endpoints were matched previously. So you're going to end up placing e in the matching.

OK, so you need to know if adjacent edges that came before e in the ordering are in the matching. You don't need to know anything about edges that came after e. OK, so this sounds like it might help. But you still could have arbitrarily long dependency chains.

So let's just take this very simple graph of the line and let's look at this edge here. How do I decide if that's in the matching? Well edge 112 came before edge 113. I don't need to check and see whether 113 was placed in the matching. I know it wasn't because it comes after e in the greedy ordering.

But I do need to figure out was 111 placed in the matching? Because if 111 was placed in the matching then 112 cannot be. OK, but to figure out 111, I have to figure out 110. To figure out 110 I have to figure 89, and so on. So that's my problem. And essentially I could have this arbitrarily long line. And I need to figure out if I'm in odd or even number of steps from the beginning.

OK, so that seems a real pain to do. But the idea that works here is just assign a random ordering to the edges. Greedy works under any ordering. The random order can be shown to have short dependency chains.

So it's really beautiful algorithmic idea. It gets simple algorithms with really fast running times. And moreover, it's quite general. So you can use it plus some more complicated algorithms to get good sub-linear approximation algorithms for other problems as well, such as sparse maximum matching, set cover, positive Linear Programming, and a number of other problems as well.

So you can get even better results for special classes of graphs, including planar graphs. These would be better in terms of approximation parameter.

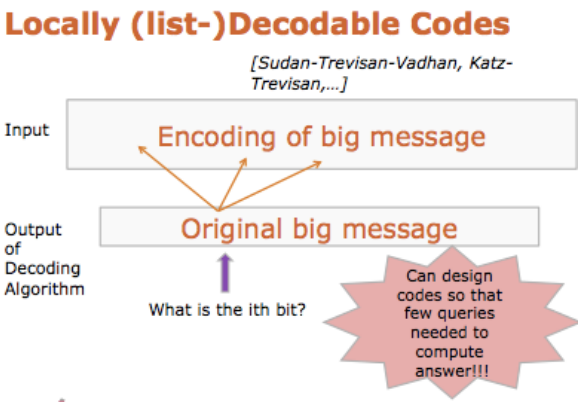
So we've seen a beautiful idea that allows us to get simple algorithms that run in constant time and give us really good approximations of many combinatorial approximation problems.

7.1.3 Local Computation Algorithms

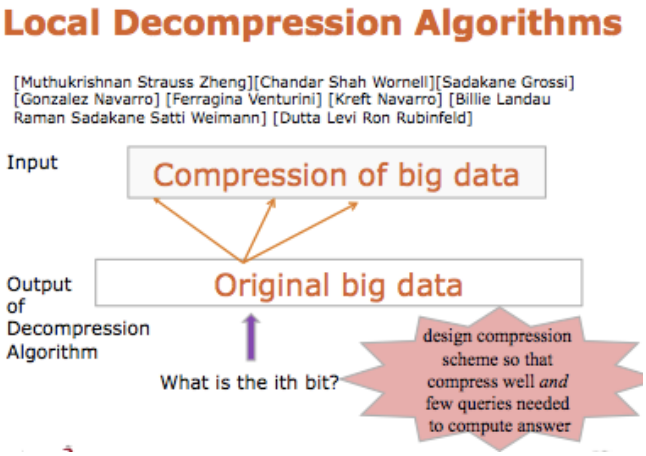
So up until now, we've been talking about the case where we have big inputs, but the answer we output was just a number or a pass-fail type indication.

What happens when the computational problem also has a very big output? Maybe we don't actually need to see the whole output. Maybe we just want to see small portions of the output. Do we need to see all the input? Do we need to solve the whole problem in order to get the ith bit of the output?

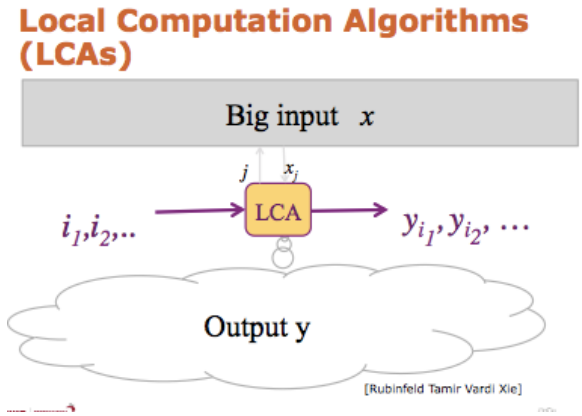
So let's take some concrete examples. Suppose my input is the encoding of a very big message. Now I'd like to decode this message, so the output of my decoding algorithm is going to be the original big message. But in fact, I don't actually need the full, original message. What I need is just the ith bit of the original message. Do I need to decode the whole message in order to figure it out?



It turns out that you can design codes, and they're called ***locally decodable codes*** so that very few queries to the input are needed in order to figure out what the ith bit is in the decoding.

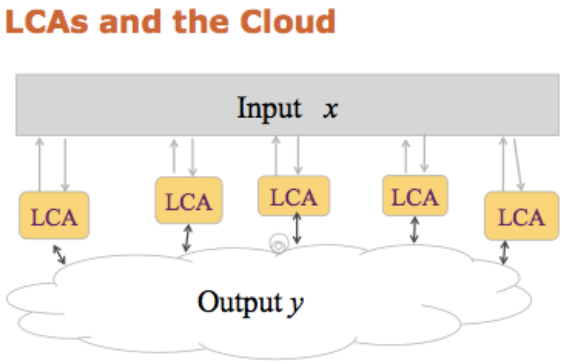


A similar idea works for ***local decompression algorithms***. In this case, the input is the compression of a very large piece of data. The output of the decompression algorithm would be the original big data, but I don't want the whole original data. I just want to know what's the ith bit or word of this original data. And it turns out that you can design compression schemes so that they both compress well and few queries are needed to compute the answer to such questions as what's the ith bit.



So this led us to propose a new model ***of local computation algorithms***. Here, our input is huge but it's written down. And we'd like to compute the output y, but we don't actually need all of y. What we need is to be able to answer queries such as, what is the I first bit of y? What is the i second bit of y? So we're going to assume there's some local computation algorithm that is allowed to access the big input x, and it's just supposed to tell us, whenever we ask it a question, what is the value of y at bit i sub 1? So it tells us y i sub 1 when we ask it about i sub 1. It tells us y i sub 2 when we ask it about i sub 2. So the models of locally decodable codes and local deeper compression algorithms can be put into this model.

But what more can be done in this model? Well, it's a relatively new model, so there's a lot of research left to be done. But we have found a few non-trivial examples, *such as maximal independent set certain constraint satisfaction problems, various graph sparsification type questions, and local algorithms for ranking web pages, graph partitioning, and even some type of data reconstruction questions*.



One hope for these LCAs is that you might be able to use them in the cloud. So there's some input x written down, and there's these independent local computation algorithms running around but not talking to each other. And you could imagine clients at different locations asking their copy of the local computation algorithm, what is the value of y at certain locations.

And these local computation algorithms can run around giving consistent answers without actually having to solve the whole computational problem. So the LCA model has just been proposed in the last couple of years.

There are quite a few recent results in this area. And we think this is an exciting new direction for research.

7.1.4 Big Distributions

Up until now, we've been talking about the case where the data is written down.

But I'd like to turn to a different model, and this is the case where the only way you can access your data is via random samples. How many samples of the data do you need in order to understand your data?

Distributions arise in every scientific and economic endeavor, but let's just take an example of the lottery. If you want to play the lottery, wouldn't you want to know that the lottery is fair?

Image Source: Google

For example, if you look at the websites, it would seem that maybe some of the lotteries are not so fair. Some websites claim that you can predict future winners. In fact, there's a true story. There is the Polish lottery, Multilotek, which is actually a machine that had a bunch of balls in it, about 80 balls. And what the lottery did is choose uniformly at random a distinct 20 balls out of the machine.

So these are 20 numbers chosen at random from 1 through 80. But the problem was the initial machine was biased. So the probability of the numbers 50 to 59 were too small, and when people figured that out, they made a lot of money. They just didn't bet on 50 to 59. So it may be possible that your lottery is not fair. Now in this lottery, you could see it, because there were only 80 balls. Every time they took draws, they took 20 out of the 80 balls. You could see how many times they showed up. So basically, every number showed up multiple times, and you could get really good estimate for every ball how often it got pulled out of the machine.

But let's take another example. The New Jersey Pick 3 or Pick 4 Lottery-- what do you do there? In the New Jersey Pick 3 Lottery, you pick three digits in order, 0 to 9, 0 to 9, 0 to 9-- so that's 1,000 possible values. And in the Pick 4 Lottery, there's 10,000 possible values, because you're picking four digits in order. So let's even assume that every time you pick a new number in the New Jersey Pick 3 or Pick 4 Lottery, it's actually chosen from the same distribution independently of previous days.

But I just want to know, is this distribution uniform?

Well, let's try it out. So we did this test a while ago. In the Pick 3 Lottery, there were 8,522 results from when it started in 1975 to when we did the test in 2000. And remember, there was 1,000 possibilities, so we had roughly 8 and 1/2 times as many trials as we did possibilities. In this case, the chi-square test only gave us 42% confidence that it was the uniform distribution. For the Pick 4 Lottery, it started only in 1977, so we had fewer results. We only had 6,500 results. And there are 10,000 possibilities, so we had fewer results than possibilities, and the chi-square test gave us no confidence.

So here we have distributions on very big domains, and given samples of the distribution, we need to know various things about it. We might want to know the entropy, the number of distinct elements. We might want to know something about the shape of the distribution, is it monotone? Does it have, like, a single hump? Is it bimodal? Is it close to uniform? Is it close to a specific Gaussian distribution? Does it have the Zipfian characteristics?

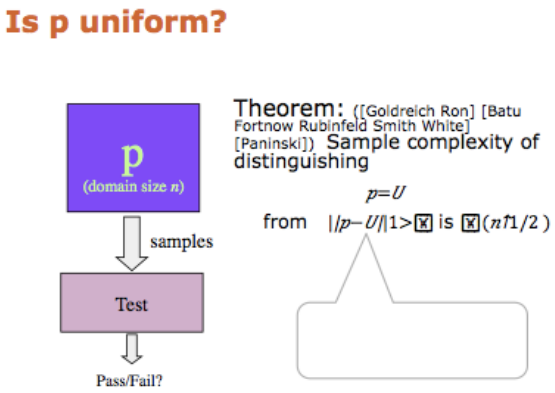
And we can't assume anything about the shape of the distribution. We can't assume that it's smooth or that it's monotone or that it's a nice, normal distribution.

These are questions that are considered in statistics, information theory, machine learning, database algorithms, physics, biology, pretty much every scientific endeavor, and the key question is, how many samples do we need in terms of the domain size? In particular, do we actually need to go through the domain and estimate the probability of each domain item? Is it possible to have a sample complexity

that's sublinear in the size of the domain? In particular, if we can get sublinear sample complexity, that would rule out the standard statistical techniques and any method of learning the distribution's probabilities on every domain element. So our aim in the upcoming segments is to see if we can get algorithms with sublinear sample complexity.

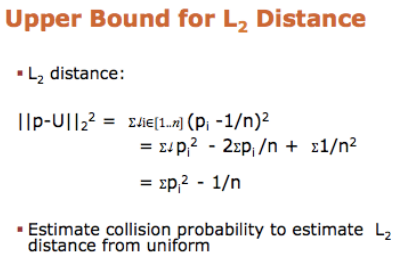
So our aim now is to see if we can find algorithms for understanding distributions, using a sample complexity that's sublinear in the size of the domain. So we're going to look at the problem of detecting similarities of distributions. And there's lots of types of questions you could ask here. You might ask if two distributions, p and q, are close or far. We're going to assume that p is given to us only via samples. The probabilities of p on every domain element are never written down anywhere.

The only way we can understand anything about p is to push a button and get another sample. But q, we can assume that q is either written down, known to the tester. Maybe we can just assume that q is a very simple distribution, such as the uniform distribution. So we don't even have to write it down. Or we might assume that q is also given via samples, just as p is. So let's start with the simplest problem of testing whether the distribution p is uniform.



