This talk is about a phenomenon that has been gaining steam over the past ten years in the back rooms of the most successful Internet companies and now is transforming diverse industries including financial services, telecommunications, rich media, e-tail, social networks, energy and defense.

In financial services companies are already storing details on every transaction to track risk and look for behaviors. Consider the number of transactions that almost happen compared to the ones that do. How many flights do you consider before you book a trip? How much time to spend browsing Amazon vs. buying? Consider the steps in a transaction vs. the steps in fulfillment.

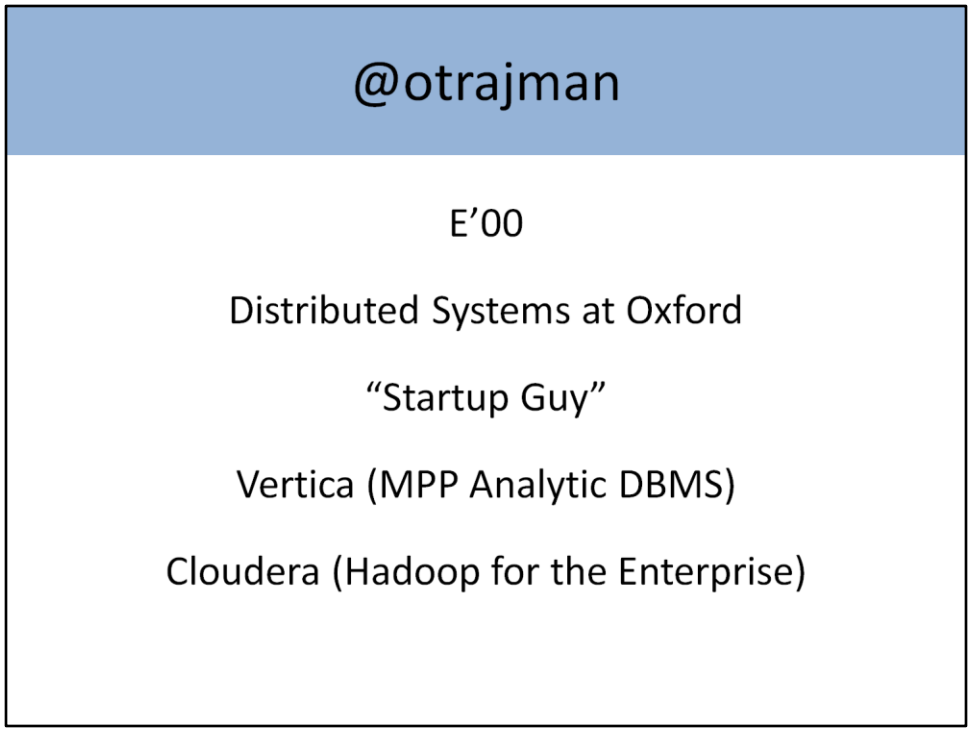
The 'how' is an order of magnitude or more greater than the 'what'. And today the 'how' is where we optimize business.

Another great story – our electrical grid was build about a hundred years ago and it's running very near capacity. Not only can we not afford it, there's huge risk in a massive overhaul. So instead we're monitoring for anomalies – sampling at 60-120hz and detecting patterns that would indicate components that need to be replaced. This is part of what's being known as the smart grid.

These algorithms are all being developed by scientists and engineers. Not in a lab or by building transaction applications but by manipulating large volumes of data

looking for patterns, building models and finding ways to optimize processes.

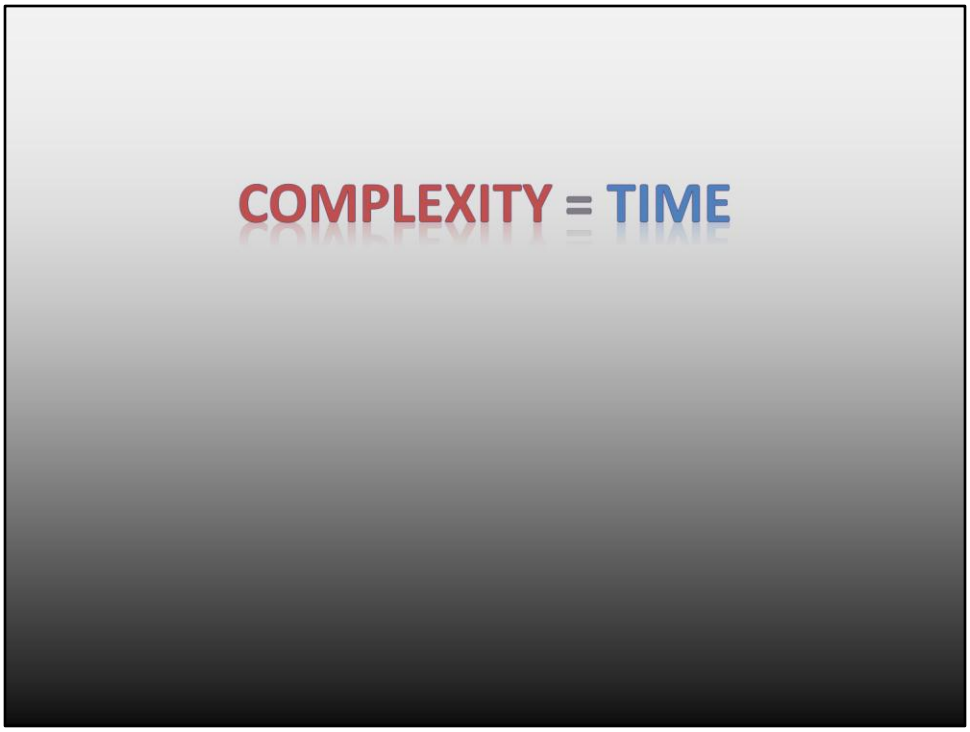
In short, the world is undergoing a major change due to what we call the rise of the data scientist.



I graduated Tufts with a degree in Computer Engineering in 2000 and a focus in distributed systems. I spent a year at Oxford studying parallel programing and distributed processing.

I'm a startup guy – my most recent two are Vertica, a massively parallel analytic database vendor founded by Mike Stonebraker and recently acquired by HP. I currently work at Cloudera who creates a distribution and provides commercial subscription for Apache Hadoop and related projects.

For the past ten years I've had the opportunity to work with some of the biggest companies solving complex data challenges. Many of these are household names and products you use today. In this talk I'll be rolling up a lot of themes and lessons learned through my experiences.