So let's start with defining the notion of distance, the commonly used L1 distance. What it does is it sums over all domain elements, i equals 1 through n, the following quantity , the absolute value of p sub i minus what the uniform distribution assigns to i, which is just 1 in n. So we take those differences, sum up their absolute values over the whole domain, and that gives us our L1 distance. So what we're going to try to do is distinguish the case where p is the uniform distribution from the case where p is far from uniform in L1 distance.

And it turns out this problem can be solved in sublinear time. In fact, square root the size of the domain. So we know that you need that many samples. You need square root and many samples. And you can do it in some constant times square root n many samples.



So we're going to go a circuitous route. And first, I'm going to talk about the L2 distance. So in fact, this is the L2 distance squared. What it does is it sums over all i in the domain, something very similar to be the L1 distance. Instead of looking at the absolute value of p sub i minus 1/n, it's going to look at p sub i minus 1/n quantity squared. And it's going to sum this up over the whole domain.

Now, some very simple algebra. You just take the squared term and write it out. And you get the standard p sub i squared minus 2/n plus 1/n squared. And then you break this apart into sums, and you get three terms. You get this one term, summation p sub i squared, that we're going to talk about in a minute. So what's this term here? We have the summation p sub i over all elements of the domain. So what's that going to be? P is a probability distribution. So the summation of p sub i over all elements of the domain is just 1. So this term is minus 2/n.

And what's this term here? Well, we're summing 1/n squared, but we're summing it for i equals 1 through n. So we're summing it n times. So this is equal to 1/n. So these two terms sum up to minus 1/n. And together, they give us this term right here. OK. So now we have two terms. We have this minus 1/n, and I'm assuming we know n. And then we have this summation p sub i squared term. What's that?

That is a standard quantity, called the ***collision probability***. It's exactly the probability that you pick two elements from a distribution and you get the same domain element both times. So we know what that is. And the idea is going to be we know 1/n. We're going to estimate summation p i squared.

And that's going to give us an estimate on the L2 distance squared of p from the uniform distribution.

And that's how the algorithms are based. This is going to give us an algorithm for distinguishing the case where p is the uniform distribution from the case where p is epsilon far from uniform distribution, as follows. We're going to estimate the collision probability. That's going to allow us to estimate the L2 distance of p from uniform. And then we're going to use known and standard relationships between the L1 and the L2 distances to make conclusions about the L1 distance. There's still some issues remaining.

The collision probability of the uniform distribution is 1/n, so we're going to need a lot of samples in order to estimate it. But it turns out that actually root n samples are enough because you can take order root n many samples and look at all pairs and see if they collide.

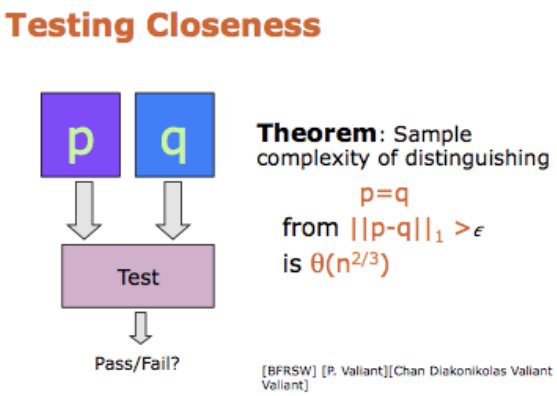
So that gives us something like some constant times n many pairs, gives us lots of pairs to look at. The problem is they're not all independent anymore. So we have to be careful in our analysis. But it works out, and it gives us a root n sample running time.

I should mention there are other ways to skin this cat. Paninski used a completely different estimator. And it's fairly straightforward to see that you need root n samples in order to solve this problem. It's essentially a birthday paradox-like argument. You can't tell anything until you see some collision. And you can construct distributions, which are very far from uniform, but you don't see collisions until root n samples.

So let's go back to the lottery question. It turns out that when we did this test in 2000, we had plenty of samples to determine that the New Jersey Pick-3 and Pick-4 lottery really are drawn from the uniform distribution.

We've seen that you can test whether a distribution p is uniform or not. And it's nearly the same complexity to test whether p is any other known distribution.

So if you have any specific distribution, let's assume it's written down somewhere, you can test whether p is that distribution or far from it in pretty much the same way we did uniformity testing with a little bit of added complications.



OK, so let's go to a somewhat different problem. Here, both p and q are given via samples. OK, so q is never written down anywhere, and we don't actually know what it is. You might use this if you're trying to see if there's a trend change in your sample data.

And here the way we're going to model it is you have two probability distributions, p and q. They are two different boxes. Each time you need, you can press a button and get a sample from either one of them.

And what we'd like to do is distinguish the case that p and q are identical from this case that p and q are far in L1 norm. And it turns out that this problem is quite a bit harder. We initially thought it should be root n samples, but it turns out that the complexity of this problem is actually n to the 2/3. So it's somewhat more.

Why it would be so different? Well it's fairly straightforward to show that collision statistics are all that matter. And one p and q could be arbitrary and, in particular, not so close to uniform. So if you have a pair of distributions where they assign identical probabilities to their heavy elements, but there's lots and lots of collisions there, and they have very different probabilities assigned to the small elements, then you won't see it, because the collisions of the heavy elements are going to cloud everything up.

All right, so we've talked about separating distributions that are identical from those that are epsilon-far in L1 norm. What if you want to get an additive estimate on the distance? Or what if you want to get something a little bit weaker?

I want to separate distributions that are epsilon close from those that are two epsilon far. Is this something I can do? It turns out this is a much harder problem. You can still do it in sub-linear time, but just barely. So here you can do it in an n over log n samples, and you need that many samples. It's a somewhat harder problem, but this is a very surprising result. It just came out in the last couple of years. And it's really taken the world by storm.

How do the algorithms work? The algorithms use collisions to determine wrong type behavior. We already saw that in uniformity testing. But what they do is they use linear programming to determine if there's a distribution with the right collision probabilities and the right property. For lower bounds, we know that the collision statistics are the only relevant information that an algorithm could use. So because we're analyzing collisions, and collisions are not independent, we need new analytical tools.

In the past couple of years, a new central limit theorem for generalized multinomial distributions has been developed. This powerful tool has already been applied in many new results.

In the next segment, I'm going to talk about a number of other properties for which sub-linear property testers have been devised.

So let's turn to other properties of distributions that you might want to understand in sub-linear samples. And two important quantities that you'd want to understand about your distribution are the ***entropy*** and the ***support size***.

The ***support size*** is the number of elements in the domain that have positive probability in your distribution.

The entropy is an important measure of information that's widely used. One setting is to determine whether neurons respond to various stimuli. Each application of a stimuli gives a sample of a signal and by looking at the entropy of this signal, you can determine whether the neuron actually responds to it.

Another place where we are interested in the entropy of information is to determine whether the data is compressible or not. Is the entropy something we can estimate? In general, we can't even get multiplicative approximations of the entropy, but kind of for stupid reasons.

And that's because it's very hard to distinguish a distribution which has all its probability on one element, from a distribution that has all its probability on one element, except a tiny, tiny, tiny, amount on a second element, because you're never going to see that in that second element. So because of that, you can't get a multiplicative approximation of the zero entropy distribution.

But what if you knew the entropy was at least some constant? The entropy can be up to logarithmic in the domain size. So assuming it's a constant is just really assuming that there's some probability that it's on a second element. In this case, you can gamma multiplicatively approximate the entropy with just about n to the 1 over gamma squared samples. What that means is if you want to get a factor of two multiplicative approximation of the entropy, you only need n to the one quarter many samples.

And in fact this is tight. We know that you actually need at least that many samples. You can't do better than that. We know that this problem requires n to the 1 over gamma squared samples in general, although there are some special case distributions where you can do better, such as when the support size is small.

There are similar bounds for estimating the support size. What about additive approximations for entropy and support size? Turns out, here again we can get just barely sub-linear, we need n over log n samples, but we can do it with n over log n samples.

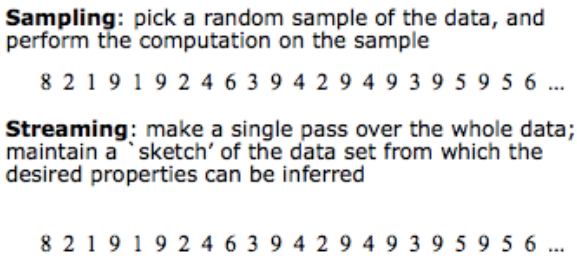
Lots of other properties have been considered. Some examples are whether a joint distribution is independent or not, whether a distribution can be succinctly described as a K-histogram distribution, whether a distribution is monotone, increasing, whether it looks like it's bimodal.

Properties of multiple distributions, are they all the same or not? It's an active area of research. I hope I've convinced you that for many problems, we need a lot less time and samples than one might think.

Many cool ideas and techniques have been developed and it's an area of exciting research, so there's lots more to do. In the upcoming modules, you'll see some of these ideas develop further.

7.2 Fast Algorithms II: Piotr Indyk

In this module you will get an overview of basic algorithmic techniques used to process massive amounts of data. In particular, we'll focus two algorithmic techniques called ***streaming*** and ***sampling***.



So what are these techniques about?

Let's start from ***sampling***. The basic idea of sampling is a very natural one. The idea is that instead of processing the whole data set, we instead pick up a random sample of the data, and then perform the computation only on this sample that we picked up. So for example, if this is our data set, which is a large collection of, in this case, digits, the sample algorithm will start by first picking up a small sample of the data set. So for example, the algorithm could pick up this element, this element, this element, this element, this element, and this element. So that will be the first step of the algorithm.

And in the remainder of the computation, the algorithm will perform the processing only on the sample. So this clearly reduces the amount of resources needed to perform the computation. And as long as the sample is picked in a representative way, the answers for the sample are still pretty accurate compared to the true answers. So that's sampling.

On the other hand, streaming makes a different approach to massive data processing. The streaming algorithm touches all the data elements. So the streaming algorithm would first start by looking at the first element, then second element, third element, and so on. It would make one pass over the whole data set. And as a result of this pass, it would store some sketch, or the synopsis, of the data.

So instead of storing the whole data set, it would only store some partial information about the set data. And after the whole data set is read, the algorithm would simply infer the desired properties of the data from the synopsis alone. So this is a streaming approach to a massive data process.

So what are the pros and cons of these two approaches?

Let's start from sampling again. What are the main positives of using sampling? Well, the main positive of using sampling is that we perform the computation only on the sample, which means that we are setting out to reduce both the storage needed to store the data set as well as the computation time.

The second advantage is that we only need an access to data elements which are actually included in the sample, which means that all the data elements which are not in the sample do not even need to be materialized or generated.

So in situations where it is costly to even generate the data, the cost can be further reduced since we don't have to generate the data which we are not using. And we'll see a few examples of this benefit later in this module.

On the other hand, what are the advantages of streaming? Well, compared to sampling, the main advantage, or one of the advantages, is that in streaming algorithms, every data element is seen at least once. So one can say that no element is left behind. All the elements are seen at least once. So one can see advantage of this approach.

For example, if the goal of the algorithm is to simply detect whether the data stream contains a particular element, for example, one. Streaming algorithm for this problem is very easy. The algorithm just makes a pass over the data and checks whether any data element is equal to one.

On the other hand, sampling can potentially miss the data element if the stream contains very few ones. So in that sense, streaming can be more accurate than sampling because all the data elements are accessed.

Another advantage of streaming is that despite the fact that that makes a pass over the whole data set, it still uses a little storage because it doesn't store the whole data set. It only stores some synopsis. And the computation time is at least linear in the data size because the algorithm makes a pass over the full data.

However, most of the algorithms we see in this module in fact have very efficient running times. In the remainder of this module, I will present an overview and examples of both of these approaches.

We'll start from streaming. In particular, we'll see a very efficient algorithm for estimating the number of distinct elements in the stream using very limited space. So for example, if the data set consists of a

bunch of web clicks, the number of distinct elements estimates the number of different users that actually clicked on this website, as opposed to the total number of clicks, which is a very useful statistic in many situations.

And I will say this algorithm's actually very efficient. In particular, one of the algorithms, in order to estimate the size of vocabulary of Shakespeare, who as an author was a rather large vocabulary, and very prolific. The algorithm is only 128 bytes to estimate the number of distinct words in all works of Shakespeare with pretty good accuracy. So you can see that using a very limited synopsis, we can perform a computation of a rather large data set.