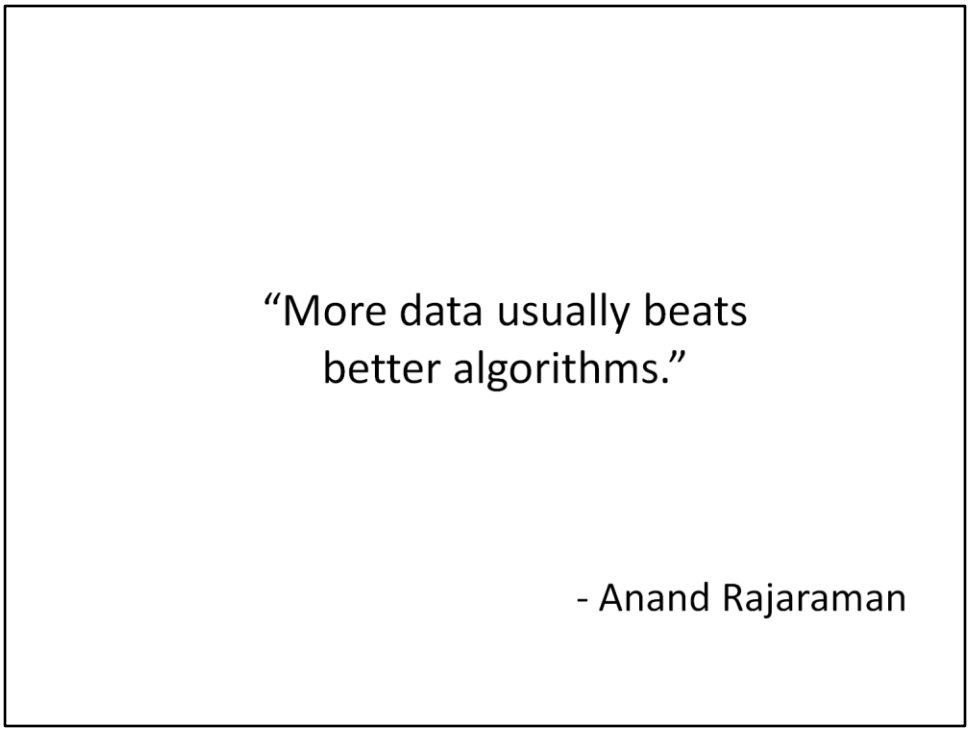
If you've taken an algorithms class you're well aware that complexity is a measure of the time (or order magnitude of time) it takes to solve a problem. While popularized by Knuth, big O was actually defined in the late 1800s by Paul Bachmann and Edmund Landau. Boiled down to its core theorem - complex algorithms take longer. This drives you to simplifying algorithms so they can be solved in less time. One of the downsides to simpler algorithms as we'll see is that you can get complex emerging behavior. Hold on to that thought.

A quotation originally attributed to Benjamin Franklin in his 1784 publication "*Advice to a Young Tradesman*" citing the choice of a man who can work for ten shillings a day and chooses to spend half of it somehow otherwise occupied isn't spending just the cost of his vacation but also the five shilling he didn't earn. That's a great way to turn everyone into workaholics. But it's an interesting proposition. If it's true then by the transitive property…

If only Franklin, Bachmann and Landau could have gotten together for drinks. The genius that these guys weren't aware of and we are now realizing is that you can solve complex problems in less time by throwing money at them. Put another way, doing lots of simple things is equivalent, and perhaps qualitatively, better than doing something complex.

What makes this interesting now is the cost of complexity. Ten years ago if you wanted 1000 cores you were looking at a $10m spend. Today faster cores are going for 1/10th the price. But the most compelling change is that you can get 1000 cores at an hourly rate. This is transformational. If you have a job to run and the algorithm is parallelizable then the cost of computing is fixed at any given point in time. What you control is the rate of outlay – how fast do you want it to run vs. how fast do you want to spend money. Before cloud computing this was never possible at such a massive scale.

So what does this imply for the field of computer science and how does it relate to data and the world at large?

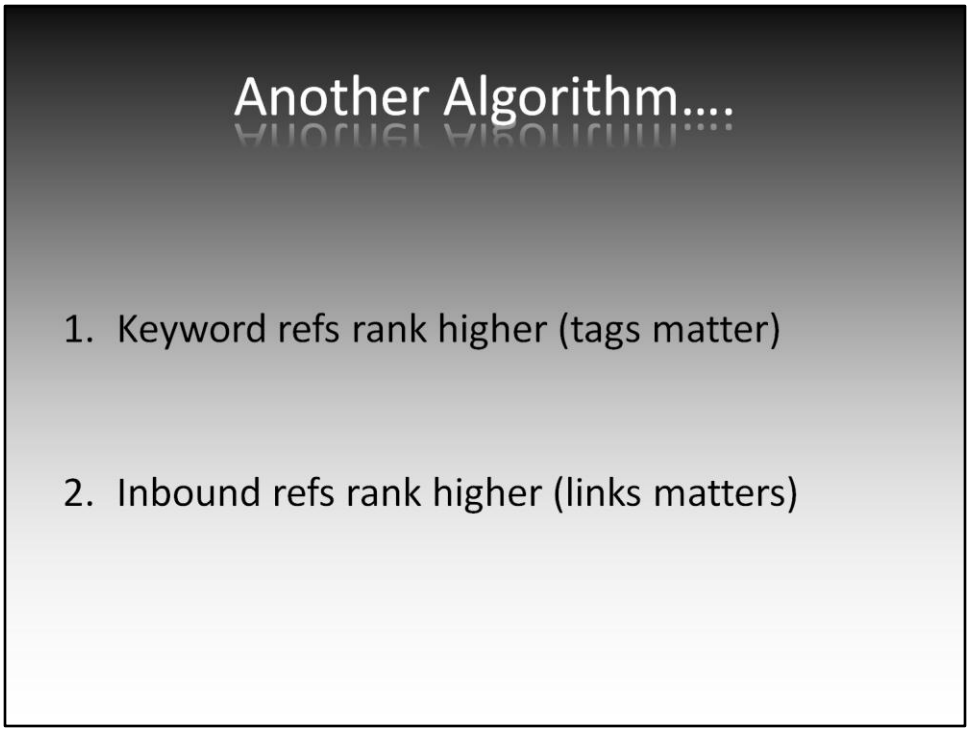


Why do we care about parallel typing monkeys? We’ll get to that in a minute. First, a slight detour regarding data. Ananda Rajaraman observed during an exercise in which his class at Standard competed for the Netflix prize that a team with a simpler algorithm that considered additional information (IMDB in this case) was able to beat a more complex algorithm that used only the Netflix data. Take a simpler algorithm and feed it a much wider data set and you’ll usually get better results.

Algorithm for Relevancy

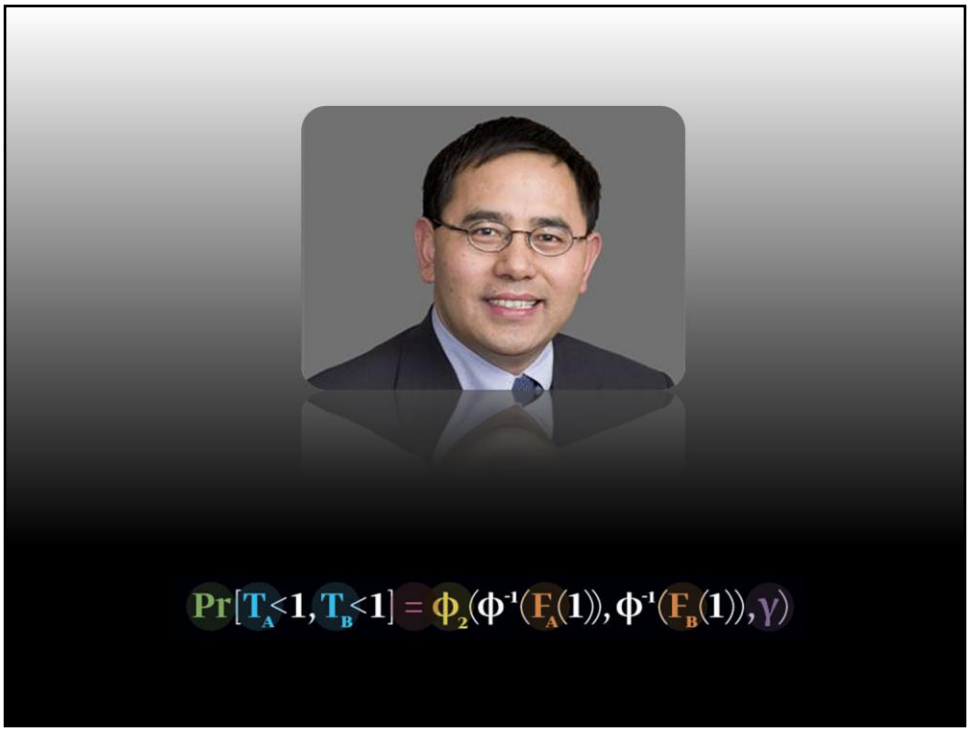
1. NLP to identify subjects within a document
2. Graph vectors of claims
3. Use cosine correlations to normalize claims
4. Extract outbound references along vectors
5. Modify rank by inbound references along relevant vectors

Let's take an example.
Sophisticated algorithm for identifying relevancy
Implemented in any one of a dozen KM systems written during the 1990s.
What are your impressions?



Here's another algorithm that's much simpler.
Would you be surprised if I told you that this algorithm beat the more sophisticated one?

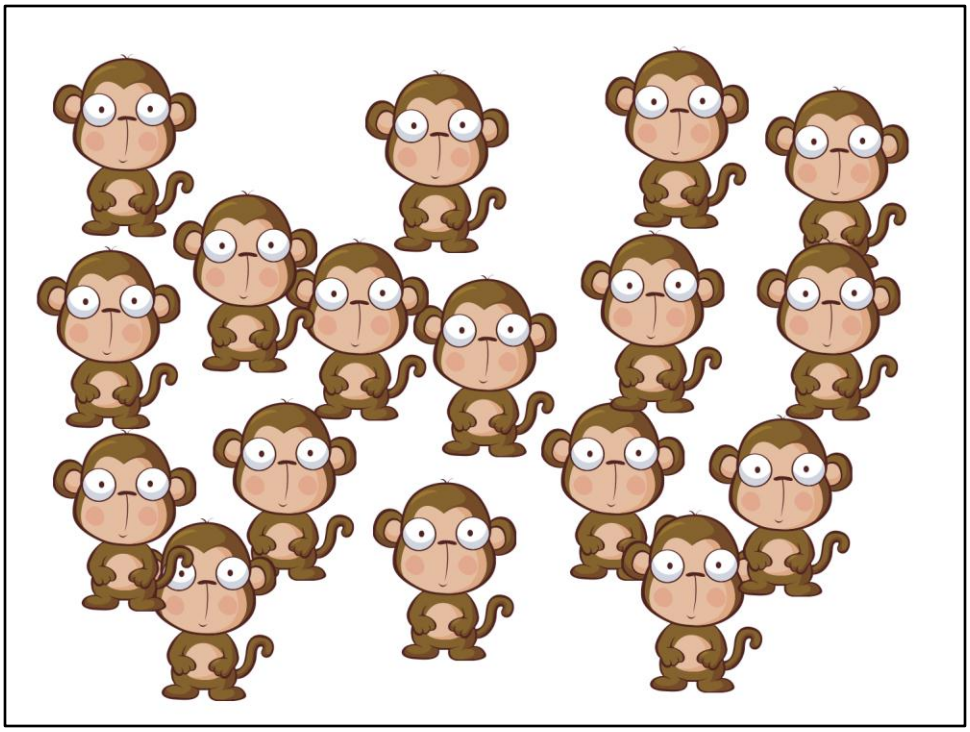
Which one do you think google uses?
They're worth $200B.
Can anyone name a company still around whose entire business is built on the first algorithm?
Autonomy is the closets worth $6.5B.



Another story about complexity and data.
In 2000, while working at JPMorgan Chase, David Li published a paper in *The Journal of Fixed Income* titled "On Default Correlation: A Copula Function Approach."
(In statistics, a copula is used to couple the behavior of two or more variables.)
Li had honorable intentions trying to simplify an incredibly complex field
Unfortunately markets ate it up (during a boom in speculative lending)
Few people tried to work out what was really going on.
This simple algorithm had emerging complexity.
Need to apply lots of data to understand the implications.
See "The Big Short" for an interesting perspective on the lack of understanding and analysis underlying the financial crisis.

We're 1/3rd of the way through
Quick summary of what we've learned
Complexity = money
Simpler means less money

You want to solve a complex problem and you have money, what do you do with it?

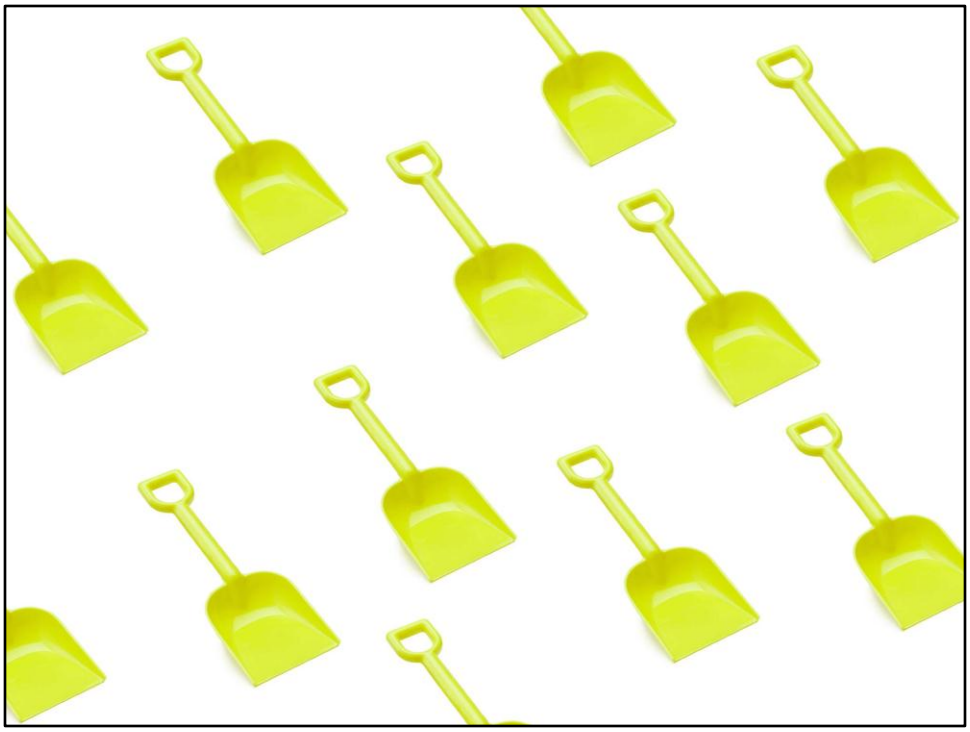


You run a bunch of simple algorithms through a lot of data.
Emile Borel realized this potential in 1913 when he wrote on statistical mechanics.
Any guesses as to the analogy he used?
Is there anything 1m monkey's can't do?