And we will also see examples of other problems which are solvable using a streaming approach. And I will give an overview of what kind of questions those algorithms answer.

In the second part of this lecture, we'll see an example of a sampling approach. And in particular, we're going to see an overview of recent algorithms for performing a discrete Fourier Transform, a well-known task in a single processing, in time which is much faster than the well-known fast Fourier Transform.

This algorithm applies to data which is sparse, namely, to signals which have a very small number of large coefficients in the spectrum. For such signals, our algorithms will run very efficiently because they will access only few samples of the data. And as we see in many situations, the running time of the algorithm will be actually more efficient than the running time of the well-known fast Fourier Transform algorithm.

7.2.1 Streaming algorithms

In this segment, we'll get into a little bit more details of streaming algorithms. So let's recap what streaming algorithms are about.

As I mentioned in the previous segment, streaming algorithms perform only a single pass over the data. So it sees first the item number one, number two, number three, and so on. It makes a pass over the whole stream of data. Typically in streaming, when we design streaming algorithms, we assume that the number of elements in the data stream is known, at least approximately.

Oftentimes, it's perfectly fine to just overestimate it slightly, because the performance of the streaming algorithm does not change that much if we overestimate the size of the date stream. OK, so that's what the streaming algorithm does. It makes only one pass over the data.

Now, as I mentioned earlier, the main feature of the streaming algorithms is that they do not store the whole data set, but they only store some short synopsis of the data. So as a result, the algorithm uses only very limited storage. Typically, the amount of storage used by algorithm is only logarithmic, or logarithmic to some power of the size of the data, which means that the algorithm can process really huge data sets, terabytes of the data using an amount of storage which is only in an order of tens of thousands, or maybe hundreds of thousands, of bytes.

So these are typically very efficient algorithms. Sometimes, the algorithms are less efficient. For example, the storage could be something like square root of n. But still, it offers a big improvement over storing the whole data set. And the main focus of data-streaming algorithms is to reduce the storage. However, all the algorithms that we're going to see in this module in fact also offer a very fast processing time per element. Again, it's on the order of a few operations in order to process any given element. Now, the key property of streaming algorithms is that they are both ***randomized*** and ***approximate***.

Randomized means that the algorithms themselves use pseudo-random number generators, which help guide the decisions made by the algorithm they also approximate in the sense that they do not give an exact answer, but they give an approximation with a controllable accuracy of the approximation. So in other words, the algorithms report an estimates of a desired quantity with the property that are the probability that the estimate is equal to truth, up to some small approximation error, the probability of this event is at least one minus probability of failure. And both our epsilon, the accuracy parameter, and p, the probability of failure, are our parameters that we control, and we can set them to any desired level to ensure that the algorithm reports are answers that we are looking for. This is often, although not always, necessary in order to obtain very efficient algorithms for streaming problems.

So let's see a concrete example of a problem which is solvable using streaming techniques.

And this will be the problem of counting the number of distinct elements in the data stream. So in this problem, the stream consists of a bunch of integers from some given range. For example, it could consist of a long list of digits. So in this case, the range of those numbers would be between one and nine. In the example of counting the number of distinct words in Shakespeare works, the range of the numbers would correspond to the total number of possible words in the universe.

So the range would be very large. But nevertheless, we'd like to estimate the number of distinct words using very limited storage. So the goal of the program is to estimate the total number of distinct elements in the stream up to some precision with some probability.

Counting Distinct Elements

- Stream elements: integer numbers from 1...m
- Goal: estimate the number of distinct elements DE in the stream
  - Up to 1±ε
  - With probability 1-P
- Simpler goal: for a given T>0, provide an algorithm which, with probability 1-P:
  - Answers `DE>T' if DE> (1+ε)T
  - Answers `DE<T' if DE< (1-ε)T

However, for the sake of simplicity, I will not give an algorithm for this particular program, but I will present an algorithm for a somewhat simpler problem, where the goal is to just simply figure out whether the number of distinct elements is smaller than or greater than some specific situation.

So what we'll actually solve is we'll provide an algorithm, which, if the number of distinct elements is somewhat greater than a specific threshold T, it says that the number of elements is greater than T. And if the number of distinct elements is smaller than the predefined threshold, it will say that it is smaller than threshold. So this is the problem that we'll actually solve. And in fact, similar techniques can be used to solve the more general problem.

So this concludes the segment. And in the next segment, we are going to see how to solve this simpler problem using a very efficient amount of space.

So let's get into the details of how do we solve the ***distinct element problem*** in a massive streams. So again, let's recap the problem. We want to count the number of distinct elements in a stream. We also have the elements in the stream integers from some range 1 to m. Our goal is to estimate the number of distinct elements with some controllable error and that was a controllable probability of failure. But instead of solving this specific program, we'll focus on a simpler problem, where our goal is to estimate whether the number of distinct elements is greater than or smaller than some threshold. So in particular if the number of distinct elements is same greater than the given threshold, then we say it's greater than. If the number of distinct elements is smaller than the specific threshold, we'd like to say, it's smaller than.

So that's how the problem that we're going to solve very efficiently in this asset. So let's see how to solve this simpler problem for estimating the number of distinct elements.

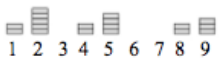
So how do we solve this problem? Well, in the first step, we'll introduce a somewhat different perspective on the stream of data. So thinking about this stream as a sequence of data elements, we'll think about it as a sequence of updates to the count vector that stores the counts of all distinct elements in the data set. So this is how this count vector could look like after we make a pass over the whole data set.

So, for example, the count of the element number 1 is equal to 2, because 1 appears twice in the data stream. That appears here and here, and nowhere else. Similarly, the element 2 has count 5, because it appears five times in the data stream.

So how do we update the counter? Well, first, at the beginning where we don't see any elements, the count is equal to 0, because there are no elements at all. And then, each time we see a new element from the data stream, we interpret as simply incrementing the counter corresponding to the data element. So if we see 1, we increment the count of 1 by 1. And you can see that in this presentation of the problem, our goal is to simply estimate the number of non-zero entries in the count vector c.

- Stream: 8 2 1 9 1 9 2 4 4 9 4 2 5 4 2 5 8 5 2 5

Vector c:



And we'll denote this number by number of NZ of c, which stands for our number of non-zero entries in the count vector c. So as you can see, the total number of distinct elements in the stream corresponds to the total number of non-zero entries in this vector here, because each distinct element in the stream will correspond to a non-zero entry in this vector. So from now on, our goal is to estimate the total number of non-zeros in this dynamically allocated vector using a small space. Now, of course, we could estimate the same number just buy storing the whole count vector.

But this will require storing the counter for each different data element. And in this particular case, as we have only digits the total size of the vector is small. However, if we want to estimate the size of the vocabulary of the works of Shakespeare, than the total size of the universe is very large, and we couldn't afford to store all these counters. So we said, now we have to compare those counters somehow, but still get an approximation to the true answer.

Well, the streaming algorithm proceeds in three stages,

Preprocessing stage, where it prepares certain data structures

Streaming phase, where it makes one pass over the data and updates those data structures

Summation phase, where, based on this synopsis, it infers the number of non-zero elements in the stream.

Pre-Processing Stage

So let's start from the preprocessing stage. In the preprocessing stage, the algorithm simply selects a bunch of random subsets of coordinates, such that each coordinate has a probability of belonging to a set equal to roughly 1 over the threshold. So each coordinate in each set is included with probability roughly 1/T. The important fact is that we actually do not need to store those sets explicitly, because storing them could actually take quite a lot of space itself.

Instead, we can simply implement a test, whether an element belongs to a set or not, by using pseudo-random generators. I will skip the details of this process, but this can be done. And in fact, you don't need to store the subsets at all. But still we can use them for our algorithm. So that's what the preprocessing does.

It generates a bunch of random sets of the coordinates.

Stream Processing

Now what does the algorithm do during the stream processing phase? Well, the algorithm is extremely simple. Essentially, for each of the sets, it computes the total sum of the number of elements in the stream that belong to that set. So the algorithm simply makes a pass over the whole datastream. And whenever it sees a new element, it checks whether it belongs to the set or not. And if it does, it increments the counter. And you shall read the counter, which we are going to denote by sums equal to 0. And for each new stream element, if it belongs to the set, the counter is incremented. So this is an extremely simple algorithm.

It's very efficient in terms of time and very efficient in terms of space, because it uses only k different sums to represent the input. This is what we store after performing the pass over the data. Well, how do we use those counters in order to estimate the number of non-zero elements? Well, the procedure is, again, very simple. After we get those sums, which sum up the total number of distinct elements that actually belong to the given set, we go through all of the sums and see which of them are equal to zero and which of them are greater than zero.

We denote this number of sums by Z. And the algorithm makes the final decision by comparing this value Z to a specified threshold, which is equal to the number of sets divided by the number e. And if the number of non-zero elements is greater than the threshold, so there are many zeros in our sum counters, then it says that the number of non-zeros is actually small.

In the opposite case, where the number of zeros in the counters is large, then it's because the number of non-zeros is actually large. So this is a cool algorithm. It is very space-efficient. It uses only the counters. It's also very time-efficient, because it computes only those sums. So the only thing that remains is to understand why this algorithm actually works.

So why does it report a great answer with a simple probability? In this lecture, I will not give a detailed analysis, which can be found in the references. However, I will just give you a basic intuition behind why this algorithm works. And the intuition is as follows. So suppose that the number of non-zeros is actually small. So there are very few distinct elements in the stream.

There are many repetitions in the stream. So if this is the case, if the number of non-zeros is very small, then there is a very good chance that none of those non-zero coordinates will be selected in the sets. Because the sets are chosen in such a way that an element belongs to the set with only very small probability. So if this is the case, if those sets, Sj, do not contain any non-zero elements, then the corresponding sums will be small, equal to zero. And therefore the number of zero sums, therefore the number Z, is going to be large. And Z will exceed this threshold. So we are going to be in this case.

In this case the algorithm would say the number of non-zeros is small. In the opposite case, where the number of non-zeros is large, then there is a good chance that the sets will actually contain some of those non-zero entries. In that case, many of the sums would be non-zero. Therefore, the value of Z is

going to be small. And this means that this condition is not satisfied, so we are going to report that the number of non-zeros is large. So that's the basic intuition behind the algorithm. And for details you can look up the references. OK, so this concludes the description of the algorithm. You can see it's a very efficient algorithm. It uses very little space. And it can be shown that the algorithm reports answers with a very small error, with a very high probability.

All right, so what we have just seen was a very efficient algorithm for estimating the number of distinct elements in data strip. So in fact, it was a variant, or a simplification, of an algorithm developed by Flajolet-Martin quite a long time ago in 1983.

However, this problem has been a subject to lots of research, and even more efficient algorithms for this programs are known. So the best result now to date for solving this problem is due to Kane, Nelson, and Woodruff who showed that the number of elements can be estimated using a space which are for relatively reasonable value of epsilon.

The space is only logarithmic in the total length of the data stream. So you can see that the logarithmic dependence means that the algorithm is extremely efficient, because you can process even terabytes. And the amount of bits needed to store is only something like only 30 or 40. However, this is not necessarily the most efficient algorithm in practice.

The most efficient practical algorithms for this problem are due to Durand-Flajolet and the related work. These algorithms are called LogLog and HyperLogLog. And these are the algorithms for which the analyses of the work of Shakespeare was performed. So in particular the algorithm LogLog, needs only 128 bytes in order to estimate the size of the vocabulary in all works of Shakespeare with a very limited error.

So, you can see that both in theory and practice, these are extremely efficient algorithms that scale very well to even larger data sets. So, this was one example of a task that can be solved using a streaming approach, however there are many others. OK, so let me just give you a quick overview of what other basic important statistical questions can be solved using a streaming approach. So another very well studied problem is that of finding heavy hitters in the stream.

The problem has been subject to lots and lots of work. And here, what we want is to identify coordinates in count vector which are large. Which means that we want to identify data elements which occur many times in the datastream. And often the algorithms in fact even provide the approximate number of times that these elements occur in the stream. So again, this is a very useful question, right?

You want to find heavy hitters. For example, again, if you have a stream of clicks on the website, you want to figure out very quickly which websites are the most popular ones. And that's exactly what the heavy hitter let's us do.