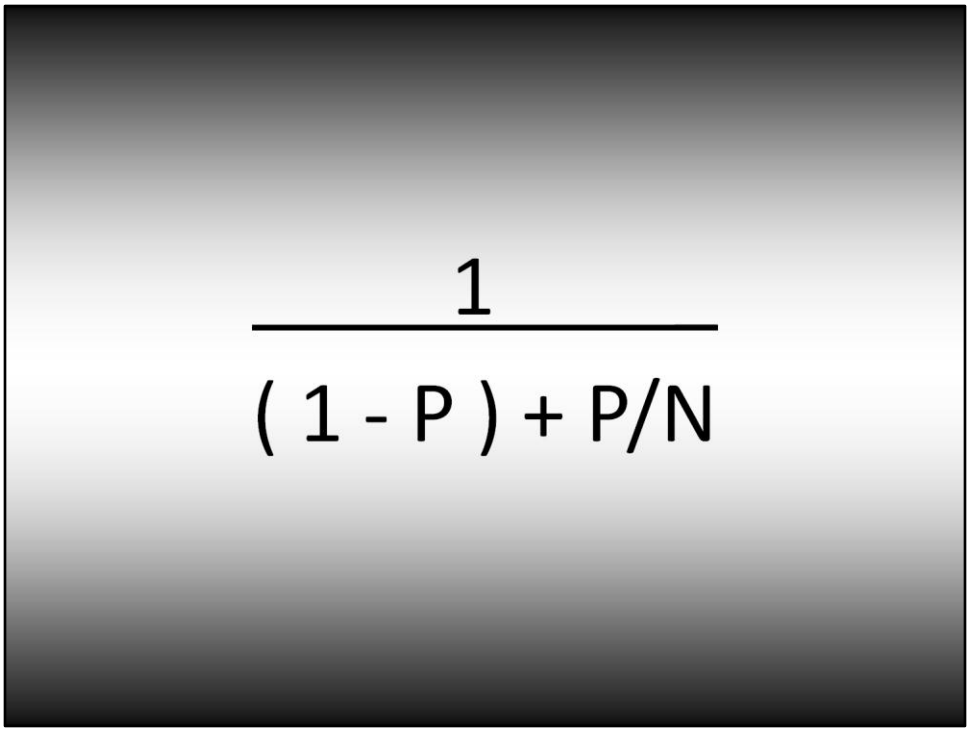
What else do you do with your money?
Buy yourself a pile of data.
Why? More data = better results.

So you've got lots of data and a bunch of monkeys, now what?

You can take your simple algorithm and start shoveling. Or…

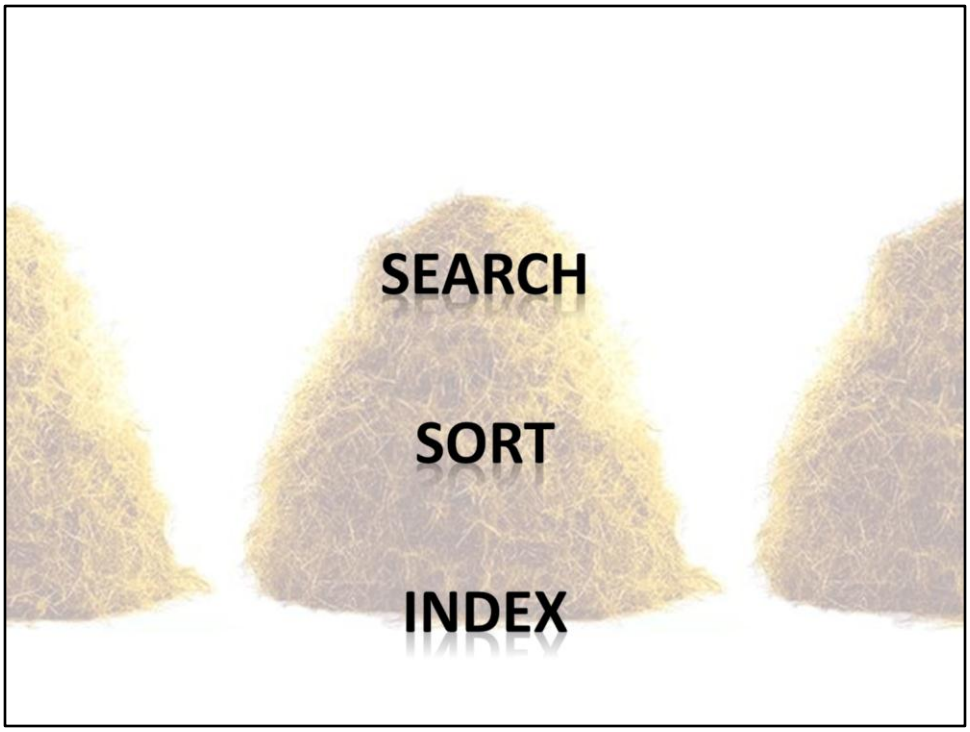


Get a lot of shovels.
Simple algorithm, lots of data and lots of processing.
Remember: Complexity = Money.
More money, simpler (parallel) algorithms, better and faster solutions.