Of course this problem can be also used using random sampling. However, the algorithms developed in the streaming folder are in fact much more storage efficient that the corresponding sampling algorithms.

Another program which can be solved very efficiently on the datastream, is there to estimate the entropy of the datastream. Entropy is a very basic notion which lets us understand how diverse is the distribution in the datastream, and this is something that very efficiently can be computed on the datastream using only logarithmic memory.

We also solve a bunch of other important statistical questions such as testing independence of our random variables. And we can also estimate basic statistical quantities of the stream, such as the median, the quantiles. We can even get the full histograms of the whole distribution of the datastream.

And all of these tasks can be done very efficiently in a streaming model. So this concludes an overview of the streaming folder.

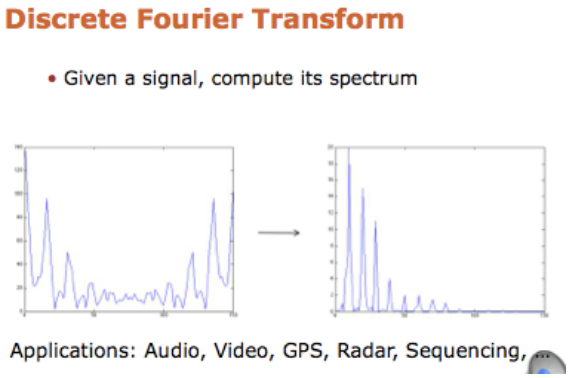
In the next segment, we'll move onto sampling and we'll see how using sampling we can solve discrete Fourier transform very efficiently.

7.2.2 Sampling Algorithms for the Sparse Fourier Transform

Now we move on to sampling algorithms for processing massive data set. So in the protocol we'll see how to do sampling in order to solve the Discrete Fourier Transform.

So what is a Discrete Fourier Transform? It's actually a very, very powerful tool for processing signals in a digital domain. What a Discrete Fourier Transform does, it takes a signal which is sampled in the time domain and converts it into a signal which represented in a frequency domain.

So this is a very simple task. But really, this task is at the heart of many modern tools for signal processing. And the reason why this task is so useful is, often, we get much more information about the signal when we look at it in the frequency domain than if we just look at the sample data.



So for example, if we look at this signal in time domain, it's not very clear what it contains. However, once we look at it in the frequency domain, you can see that the signal consists mostly of a few peaks, right? So there's one peak here. There's another peak here. They form harmonics.

So we can see what are the dominating frequencies in the spectrum, which let's us process and understand this signal much better. And because the Discrete Fourier Transform is a ubiquitous tool and it's used all over the place, it's used for audio/video, in particular for compression, algorithms such as JPEG or MPEG, it's very useful as a computational tool, for example, for locking in GPS devices. It's used in radar. It's used in genetic sequencing and in many, many other applications.

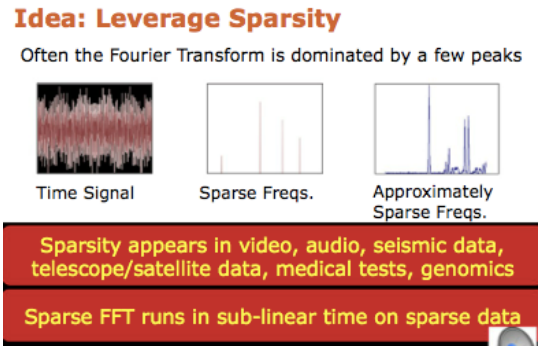
So how quickly can we perform this task? Well, the most naive algorithm for performing Discrete Fourier Transform takes quadratic time. Unfortunately, quadratic time's typically too big in order to be applicable to massive data.

Fortunately, in 1965, Cooley and Tukey introduced a much faster algorithm for this problem, called Fast Fourier Transform. And this algorithm computes all the frequencies in time n log n. So this is a much more efficient algorithm, in particular, when n is large.

Unfortunately, even n log n is too slow for big data problems. And often, we'd like to solve this problem in linear time, or ideally, even better, in sublinear time, so that we don't even have to make the whole pass over the signal.

So the question is, can we design an algorithm for the Fourier Transform that is sublinear in the signal? So in fact, the answer is now that we can. And the basic idea behind this much more efficient algorithm for the Fast Fourier Transform is to leverage the sparcity of the signal. So what does this mean?

Well often, for many signals occurring in practice, the spectrum of the signal is dominated by a small number of peaks, OK? So for example, in many situations, if you look at the spectrum of the signal, you see that the spectrum is dominated by a small number of large coefficients.



So this is what sparsity means and, of course, the restriction over the signals. However, it's not a very severe restriction because, in fact, sparse signals occur in all of the applications that I mentioned at the beginning of this segment, right?

So we have a sparsity of video, audio, seismic data, and so on. And in fact, for video and audio, sparsity is the main reason why compression works. So the algorithm we'll design, which is called Sparse FFT, will run in time which is sublinear in the signal length, given signals which are sparse.

So what are the benefits of a Sparse Fourier Transform? Well, it runs in sublinear time, which means that it takes only a small number of samples of the signal, which means that the computation is done much faster, because it's done only on the sample of the signal, which means that it's scalable to much larger data sets.

Another benefit of Sparse Fourier Transform is that, well, it uses only samples of the data, which means that only the data elements that are sampled actually have to be acquired in the first place. In many situations, it is the acquisition time, which is actually very costly.

So since we don't have to acquire all the data, it reduces the acquisition time. And it also means that the data, even if it's acquired, it doesn't have to be communicated through the network, which means that it also reduces the communication bandwidth.

Last, but not least, both the faster computation and reduced acquisition time means that the whole process takes much less power, which is of great importance if the computation is performed on, for example, a handheld device, where energy is the main bottleneck.

So these are the benefits of the Sparse Fourier Transform.

So let's see how the Sparse Fourier Transform algorithm works. Now, on the high level, the algorithm is due to two very simple ideas, which is to split the signal into bins. And the second is to estimate, which is to estimate the energy of the signal inside those bins.

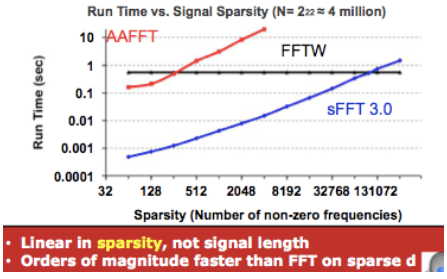
So let me get into a bit more detail about each of the steps. The ***bucketization*** step divides a spectrum into a small number of bins. And the main benefit of it is that we can design, using sampling, a very efficient algorithm, which can detect quickly which buckets or bins are empty, OK?

So in this example, we can see that there is only very small number of no-zero entries, which means that, for example, this bin, this bin, all of these bins are, in fact, empty. Now, once we detect bins which are empty, it means that we can ignore them. For the rest of the computation, we can only focus our attention on bins which are non-empty. And this is exactly what the second step does, OK?

Once we quickly identify the number of bins which are not empty, we can now focus on each of those bins, and we can design a very efficient algorithm, which identifies the position, as well as the value, of each frequency in each non-empty bucket.

So to recap, the first step lets us restrict our attention to a small fraction of the spectrum, which contains the interesting information. And the second step looks at the buckets, which contain this information, and identifies exactly where the interesting coefficient is, and what is its value. So that's the basic idea behind the algorithm.

Now let's see how this algorithm performs. So to understand the performance of the algorithm, we perform experiments on a rather large signals of a size, a few million. And now we compare the performance of the Sparse Fourier Transform to standard Fast Fourier Transform, which does not take the sparsity into account. And we measure the running time of each of these algorithms as a function of the sparsity. So let's see how this algorithm compares to the sparsities of the algorithms that we discussed earlier in this module.



Well, the first curve, in red, shows the performance of an earlier version, older version, of the sparse Fourier algorithm called AAFFT, which stands for the Ann Arbor Fourier Transform. And you see that unlike standard FFT, AAFFT running time does depend on the sparsity. In particular, you can see that for low values of sparsity, the running time improves over FFT by almost an order of magnitude. So this is a very efficient algorithm for low sparsity. However, its running time increases with sparsity, and for

sparsity which is up to around a few hundred, there is a break point, and from now on, the algorithm runs slower than FFT. For the third curve, the peak performance of the most descent sparse of the algorithm, called sFFT 3.0, and what you can see is that, as the previous algorithm, this algorithm also scales with sparsity, all right?

So it's particularly efficient for the low values of sparsity, and you can see that if the sparsity is very low, the running time improves over FFT by orders of magnitude. The running time also increases as sparsity increases, and for the sparsity value of around 100,000, the algorithm also become slower than FFT, as before. However, you can see that for a very wide range of sparsities, up to hundreds of thousands, the algorithm, in fact, improves over FFT, sometimes very substantially.

And the reason for that is that our sparse FFT algorithm has running time which is linear in the sparsity, as opposed to signal length, which means that the performance scales, according to sparsity. So for very low values of sparsity, the algorithms are actually much more efficient than the FFT, in some situations, even by orders of magnitude.

So this concludes the description of the sparse FFT algorithm, as well as its performance.

So let me give an overview of the most recent research directions that we are pursuing with this algorithm.

So one very recent and very interesting line of research that we are involved in is to build this algorithm in hardware. And very recently, we managed to build a chip, which performs Sparse Fourier Transform in hardware over really, very large signals. And we are also looking at the application of sparse FFT to medical imaging, in particular to technology such as NMR and MRI. This concludes the segment on the Sparse Fourier Transform, as well as the whole module.

So to recap, in this module we have seen two basic approaches for processing massive data, namely streaming and sampling. Both streaming and sampling have their pros and cons. But both are very efficient ways to process a massive amount of data using only very limited resources.

7.3 Data Compression: Daniela Rus

I would like to talk with you about data compression for big-data analytics. We will look at techniques for learning big-data patterns from tiny coresets.

What does this mean? We are going to examine how to extract, in a very efficient way, the most meaningful parts of our datasets.

Here's the intuition. Say you have an algorithm that runs well on a small dataset, and now, all of a sudden, you have to run the algorithm on a huge dataset. What options do you have?

Well, if you have to look at every data item, your algorithm might not ever finish. It might be intractable. One option is to try to make the algorithm more efficient to improve its running time. But another option is to leave the algorithm as is, and focus on the data.

Trying to extract the most meaningful portions of your data stream in such a way that running the original algorithm on the entire dataset will have the same performance as running the algorithm on this vastly reduced dataset.

How are we going to do this? We're going to develop a technique called ***coresets***.

But first let's look at an example.We have a lot of devices today that are producing a lot of data. In fact, in 2012, we have produced 2.5 quintillion data bytes per day. That's a lot of data. This data comes from phones, from cameras, from our computers, from our sensors embedded in the world.

Let's look at an example. Let's look at the data that comes just from our phones. And in fact, let's focus on one type of data. Let's just focus on GPS data. Here's our smartphone. Each GPS data packet is about 100 bytes. If we log 100 bytes every 10 seconds, we end up with about 0.4 megabytes of data per hour, or 100 megabytes of data per day.That means 0.1 gigabytes of data per day, per device. Wow, that's a lot of data.

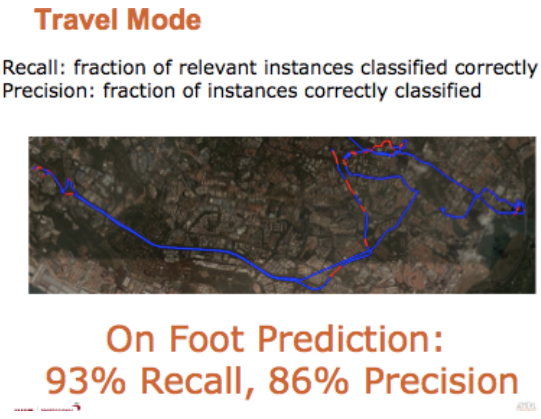
But now, if we consider many phones producing data, let's see how the numbers look. In 2010, over 300 million smartphones were sold. Let's say only a third of these phones, about 100 million of them, produce data. So for 100 million devices, we have 10 petabytes of data per day. That's about 10,000 terabytes per day. If you take your favorite external drive that stores 2 terabytes of data, you need 5,000 of such devices in order to store all this data for just one day. That's a lot of data.