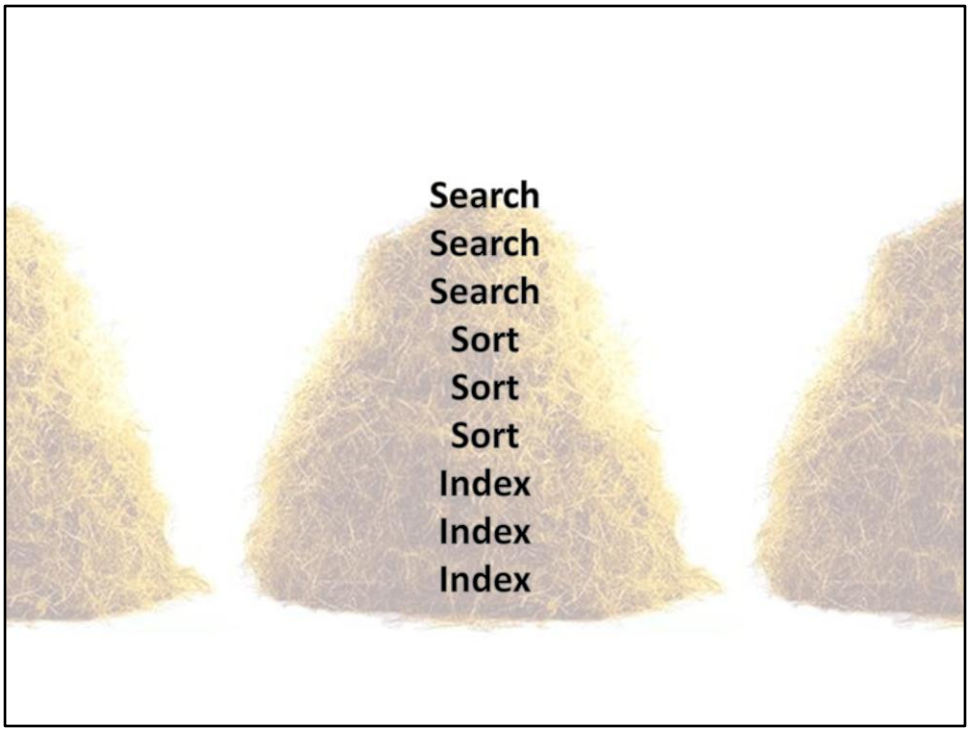


There's an exception to this of course known as Ahmdal's law. Roughly speaking you can only speed up the parallelizable part of an algorithm. You will always be limited by the speed of the serial component. Of course this means you get the most bang for your buck with embarrassingly parallel algorithms. More on that in a few minutes.

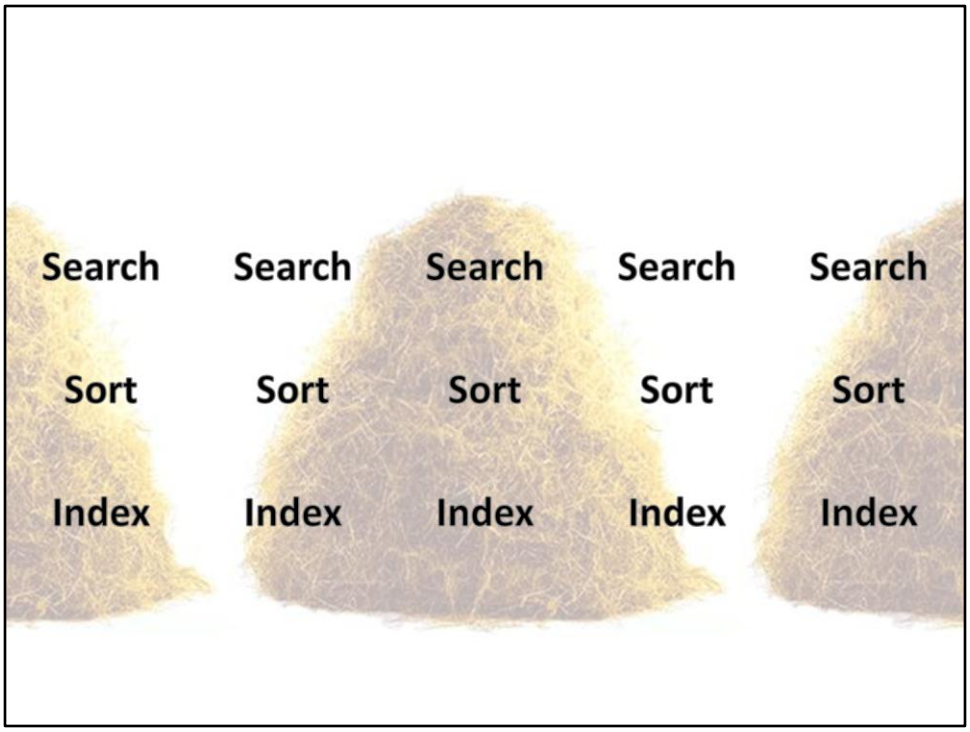
Here's a basic example of parallelizing simple algorithms.
Given a pile of data, how do you find a singe value – so called needle in a haystack.



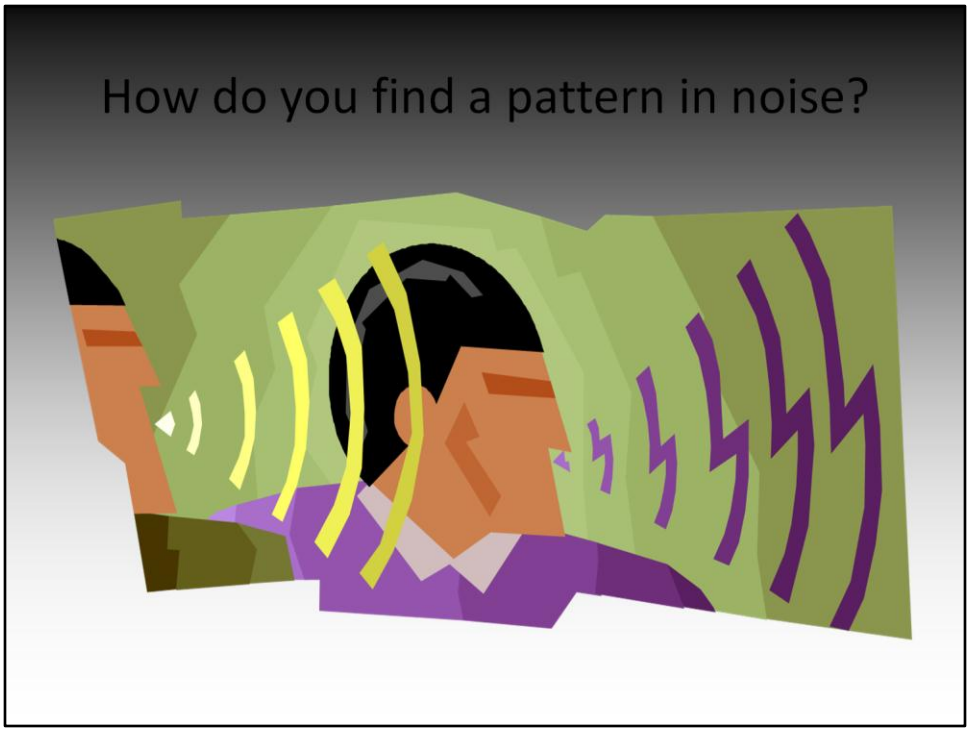
There are a few simple algorithms you can throw at it.
Brute force search, simple sort and scan or build an index like a b-tree.



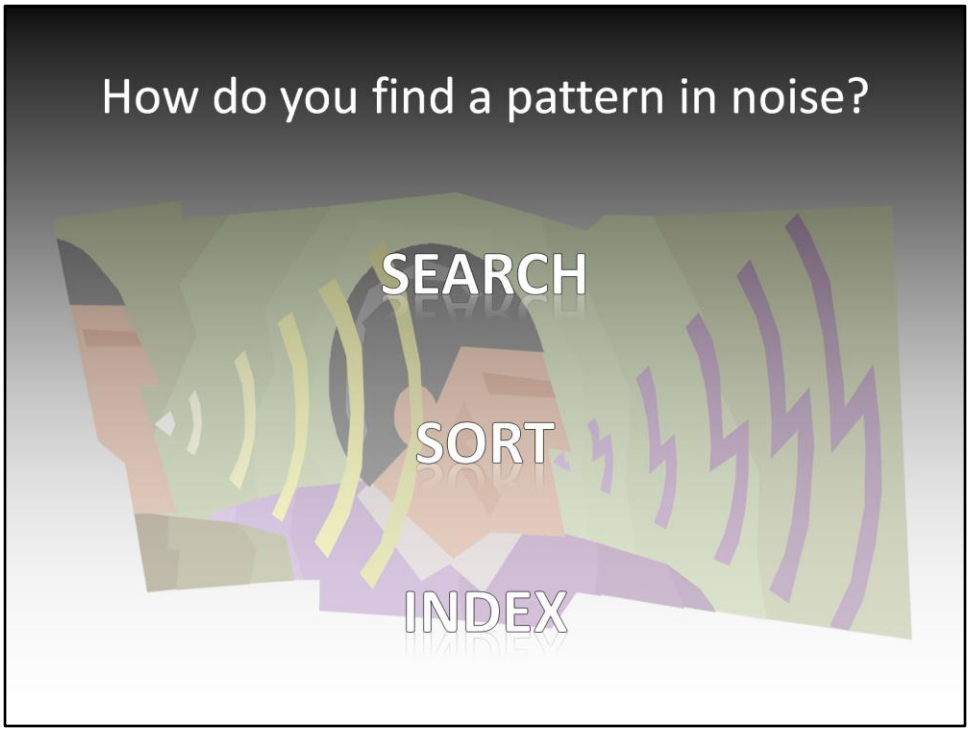
How do these scale?
Search is O(n)
Sort is O(n log n)
Building an index is same as out of place sort



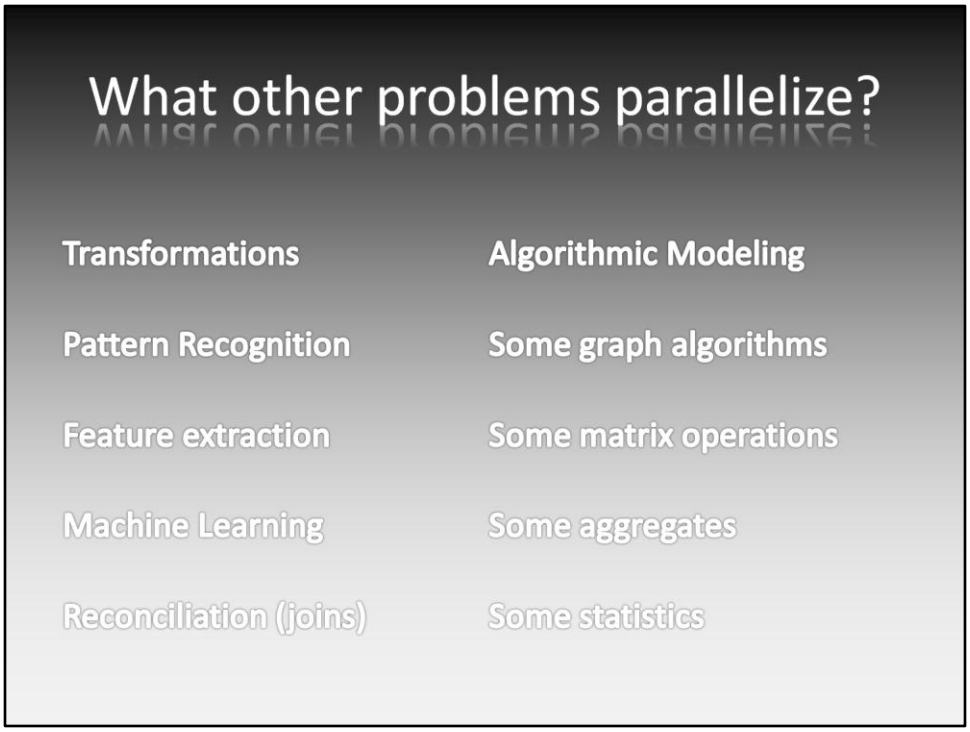
Which of these parallelize?
All of them.
How?
Partition a search is trivial
Sort may require a merge phase
Indexes may as well the merge phase is trickier



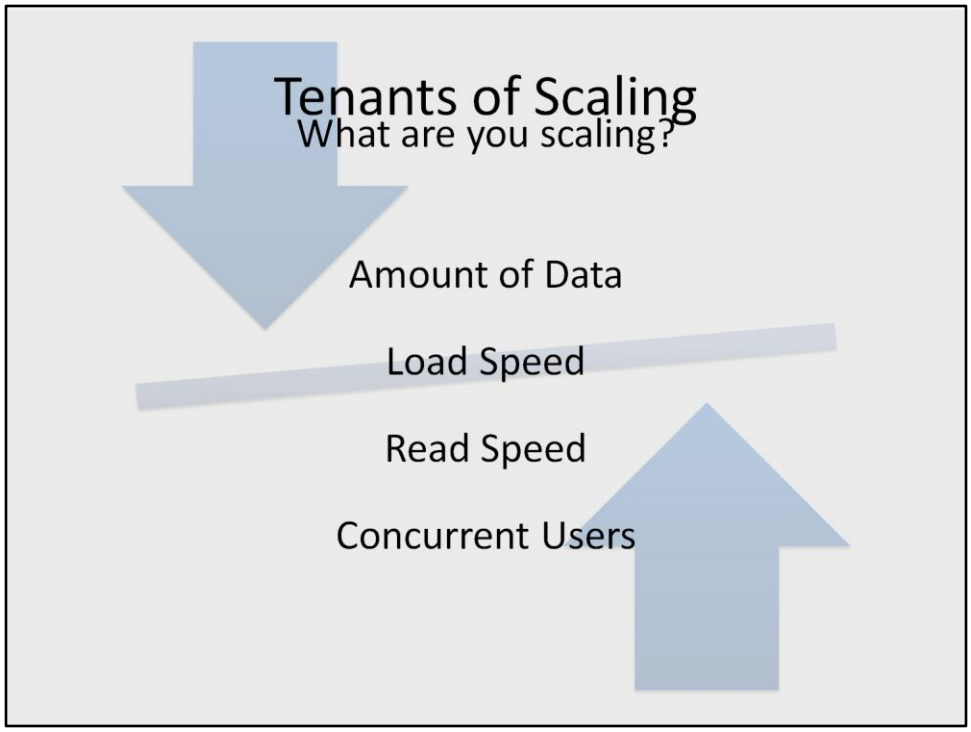
How do you find a pattern in noise?