So how useful is it to capture such a lot of data? Let's look at an example, which captures a day in my life, in the form of a movie. I'm represented by the beautiful, yellow icon in the movie. In this example, my phone keeps track of all my activities. I start the logging at work, then I drive home. I walk to a restaurant to have dinner, and then I drive to a shopping area. I can extract all this semantic information about my activities just by looking at GPS data. Keeping track of this history, in the form of a diary, is not only exciting as a way of logging my life, but it's also useful.

You can create a system that can automatically keep track of your work travel, work meals, and work meetings for tax purposes. We can keep track of where we go and who we meet. And if many people do this all together, the system can synthesize all the relevant correlations. And we can do much more.

We can figure out which of our travel was on wheels, and which of our travel was on foot. In other words, we can take GPS data and extract quite reliably the travel mode.

How reliable? We can measure reliability using two important measures used in information retrieval. The first measure is ***recall***. Recall tells us the fraction of relevant instances classified correctly. In other words, how many of the data points we extracted were extracted correctly? The precision tells us the fraction of instances that were correctly classified. In other words, if we look at all the data we have classified, in our example, on foot or with wheels, which of those data points were classified correctly?



For on-foot predictions, the recall is very high, 93%. The precision is 86%. This is because, at rush time, you can often walk faster than you drive. For wheels prediction, the numbers are high along both axes.

The recall is about 96%, and the precision is about 98%. So what else can we do with this data? It turns out we don't walk randomly on the planet using a Brownian motion. In fact, we tend to have patterns. We tend to go to the same places with a certain degree of repetition. And we can determine our normal activities, and we can detect what is abnormal, on a day by day basis. This is all very useful information, which we can extract just by looking at GPS data.

For example, we can take the GPS signal, which you can see on the left, and with our coreset data-compression analytics, we can map that onto textual descriptions of our activities that correspond to geographic locations. And then we can further take that information and map it into real activity, semantic activities. Have we been drinking coffee? Working on homework? Doing the dishes, et cetera?

There are many challenges associated with solving and enabling all these applications. The first one is that storing data on smartphones, and in general, on sensor nodes, is very expensive. Transmitting the data is also expensive, and it's very hard to interpret raw data. The GPS signal consists of three numbers.

How do we know those three numbers correspond to a particular geographic location? Our techniques are going to help us address how to mitigate these challenges, by using the notion of coreset.

And here is the intuition. We will address these challenges by a computational technique that will allow us to find the right data from big data. Say you're given a huge dataset, billions of points, which are represented by the blue dots in the slide.

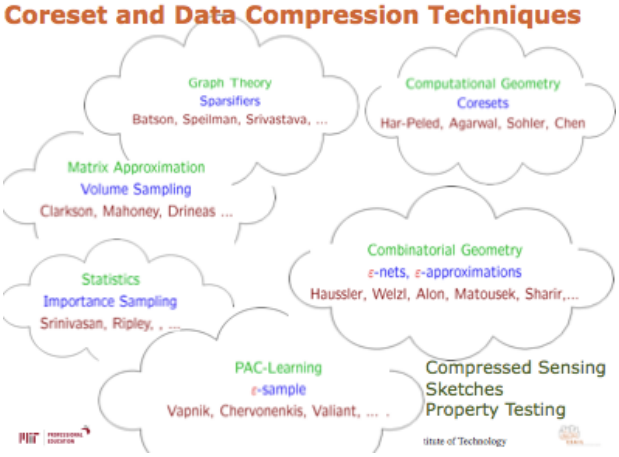
The set is called D. And say you have an algorithm called A. And trying to run the algorithm on the dataset, D, is intractable. Takes too much time. The question is, can we efficiently reduce our dataset, D, to subset C, denoted by the red points in my slide, so that running the algorithm on the subset, C, is fast? And the solution computed by the algorithm on C is approximately the same as the solution computed on the entire set, D, which we can't do, because there's too much data.

The challenge is to find C very fast, and to be able to prove that running the algorithm on the two different sets gives approximately the same answer. In order to provide concrete solutions for how to compute this special, magical set, C, I would like to introduce our big-data computation model.

In this model, we assume an infinite stream of vectors. We have seen n vectors so far, but there are more coming up. We only have log n memory to store the data, so we clearly cannot keep all the data around.

We have M processors, which will enable parallelism. And we wish to have insertions for each data point be efficient. In other words, they should take on the order of log n, over M time.

The coreset techniques we will describe in the rest of this module belong to a wide set of techniques for data compression that have been developed by various communities in computer science.



Coresets originated with a computational geometry community. And you see some of the key names of authors you might want to consult if you'd like to learn more. But there are other techniques that have the same spirit as our approach to data compression by coresets.

In graph theory, people have developed sparsifiers. In matrix approximation, volume sampling. In statistics we have important sampling. In PAC learning we have epsilon samples. And in combinatorial geometry, you have epsilon nets and epsilon approximations.

If you would like to learn more about data compression for big data, I encourage you to follow up with some of these references. Next we will look at some concrete examples for constructing coresets.

References for Compression

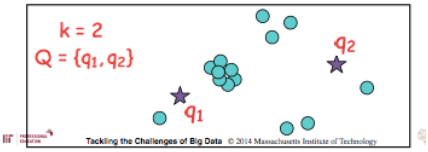
- k-Means [Feldman, Langberg, STOC'11]
- k-Segments [Feldman, Sung, Rus, ACM-GIS'12]
- Text Mining (LSA/PCA) [Feldman, Sohler, SODA'13]
- k-Lines [Feldman, Fiat, Sharir, FOCS'09]
- Mixture of Gaussians [Feldman, Krause, NIPS'11]
- Google Pagerank [Feldman, Yahoo! Research]
- Image Compression __ [Feldman, Sochen, J. of Math. Image & Vision]
- Video Compression __ [Feldman, Newman, submitted]

We're going to look at the concrete example for corsets. Specifically, we're going to look at the k-median queries problem. The input to this problem is a set of points, the blue point in this picture.

The blue points are drawn in the plane, but they can live in a d-dimensional space. We are also given a query.

K-Median Queries

- Input: $P \subseteq R^d$
- Query: A set Q of k points



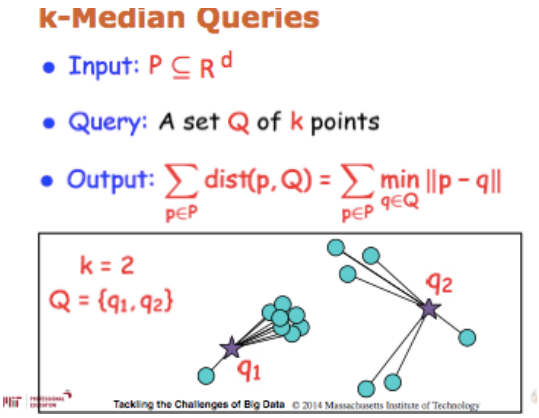
The query consists of a set of k points, q1 and q2 in this picture. The objective is to compute the sum of minimum distances from the initial set p to the query set Q.

And you can see what this might look like intuitively in this diagram. The points in this part of the picture are closer to q2. Therefore, we measure the distance between the blue points to q2. The point in this part of the picture are closer to q1, therefore, we measure the distance between q1 and these points.

Now, we can compute the output using a naive algorithm that looks at the pairwise distances between all the blue points and all the k points in the set Q. However, this may take a huge amount of time. If the set p has billions of points, it might not even be possible to compute these distances in a tractable way.

So what can we do? If I give you the points today and the query tomorrow, can you do something overnight so that, tomorrow, when the query arrives, you can answer it very fast? And the intuition is the

following. We can replace many points by one weighted representative. In fact, if you look at the collection of my points, you can see that the points naturally cluster.



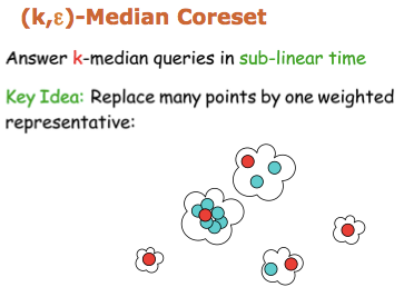
Some clusters are bigger. Some clusters are smaller. Is it possible to choose a representative point, the red points in this picture, such that each of these clusters is represented exactly by one of these points?

And is it possible to do so without altering the overall weighted sum of distances? We can formalize this intuition in the following mathematical definition for a k epsilon median coreset. We say that a set c is a k, epsilon coreset for a set of points, p, if the sum of distances between our original set of points p and the query points is approximately the same as the weighted sum of distances between these carefully chosen points in the coreset c and the query. The reason we need to do a weighted sum is because some clusters are big and some clusters are small.

And we need to weight the size of the cluster in the manner in which we compute the sum of distances. Why do we need all this mathematical machinery? Wouldn't it be easier to just do naive uniform sampling?

And the answer is no because if we were to do naive sampling, we leave out the outliers, or the small clusters, that occur in our data set. And those clusters might have some of the most important information.

We want to make sure that all the parts of our data set get represented. Another option is to consider weighted sampling. In weighted sampling, we need to figure out a way to define importance because importance will allow us to do biased sampling.



So here's the idea. Far away points get high probability and low weight. Nearby points can get low probability and high weight. Now, putting this together, here's how we might create an algorithm for computing a sampling distribution for the coreset. We take our data. We cluster it with a very simple, straightforward algorithm and find a good first guess. Then we sample with respect to distance and one over the number of points in each of the resulting clusters. This will allow points in sparse cells to get more mass and points far away from centers to get more mass.

All together, we can put these results in the following theorem that tells you how to compute a distribution and the coreset for every problem in the world. Understanding the mathematical details of this theorem is beyond the scope of this module.

However, for those of you interested in digging deeper in this space, I encourage you to consult the reference. For the purposes of this model, an instantiation of this theorem says that the coreset for k means can be computed by choosing points from the distribution defined by our sensitivity formula.This is a straightforward distribution to compute. And it also tells us that the size of the coreset is actually very reasonable. It's proportional to the dimensionality of the space in which you work, d, to the size of the query set, k, and to the error you're willing to tolerate as you compute your answer.

Now computing with corsets is very exciting. And there are some properties that we will be able to use in any problem we want to compute coresets for in the future. Here's something interesting. You can

merge two coresets to obtain another coreset. You can take the merged coreset and compress it further. And you can compute the error you will get as a result of the second compression. The error is a little bit bigger, but not too big.

Here's another thing you can do. You can compute coresets on streams. Say you have a large data stream. You see the first type of data, and you compute its coreset. When the next chunk arrives, you can compute the coreset of that chunk. And now you can merge the two coresets into a compressed version of the two coresets. When the third data chunk arrived, you can compute a coreset for that part of the data and so forth.

You can create an entire tree of coresets by compressing coresets of coresets. The error with increase with every level in the tree. But since the tree is quite shallow, since the tree is not very high, the error will not be too large.

Finally, you can compute corsets in parallel. If you have access to multiple machines, you can do all the computation that generates our coreset tree in parallel. So now that we have a mathematical definition of a coreset, we have seen some intuition, and we have seen some of their mathematical properties, we're ready to dive into the first actual algorithm for computing corsets.

We will look at a concrete example of an algorithm for computing coresets. This example is very general, and you will be able to apply it to other types of data and other problems. Specifically, we will look at computing coresets for travel data. We look at the k-segment coresets problem. Remember the example from the introduction focused on the exciting life of Daniella?

Coresets for Travel: k-segments

The blue trajectory in this image is a summary of a day's worth of GPS points for my travel. And you can eyeball this trajectory and already see the patterns. Specifically, we would like to take a GPS stream, where we expect to have noise in every GPS point, and we would like to summarize that with a few points and a few lines to define, roughly, how I move throughout the day.

Now, this, intuitively it makes sense to do this for the following reason. Consider your drive from home to work. Not every GPS point you might collect every 10 seconds is relevant to summarizing how you moved. The only thing you need is the intersection points where you change direction, you move from one street to the other if you have those points, you can reconstruct your trajectory. And that is what is captured, intuitively, but the idea of a corset for travel data.

In this segment, we look at the algorithm for going from the red points to the blue lines. The algorithm goes through several phases, five of them, which are captured in this image, and we will understand the computation that goes behind each phase. We have a mathematical tool available to us for summarizing data points with lines, and for fitting lines to data points. This is the case k-spline.

And I want to recall that a k-spline is a sequence of k connected segments in R to the d. But our points have a lot of noise in them. For GPS data, there is a lot of built in error in our points. It's very unlikely that we will be able to approximate our points using k connected segments.

We would like to relax this assumption. In other words, we would like to create an epsilon approximation of our data points using segments that are not necessarily connected. We would like to allow them to rotate a little bit and maybe not quite meet at the same point. We can use this intuition to define the k-segment mean problem, where the objective is to find k segments that minimizes the fitting costs from points to d-dimensional signal. This signal could come from GPS. It could also come from a camera. It could come from a radar. It could come from any information source. More specifically, the inputs to our problem is a d-dimensional signal over time. And we have a query, which is k segments over time. What we would like to compute is the k piecewise linear function f over time. The output of the k-segment query problem is the sum of squared distances from p to the k segments.

And the coreset for this problem is the weighted set of segments such that the cost between the initial data set and the k segments is approximately the same as the cost between the coreset, the red dots in my chart, and the k-segments.

How can we pick these red dots so that we don't pay too much penalty in terms of the accuracy of our computation? Well, it turns out that we have a theorem that says we can do it. And we can do it always.

In other words, for every discrete signal of endpoints in R to the d, whether the signal is GPS or something else, there is a corset that's not too large. The coreset is on the order of k over epsilon squared, where epsilon is the error you're willing to tolerate. And this coreset can be computed in the big data model. How do we compute it?

Well, here's the algorithm. You have a signal of end points. You're also given how many segments you want, the constant k, and how much error you're willing to tolerate, the constant epsilon. The first step is to use some off the shelf algorithm to compute the k-segment mean. And you can see the result of this computation in the red lines on the chart.

The next step is to project our original points onto segments. We calculate the sensitivity of all points because, if you recall from our mathematical definition, coresets have weights and sensitivities associated with them to capture the signs of the cluster that they represent. And then we sample points according to these computed sensitivities of probabilities. We assign weights to each point, we select, and we output the k-segments in the selected points. And that's all there is to it. Wow, isn't this easy?

So going from our original data, we compute the k-segments mean. We sample the points. We project the points onto the segments. And then we compute the final corsets, which consists of the segments and the critical points we have selected as defining those segments.

In the next segment, we will look at how this algorithm can be used to go from GPS signals to recognizing activities associated with those signals.

In the previous segment, you saw the coreset algorithm for k mean segments. In this segment, we will look at the evaluation of this algorithm on a particular dataset. For this data, for our evaluation, we use the taxi dataset from San Francisco.

There are 500 taxis logging GPS data every minute or so, and specifically focus on a subset consisting of 100 taxis. In our first example, we look at data from one taxi only, and the data arrives under a streaming model.

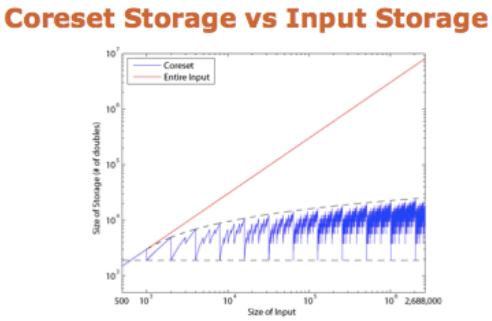
<Diagram>

When the first section of data arrives, we can compute the coreset, which is represented as the red dots and blue line segments in this chart. When the second part of the trajectory arrives, we can compute the corset for that piece of the trajectory, and then we can apply the coreset merge theorem to merge the two coresets into a composed coreset. When the third and fourth trajectory chunks arrive, their coresets can be computed, and the merge coreset can be done as well.

Now the two merged coresets could be computed again, and you can see that the general travel pattern is emerging in the story that is summarized by the coreset. Now, we can evaluate how well this computation does with respect to two critical resources,

- How much space does it take to do this computation?
- And how much time does it take to do the computation?

For space, we use the same data that you've seen this chart, and we plot where is the size of the input on the x-axis. In other words, how many data points do we see? And what is the size of the storage required to summarize the trajectory? In red, you see the input, and you see the input growing over time.



In blue, you see the size of storage required by the coreset, and you can see that the size is approximately constant throughout time, no matter how many data points you receive. The reason for this is because we have the ability to compute individual coresets, and then we can merge coresets into a composed coreset, therefore computing this tree of coresets. And, if you recall from our second

segment, we know that, even though the error increases from level to level in this tree, the overall error will not increase too much because the tree will be shallow.

For computation, the main benefit of the coreset is as we apply it only to the coreset points, not to the entire dataset. Because of that, the construction time will be proportional to the size of the coreset, not to the size of the input. If we're looking for k segments, our smallest algorithm for k segments runs in cubic time.

Therefore, the complexity for computing on the coreset points is on the order of k to the third, whereas the coreset for computing using all the data points is on the order of n to the third. We can do further optimizations by computing the coresets in parallel. If we have access to multiple machines, each level in this tree of coresets can be computed in parallel.

So what is the bottom line? We're looking at 100-segment mean on GPS traces using 100 taxis. The data is average over the trajectories of all these 100 taxis. On the x-axis we have the size of input. And on the y-axis we have the total cost. In other words, how much time does it take to compute the 100-segment mean using the points represented by the x-axis? Using the entire dataset, we can compute only about this much in about a week.

So that's totally impractical. However, if we use coresets, we can compute the 100-segment mean for the entire dataset, consisting of over 2 million points, in approximately one minute. That is savings. The next question you might ask is, how well do we do, accuracy-wise? And in this chart we show again the size of the input and the relative cost. In this case, the size of the input refers to how many points we have used for our coresets, and in this test we compare the coreset method against a method that selects points using uniform random sampling.

In red, you see the relative cost for coresets. The lower the curve, the better the error. And you see that for coresets the error stays low, starting early on. So, whether we use the coreset of about 20 points, or 200 points, or 600 points, the error is approximately the same. The error is quite low.

However, for uniform random sampling, you see that if the science of the inputs you consider is low, around here, the cost, the error, is very high. If you select a lot of points on the order of 1,000, the two methods to begin to converge. But why is 1,000 points when you can get away with 50 or 75 or 100?

So I hope I convinced you that coresets are wonderful tools to use for your problems. They don't use as much space, they don't use much time, and they are very accurate. I have become extremely passionate by coresets, and I see them everywhere I look, at work or in life, and I hope you will find a way to use coresets in your own activities at work.

7.3.1 Core Sets Use Case: Life Logging System

In the previous segment, you saw the algorithm for computing the k-segment mean coreset. In this segment, we will look at an application of this algorithm to create a novel life-logging application, called **iDiary**.

iDiary allows users to upload their GPS data streams. And from these GPS data streams, iDiary automatically creates a storyboard of what the user did, where, and when. Let's start with an example that shows you how the coreset computation enables efficient processing in iDiary. In this video, you see the benefits of using coresets with travel data.

Our user is taking a Duck Boat tour in Boston. And you can see that in the top panel, the camera feed from the user's phone. The user's phone is also collecting GPS data. At fixed points in time, the GPS data stream is processed for coresets. And on the left, you'll see the number of points used by the k-segment mean algorithm, using the entire data set.

And on the right, you'll see that the same k-segement mean can be computed with many fewer points. Dropping from 1,500 points to on the order of 40 points. And you see that the two maps computed by the algorithm roughly line up. So the benefits for using coresets are tremendous.

iDiary can do much more. iDiary can associate activities with a user's geographic location. And wow, this particular user spent an awful time at home, and not very much time at work. iDiary also supports correlations between your activities, and the activities of your friends.

For example, iDiary can compute when two people find each other at the same location, at the same time. The system architecture is presented in this chart. The input to the system consists of GPS data streams coming from the user's iPhone. The system is able to give the user a query interface, and also an interface that allows the user to aggregate historic data about the system.

The system takes the raw data, databases it, processes it through the coreset computation, processes the data even further, to identify the patterns in the user's activity, and outputs the results.

In the next segment, we will focus on the bottom part of the diagram. You have already seen the coreset algorithm from the previous segment. And the databasing of GPS data is pretty standard, off the shelf. The system has the ability of taking raw GPS data, for example, as in this sample data set, map

this GPS data into a map, and then convert this map into street directions that can be presented to the user in human-readable form, in text.

This human-readable form can also be used as the basis for posing queries to the system. So this is really cool. Starting from millions and millions of GPS data points, you can actually come up with a textual description, or a story line, of what the user did, and when.

The system is able to go from GPS points to textual descriptions using two ingredients. The first ingredient is the coreset creation, because the coreset creation tells us which are the points that should be checked as special points in the system. If the trajectory of the user includes two million data points, we don't have to check every single one of those points.

And how do we check the points? Tools, such as reverse geocoding, can be used against those GPS points to find the exact street address corresponding to that GPS point.

In this segment, we have looked at how coresets enable the functionality of iDiary.

iDiary is a life-logging system that is able to take raw GPS data, and convert that data automatically into a story line that can be presented to the user in text. That can also be queried using regular vanilla textual queries.

In the next segment, we will look at the algorithm that enables iDiary to go from geographic locations, to activities associated with those locations.

We saw how the k-segment mean algorithm enables new capabilities in the iDiary system. We can go from streams of GPS data to textual descriptions of movement patterns and to activities.

In this segment, we will look at the algorithm that allows us to go from textual descriptions of users' trajectories to actual activities. In other words, to semantics of what the user did and at what time. And the intuition is that most of us do not walk randomly on the planet. In fact, we all have a pattern of movements and activities that repeat fairly consistently from day to day.

I go to my children's school, work, my favorite shopping store, and my favorite gas station. And I bet all of you tend to do the same. Here is the sample trajectory from one of my data sets. And if you look carefully at the data on the left, you see that the trajectory captured by the first segment of my GPS stream and summarized by the core set has roughly the same pattern as the second and third segment.

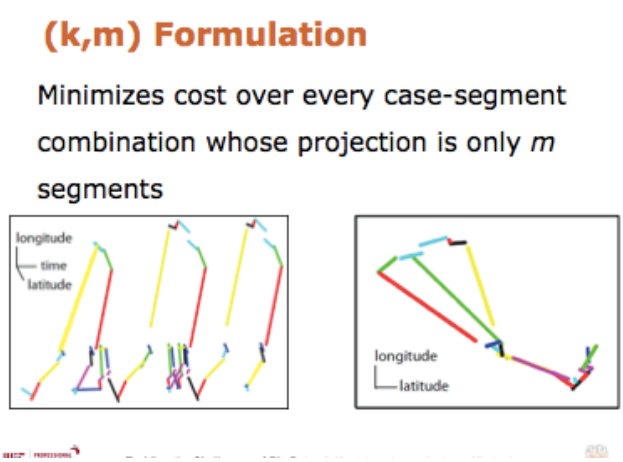
What we would like to do is recognize this pattern and produce a general activity map, which is captured on the right. This is precisely what the k,m problem formulation captures. The objective is to minimize the cost of every case segment combination whose projection is only m segments.

The input to the problem is GPS points, and the desired output is the m locations the user visited. We would further like for these m locations to be described in textual form. And they would like to have the semantic activity that took place at that location.

The intuition behind going from physical locations to activities is the following, if you're in a grocery store, you're very likely to be shopping. If you're at Starbucks, you're very likely to be drinking coffee. If you're at MIT, you're very likely to be inventing the future. So how do we go from physical descriptions to these activities?

The answer is to use the world's knowledge, which is captured in databases such as Yelp, Yelp associates activities with addresses, or even beyond Yelp going to the Wikipedia. This is what our iDiary system has the ability to do. Let's look at the algorithm for determining the m locations in greater detail.

The input to the algorithm is a GPS signal, in other words, a sequence of geographic points processing through time. This is derived from an entity's repeated travel of a set of distinct geographic paths.



The right diagram shows a user's trajectory over time where the z-axis captures time. The left side shows the projection of all these trajectories in a plane, and you can see that a beautiful map emerges out of the user's trajectory over time. Now we can eyeball this data and immediately recognize this underlying pattern.

How can an algorithm do it? The objective of our k, m segment trajectory clustering algorithm is to partition the signal into k sections, to group the sections into m clusters, and assign a line segment to each cluster.