Which approach do you take?
Why is this different than a needle in a haystack?
Does big O change?



What other problems parallelize?
Transformations (e.g. extract IP from log file and do a GeoIP lookup)
Pattern Recognition (when does a power transformer look like it's going to fail)
Feature Extraction (what are common buying signals preceding stock trade volume increase)
Machine Learning (iterate over data set to build a predictive model)
Reconciliation (map incoming credit reports to existing credit reports using name matching)
Algorithmic Modeling (test whether a model backfits historical behavior)
Some graph algorithms, matrix operations (hint, edges are hard)
Some aggregates (sum, avg, count)
Some statistics (correlations if you colocate by the correlated dimension)



Since you're going to be digging into these systems and underlying implementations I wanted to cover a few tenants of scaling.

The first question is: what are you scaling?
You can scale a system based on the amount of data you're storing.
You might need to write data quickly.
You might need to read data quickly.
You might have a lot of concurrent users doing small actions.

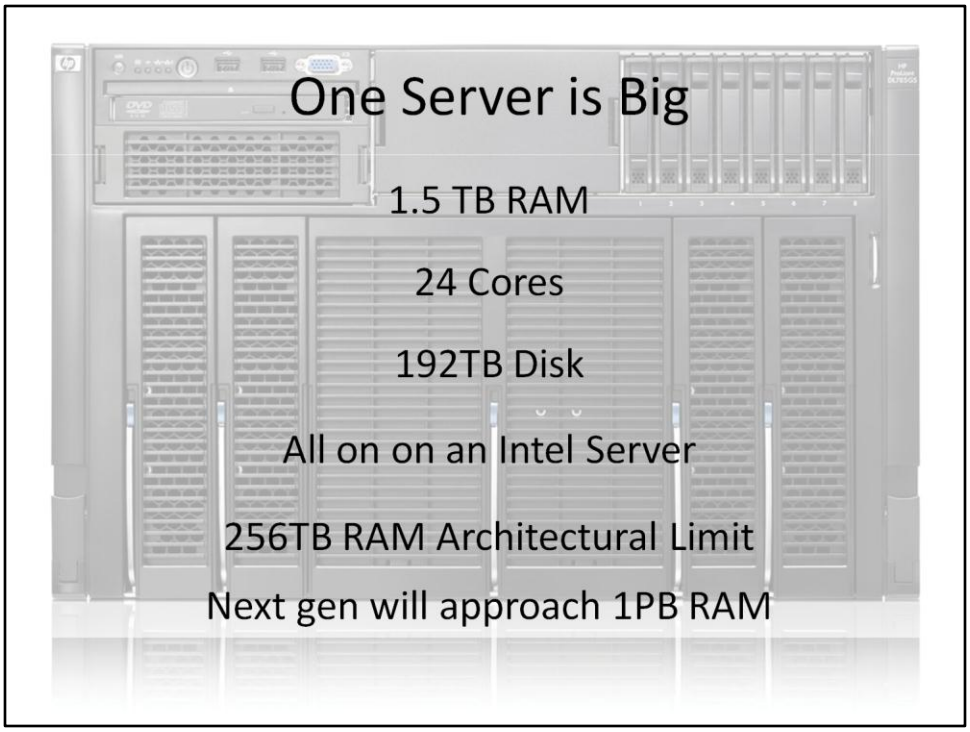
Depending on what you're trying to scale you may need to optimize differently and you may need a different type of engine.

Before you start jumping into building MPP systems, consider this:

One server is relatively easy.

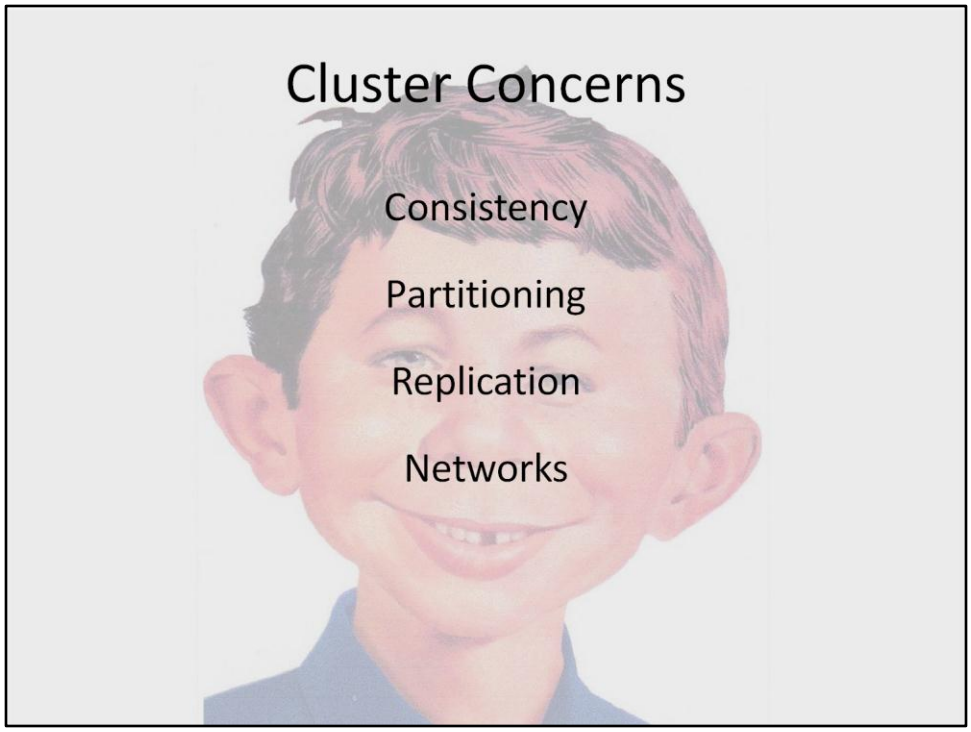
Consistency is easy.
Master/Slave replication, while it has some trickiness operationally, is conceptually easy.
Failover (assuming you test well) is also conceptually easy.

So if you're scale is never going to exceed one server, just buy a big server. Seriously.



Single servers (on industry standard hardware) are big and getting bigger.

Got 1TB of data to crunch? You can throw RAM at it. Take a look at Backtype's blog as an example. Graph analysis is compute and network intensive. If you can fit it on one box you're better off.



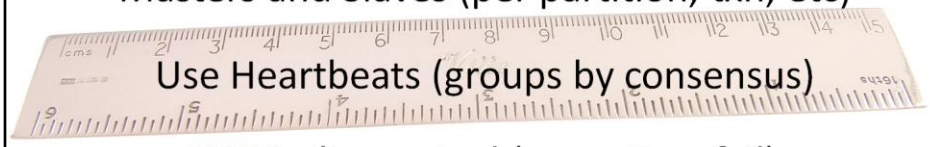
Of course, as soon as you're talking about more than one box, you need to concern yourself with a mess of other issues.