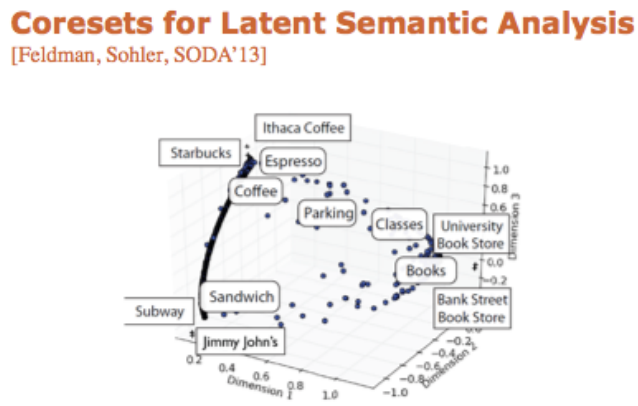
This is a generalization of the k segment mean algorithm you have seen in the previous segment.

The objective is to select the k, m partition in such a way that we minimize the point signal's fitting cost. The k, m segment mean algorithm uses expectation maximization to find the local minimum of the fitting cost, and the algorithm is very efficient in run time and space.

Here's how it works. You start with a GPS signal. There is no underlying roadmap. You partition the signal into k distinct sections using an off the shelf algorithm, for example, the Ramer-Douglas-Peucker algorithm is a good first guess. Next, we take the output of this algorithm and we group k sections into m clusters, which are denoted by different colors in this graph. We use the k means for a good first guess. We calculate the segment mean for each cluster, and then holding fixed the odd partition boundaries, we can play with the locations of the even partition boundaries, and then reverse. We hold fixed the even partition boundaries and move the red partition boundaries. This process repeats as long as the cost of the partition continues to drop. When the cost stabilizes or starts going up, we complete the loop and we produce the partition.

Upon termination, the locally optimal k, m segment trajectory has been found, the line segments give the map, and the special coreset points along the map give the locations where interesting activities took place. The output of the k, m segmentation algorithm can be further used to go from street addresses to an actual story that describes the user's activity at that address. And the magic sauce for doing this was actually invented by the information retrieval community in the form of a technique called ***latent semantic analysis***.

It is beyond the scope of this module to explain latent semantic analysis, but if you're interested, just google it. You will find a lot of information about it. At the high level, the idea is that we can compare the geographic descriptions of the locations frequented by our users against the world's knowledge associated with those locations. The iDiary system happens to use the Yelp database, but any other database can be used in this step. You can even use the Wikipedia.



So what do you get using latent semantic analysis? The words described in the Yelp database can be used to associate Starbucks with coffee and espresso, or the university bookstore with classes and books, or Jimmy John's or Subway with sandwiches.

In other words, we have the ability of going from GPS points to textual descriptions of street addresses and semantic activities associated with those street addresses. All together, this provides a story line which has been computed automatically from the GPS stream and has been enabled by the coreset.

The coreset not only speeds up the computation, but also helps us identify the critical points in our data. And for those points only, we run the latent semantic analysis in order to associate activities with those locations.

I hope I have convinced you that coresets provide you with a very powerful tool for solving a lot of problems. In this module, we looked at one particular data source, the GPS. However coresets are broadly applicable to lots and lots of other types of data.

You can use them on image streams, you can use them on laser streams, you can use them on streams from just about any other sensor system, provided there is a time element and locality associated with your data.

So in the future I hope you will find ways to use coresets in your work because coresets won't take too much time, they won't take too much space, and they will give you an accurate result.

7.4 Machine Learning Tools: Tommi Jaakkola

Today we'll talk about machine learning tools for big data analytics. I have a group of seven people at MIT working on machine learning. Machine learning, you can think of it as computer programs that learn from experience.

There are lots of issues from theory algorithms to applications related to such methods. So, for example, from a theoretical perspective, we try to understand how learning is possible, when it is possible, when it is not, how much experience is required for learning anything, and what guarantees we might be able to give for the outcome of learning.

Algorithms are needed to effortlessly implement those computer programs. And there are lots of applications that we are trying to apply these methods to, and serve as a motivation for the underlying theory and algorithms.

So, for example, we work on ***natural language processing***, trying to understand how computers could understand natural language documents. If we could do that, we could have very natural user interfaces for computers. We could ask computer to search for information. We could translate from one language to another, and so on.

Also, nowadays the access to information is almost invariably through ***recommendations***. If you type keywords on Google, you will get back recommendations, what Google might think that you're likely searching for. Such recommendations are used for interfaces all over the place, whether you visit amazon.com site, whether you are watching movies, you get recommendations for movies that you would like to see, and so on.

There are lots of other applications that we are motivated by that you will see here, from ***predicting user behavior***, to ***reverse engineering biological systems***, and so on.

Let me start with a brief introduction to what is machine learning. So as I said, machine learning is about computer programs that learn from experience. At the core of machine learning problems is the problem of predicting, forecasting, trying to predict what is the outcome of future events, how financial assets might evolve. We can try to predict outcome of medical tests to save time and resources, and so on. What distinguishes machine learning methods from other ways to carry out predictions is that we are really trying to learn those from data.

So let's take a particular example here. Let's say we're trying to predict whether credit card transaction is fraudulent or not. Now, we could try to solve this as a traditional engineering problem, ask an expert to write detailed rules that characterize whether that transaction would be fraudulent or not. This would be time-consuming, requires expert knowledge, and may not be all that accurate.

It is not guided so much how accurate the predictions would be. Instead, it is much easier to just list a few examples of outcomes that you would want to the predictor to produce. For example, given a description here, a description of a credit card transaction in terms of time and location and so on, you will get, for a particular description, you actually went ahead and investigated that, and you got an outcome, whether it was a fraudulent or not. Another description, another outcome. And you get few of these. This available data now represents a guiding data for how your predictors should work, how it should map descriptions to outcomes.

And machine learning methods really are methods that try to learn from such data to produce that accurate predictor. Now, many of the other speakers will talk about classifiers. and we just mentioned classifier on the previous slide. The mapping that we wanted from credit card transactions to outcomes was a classifier. So let's look at that in a little bit more detail.

So you have transaction here, description of transactions. You map those to a set of features here that describe that transaction. So time, location, company involved, user involved, and so forth. You can compile these features into a vector. So you have now a vector that's called a feature vector, that represents the transaction. Now, this feature vector is in the form that computers can understand. And we can now learn functions that map from those feature vectors to plus-minus 1 outcome, whether their description was fraudulent or not.

So we try to explore here a set of possible classifiers. We parameterize that set, parameterize that set of functions, that map from feature vectors to outcomes. And try to pick out of that set one that best matches the experience that we have, the data that we have in terms of the records that I showed you earlier.

Now, what has made machine learning very popular is that these type of problems, where you have to learn from experience, are very common. You might be trying to classify news articles, what you might like to see. You give few news articles, you rate them. And now you ask, give me other news articles that you would also like. You might look at images and ask whether a particular person is in the image.

And now give few examples where that person is or is not there and learn from that to make predictions.

You can classify biological samples, tissue samples into whether it contains tumor or not, and so on. So these classification problems are all over the place. And you will see them all over the place.

In terms of effort in machine learning, this represents decades of work. And lots of algorithms have been developed to solve these type of problems very efficiently, very scalably and accurately.

Particular class of such algorithms that are very appropriate in the big data context are called online algorithms. What it means is that the algorithm simply scans through the data or subset of the data, and tries to nudge, based on each available record, in the direction that makes the predictions more accurate. And a series of these in little nudges now allows you to create very accurate mapping from the feature vectors that you have to the outcome.

So to summarize that, you have a set of records from trends that describe transactions. You have a set of outcomes. And what we are trying to fill in, what happens in between in terms of predictions.

Now, today we will not talk about classification methods. We'll try to go beyond simple classifiers in two ways.

First, over the past decades, machine learning methods have been used much more prominently to make predictions that are not just binary outcomes, fraudulent or not, plus-minus 1, yes or no. But we are actually predicting more complex objects. We are predicting how to annotate a genome. We are predicting what word underlied an acoustic waveform, so transcribe speech into words. And we are looking at how to map sentences into syntactic parses that help us understand how to structure that sentence to translate into different languages to answer intelligent human queries.

Now, when we are making more complex predictions, we have to ask, in the big data context, do such methods scale? And that is one point that we are going to be focusing on.

Now, another part of that we will focus on is to try to understand how to better use large, fragmented, incomplete, erroneous, and noisy databases.

So let's take a particular example. Let's take a visitor to Amazon.com. And our task here is to recommend what that customer might like to buy. So if we have seen that customer only couple of times, we have very limited experience of what they would actually like to purchase. However, we have lots of users. Millions of users also have visited that site. The question here is now how can we leverage the experience of others to make better predictions for an individual user? And if we can do that effectively, we can make better use of large databases that are now increasingly available.

So to summarize, simple classification problems have decades of work in machine learning, very effective solutions, very scalable solutions. And you will hear about such methods in other talks. What we are going to be talking about today is going beyond classification, predicting more complex objects than just plus-minus 1 labels for the instances that we see. And we also try to better leverage incomplete, fragmented databases for the purposes of making accurate recommendations.

7.4.1 Example: Scaling Structured Prediction

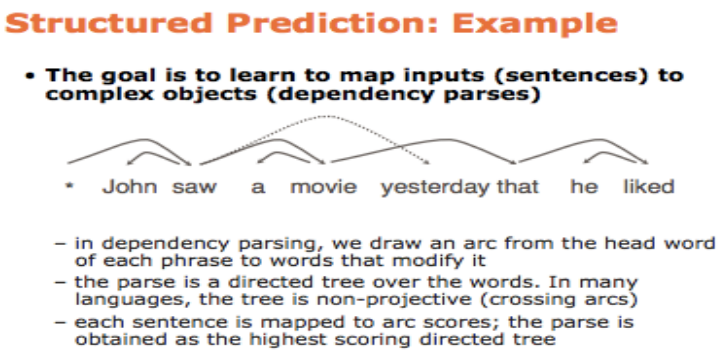
I will talk about machine learning tools for big data analytics, and in particular, scaling structured prediction.

So structured prediction is the problem of predicting more complex objects than just plus minus one labels. That is known as a ***classification problem***. It is a structured prediction problem where the object of prediction is a complex object.

So think about a word that has lots of suffixes. We would like to segment that word into the stem and the suffixes that are meaningful in terms of understanding what the word actually means. We can parse natural language tendencies syntactically and it is that syntactic parts that is now our prediction for sentence.

You can segment image in terms of whether it contains which part of the image contains sky, which one is forest, where is the car, and so on. Now the prediction is the segmentation rather than yes no answer to whether this particular part of the image corresponds to sky or not.

So we are predicting things where there are lots of complex features in the target object. This is known as a ***structured prediction problem*** and it appears all over the place from natural language processing, computer vision computational biology, to other areas.



So let's make this a little bit more concrete and take an example. So our example here is in natural language processing, syntactic parsing. So here's a sentence. John saw a movie yesterday that he liked. Now we would like the computer to understand this sentence. And in doing that we want to automatically parse that sentence. And what parse here means is to draw little arcs between the words that tell you how head words, so-called head word, what other words modify that word. So, for example, it was movie that he likes. So we draw an arc here from movie to that. That he liked is a phrase here. So we can parse the sentence into phrases.It was movie that he liked. It was yesterday that it happened. It was John that did the seeing, and so on.

This helps the computer to understand what the sentence really means. So our task here, from a machine learning perspective, is to take sentences, and parses, and learn to produce such complicated objects, the trees, over the words in the sentence, and learn to do that from examples. The particular parse that you see here is something called dependency parsing where we just draw an arc from a head word to words that modify it and it has a syntactic meaning.

We try to find a tree, directed tree over the words in the sentence, that is the object of prediction. Now somehow we have to score these alternative structures that could be used for this particular sentence.

There are exponentially many possible trees that you could use for this sentence, OK? Only typically one or a few are correct. So how do we search over this exponential space? How do we score different alternative trees that we could attribute this sentence to?

That is the structured prediction problem that we are trying to solve. Now in the big data context this has lots of scaling issues. We have now a complicated object, exponentially many possible alternatives for any particular instance that we are doing a prediction.

So how do things scale? There are many things that must scale in this context. So, for example, we have to be able to make the prediction. We have to be able to find the highest scoring tree. So what is the actual parse for a particular sentence?

Even that problem can be challenging. And in fact, in some cases provably hard. So this problem becomes hard where we try to exert fine control over which trees we prefer in a particular context of a sentence.

So we prefer fine control based on increasing accuracy of prediction, but it has computational cost.

Now estimation, or alternatively, learning relies on prediction. In order to learn, you need to be able do to assess how your prediction would differ from the target value that's given. So if prediction is hard, estimation will be hard as well. In many cases, we also have to represent ambiguity in the sentences. It may be ambiguous what actual syntactic parts is. And these are sort of in the increasing order of difficulty here from prediction to representing uncertainty.

And all of them rely on the prediction. That is the task that we will focus on today. So let's look a little bit more concretely on what this means in the context of parsing a sentence. So our problem here is mapping from a sentence here, John saw a movie yesterday that he liked, to a directed tree.

So x is the sentence and y target value is directed tree. Now, this problem here really has two components. One is ***modeling***. How far in control do we exert on the aspects of the trees so as to prefer the correct one for any particular sentence? The other one is solving the axle highest scoring tree. How do we actually make a prediction?

When we have only a simple way of scoring, so we can now say that we score a particular arc in the absence of anything else given the sentence. Then just say that the score for a full tree is the sum of its parts. In that case, we can actually solve these problems efficiently. The problem here is that we wouldn't get very accurate predictions.

We need to exert finer control over what trees to prefer in any context. So one way to do that is to look at essentially a bundle of arcs that are out-going from each word here, and exert control over what the score would be for each such bundle.

So now notice that the different arcs that are going out from say, the word "movie" here, are tied. So I don't score individually, drawing one arc movie to that, or movie to a. I will score them together. So "a movie that." Exerting such fine control improves the accuracy of the prediction.

Unfortunately, it has a consequence. In this case, finding the best tree in the context of a particular sentence is already proven pretty hard. If the prediction is hard in the context of a single sentence, how can we scale this to actually work on a big data context?

So let's see what we can do here. We have essentially two extremes here. One, if we only start simple control, simple preferences of what trees to associate with each sentence, we can solve it even simply. If we want higher accuracy, the problem becomes computationally hard. How do we resolve this? Can we do something close to refined control but in an efficient manner?

And, indeed, we can. And what we can do is really similar to how we want algorithms to scale in the big data context more generally.

We ***fragment*** the problem. We shatter the problem into pieces, where each piece is quite easy solvable. Now these pieces here, in the context of each word, what are the outgoing arcs? We can solve these individually. But there are also constrains how these pieces must fit together. These pieces together, the individual choices of outgoing arcs from each word, must form a tree in order to represent a valid part of a sentence.

So what we do here is we have a set of pieces. We blew up the problem into easily solvable sub-parts, but we must make those parts agree. And when they agree, we're actually making the same prediction as the refined scoring method would do. And typically, in many cases across many languages, we can actually make the same prediction as the more complicated prediction method would do.

This is just a simple illustration that it actually works in practice. So here you have a set of languages, and here the percent of cases, percent of sentences, where this method of blowing the problem into pieces, allowing us to solve it in parallel across loosely coupled sub-problems, actually succeeds in getting the same prediction as the more complex, refined scoring method would do. So about 98% of sentences, we can get the exact same prediction much more cheaply.

So in summary, the prediction problems for structured prediction, predicting complex objects, we can do in much the same way as you would parallelize normal algorithms. Farm it out in the small pieces that are loosely coupled. We can solve in that matter individual sentences very efficiently. And now have a method that makes accurate predictions in a scalable way.

We can leverage the ability to make predictions in terms of the other problems, as well. I will not address them today, but the methods are available in references. In the next segment, we're going to focus on leveraging large, incomplete, fragmented databases. How that can be done efficiently in the big data content.

7.4.2 Example: Collaborative Filtering

Today we'll talk about machine learning tools for big data analytics. And in particular, we'll talk about how to make recommendations. So there are lots of problems that can be formulated as recommendation problems.

If you have ever visited Netflix.com, you'll see that they provide recommendations for movies based on in the ratings that you make. So you have movies, and a particular user may have rated some of them.

Another problem would be to fill in the missing ratings in order to recommend new things for them that they have not seen. Lots of other problems can be put into this form.

The movies here could be books, they could be financial assets, they could be documents, and so on. I will use the movie and user example here, just as a concrete way of trying to understand how we can use, leverage incomplete information from across a large number of users.

So the goal here, really, is to take any particular user, say this one here, and look at their ratings. And in this case, they have only a couple of ratings of how much they liked, how many stars they assigned to a particular movie here. Now, based on just two ratings, we cannot do much, but we can leverage the fact that users tend to behave similarly.

There are groups of users that act the same. So we can try to identify other users here that have similar experience in terms of those few instances and use their experience to complement the lack of experience with this particular user.

So the idea here is to do collaborative prediction. So use experience, borrow experience, from other users to bring out predictions for any individual ones. So, for example, in this case, if our problem here is to predict whether this user would like a new movie that just came out, we would have to now look at other users, for example, this one, that had similar ratings for the ones that both of them have watched, and now supplement the experience of this user based on that.

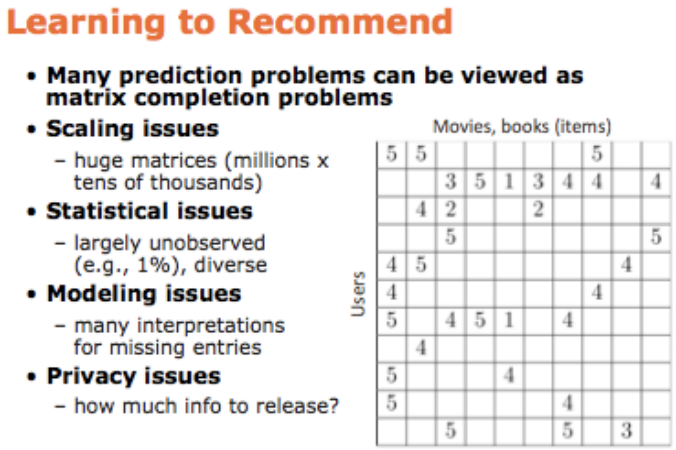
So now we have a rating for this user for that new movie based on the experience of another user. Now, there are lots of other users that are more or less similar to this one as well. So the actual predictions that we make cannot be just identifying the closest one. But the basic idea, basic intuition, is this, how to leverage other users, because people don't act in that diverse way. There are actually many similar users. So let's try to understand what the issues are in making these type of predictions.

Well, the first of all, there are ***scaling issues***. This is a big data problem. The matrices look small here, but they are actually huge. You might have tens of thousands of movies, tens of thousands of books, millions of users, and so on. So the matrices here are larger than you could possibly hold in the memory of a single computer. So the dimensions are great, and we have to deal with that scaling computationally.

There are also ***statistical issues*** here, where you have very limited experience of any individual user here. How can we possibly guarantee anything about the accuracy of their predictions? What we will recommend? Will they actually like our recommendations?

There are also ***modeling issues***, ambiguities, how to interpret the data and that we actually have. How would I interpret the fact that the user has not rated this particular movie? Maybe they saw the movie, didn't tell us that they did. Maybe they saw the movie and didn't like it, and refused to rate it, and so on. There are many possible interpretations for what these missing entries mean. And our predictions would change based on knowing what that interpretation is.

Lastly, there are ***privacy issues*** in making these recommendations. How much information are you willing to release in terms of your ratings, in return for accurate predictions of what items, books or movies you would like to see.



So this trade off between privacy and accuracy is very real in the big data context, in particular when involving users. And we'll try to talk through some of these issues and how to address them.

So let's take an example. And this is a little bit more technical, hopefully not too much, of how we might actually make these type of recommendations. So here we have a large, incomplete matrix. And for simplicity, let's assume that the missing entry here just means that the user has not viewed the particular item. And the selection of items that the user has viewed is more or less random, and they provided us ratings for the ones that they did see. Now we have a huge matrix, and we have to somehow constrain ways of filling the missing entries here.

The way we can constrain that is by factoring that matrix into a product of smaller matrices. And then predict each entry here in terms of what that product of two matrices would tell us to predict in that context. So when we make these predictions based on factoring the matrix, incomplete data matrix, into a product of smaller matrices, we are effectively finding features for users, features from movies, in this case.

So we will end up representing any particular user here in terms of a feature vector. So each user now becomes a point in space. Similar uses are close to each other. Their vectors as feature vectors will be close to each other, similar to each other. Each item here, each movie, that we focus on here also has a feature vector.

So we represent items as features and users as features. And now our prediction is looking at comparing those two features together. And the features here are parts of those matrices. So you compare how similar is this feature vector for a movie to a feature vector associated with a user? Now none of these compositions are given to us.

We must learn them. So we must learn to represent uses as features, movies as features, and then predict based on how similar the two are. This is the simplest way to solve recommender problems known as matrix factorization methods.

Now let's go back to the issues that we outlined earlier. How do we address them? The fact that these matrices are huge. Can we actually construct the product of smaller matrices efficiently on that scale?

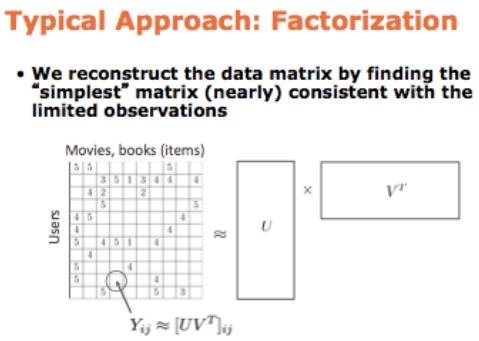
Now we can by solving these problems piecemeal. We can try to introduce very skinny feature vectors here, essentially representing each user in terms of one real number, each movie with one real number. And estimating that pair of two such skinny matrices at a time. This problem can be solved efficiently. And our actual solution then will be solved successively adding these type of features.

So we can actually scale, and people have scaled, these methods to realistic problem sizes. Now other way issue that we have is a statistical one. How can we accurately fill in missing entries here based on so limited data from its user?

We can do that by limiting the size of these matrices. How many features were used to represent each user? How many features are used to represent each movie? The fewer we use, the more constrained our predictions will be for any particular entry here.

Few data points we need to estimate the corresponding features for users and movies. So we can balance here the amount of data that we have and how constrained our factorization tells us the different entries here are. Without any constraints, having unlimited dimensions for these matrices, we could arbitrarily set each of these entries in the matrix. As a result, we would simply be storing the incomplete matrix and not learning anything. So learning here means constraining. Only through constraints we can learn something.

Now another issue here is our modeling issue. How to interpret the selections that users have made. Users do not arbitrarily rate movies. They don't arbitrarily select which movies to view. What we must model here is also their selections, which movies they chose to view in addition to how much they actually liked those movies. And it is only by decoupling these two that we can make recommendations for things that the user would like but would not otherwise find. That is the setting where these recommendations would actually be useful for users.



So more concretely, what we are modeling here, for simplicity here, in terms of ***factorizations***, the ratings, how to rate movies, but also how the movies are selected to be viewed. And by decoupling these two, we can make recommendations that are actually useful for users.

Now the last key issue here is a privacy issue. How can I trade privacy? How much information to release in return for accuracy of predictions? The more I release, the more the system can use my experience to complement others. And the more it can use to identify users that are similar to me.

So the more I release, the more accurate predictions I will get. Now, as a user, I would wish to limit that. And I would wish the system to quantify how much privacy I am losing in return for accuracy. So instead of just releasing my ratings altogether, I can answer certain queries about my ratings.

So I can, for example, answer a query about an arbitrary user that the system might send me as an abstraction of a user and answer how similar am I to that particular one? And these queries that the system would use would be such that they don't allow the system to infer, for any particular movie, what my rating would be. But they collectively tell the system something about what that user might or might not like. So we can quantify how to trade prediction accuracy with privacy. And users should be able to set that based on their own privacy considerations.

Everything I covered here today is already available and can be used to solve particular problems. We covered just two of them, how to make complex predictions and how to make recommendations. There are lots of machine learning algorithms available out there that solve all kinds of problems where we need to learn from experience.

And I hope and strongly encourage you to learn more about them and try to apply them to problems that you are interested in.