Consistency becomes hard. You need to synchronize with no shared state (remember, this *is* the datastore).
You have to partition data so some lives on some machines and some lives on others.
You also need to replicate data. With one server, if it goes down you fail over (or just fail).
The more machines you have the more susceptible you are to failure. You can replicate to a mirror cluster or you can replicate within the cluster.
You have to take networks into account. In one box, either it's online or not. In a cluster, the network is a part of the system.

Scale out 101

Segment Data (a.k.a. auto-sharding)

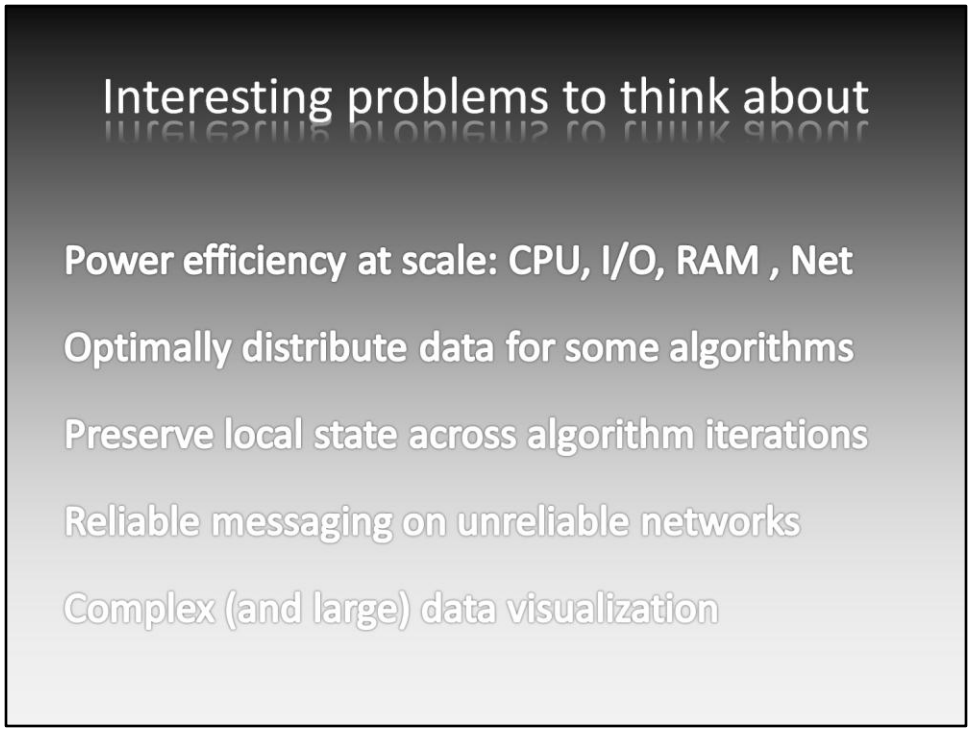
Masters and Slaves (per partition, txn, etc)



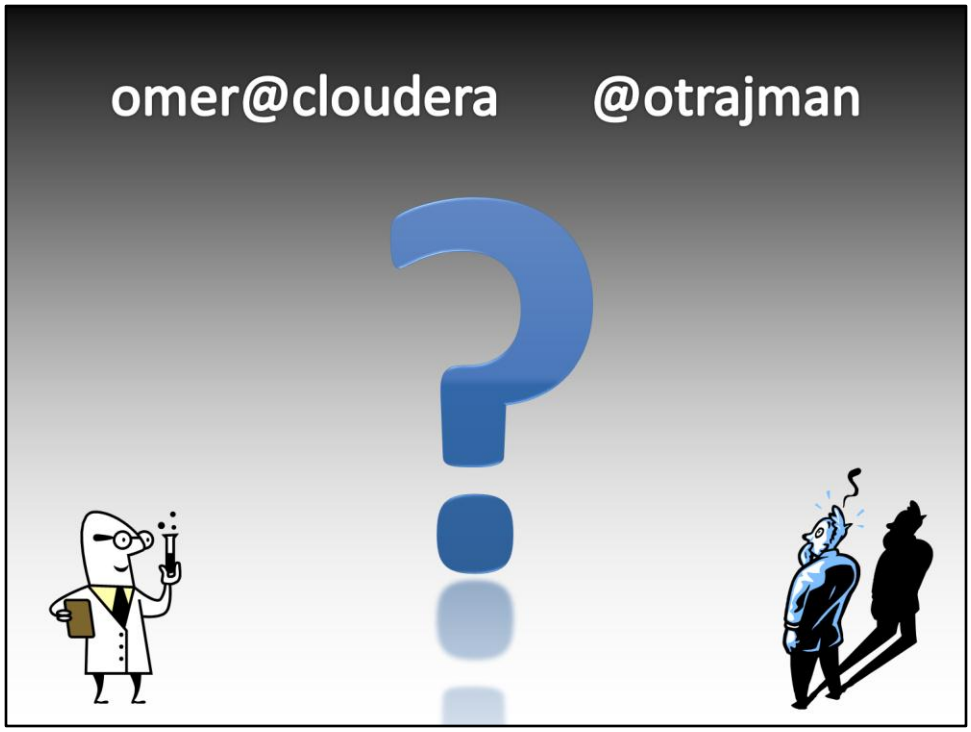
2PC isn't required (commit or fail)

Recovery and Rebalance (easier said than done)

We're running short on time so I'll just cover scale out 101.
First – segment your data in the system (built in sharding). Some goes to server A, some to server B. How flexible you are about the sharding algorithm will dictate what you can optimize and how difficult recovery and rebalancing will be.
You're going to need build in master/slave. This doesn't have to be static, but for some granularity you should have one master and some number of slaves. A few systems can use eventually consistency for data but it's very difficult to program against and has limited applications. Most apps need a consistent view of data.
You're going to use heartbeats. Find a good hearbeat system and use it. Define a cluster by consensus (this is an area where eventual consistency is a good thing). Learn Paxos.
You don't actually need 2PC. In fact for scale out systems you don't want to wait for everyone. You need a minimal number of commits to sustain your cluster and everyone else will go offline until they catch up.
Recovery and rebalancing is difficult. Lots of data and depending on your sharding algorithm can be tricky to coordinate.



Interesting problems (if you're looking for a thesis)
How do you optimize Watts/Gflops? RAM today is 3 channel – more bandwidth but higher power.
Disk is cheap but limited to 100MB/s. Switches are going to 10GigE. All of this uses more power.
Given a set of algorithms, how best to distribute data for minimal redistribution.
If your algorithm has iterations how do you preserve local state (hint, look at Pregel and BSP)
Networks are inherently unreliable (see CAP theorem). How can you abstract this and provide reliable messaging?
How do you visualize 1PB of data? Think multi-dimensionally…



Questions?