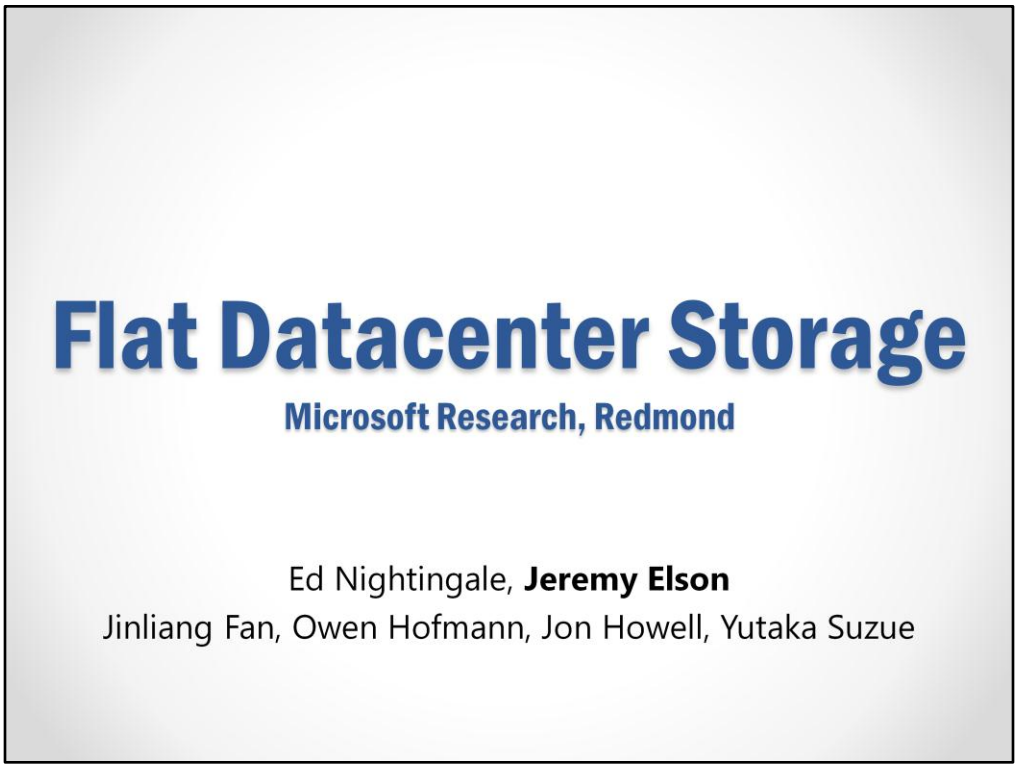
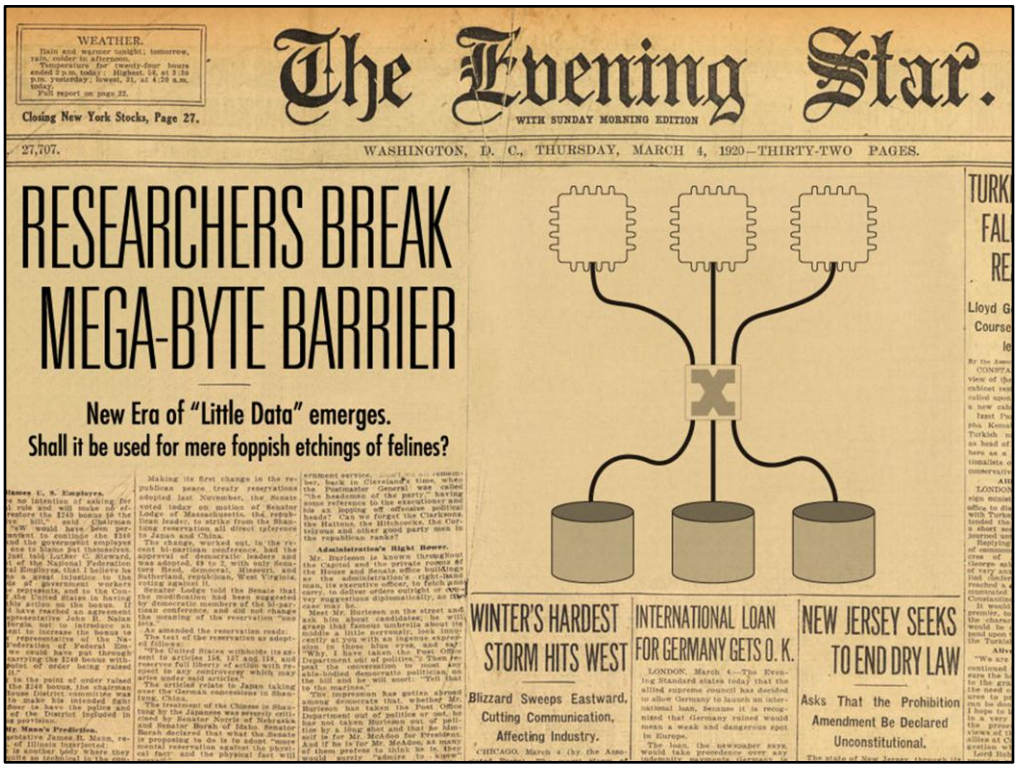


Thank you, and welcome to OSDI! My name is Jeremy Elson and I'm going to be talking about a project called flat datacenter storage, or FDS.

I want to start with a little thought experiment. This session is called "big data", but imagine for a minute we're in an alternative universe where what's really important is "little data".



WEATHER.
Rain and warmer tonight; tomorrow, rain, colder in afternoon. Temperature for twenty-four hours ended 2 p.m. today: Highest, 54, at 3:30 p.m. yesterday; lowest, 31, at 4:20 a.m. today. Full report on page 23.

Closing New York Stocks, Page 27.

The Evening Star.
WITH SUNDAY MORNING EDITION

27,707. WASHINGTON, D. C., THURSDAY, MARCH 4, 1920—THIRTY-TWO PAGES.

RESEARCHERS BREAK MEGA-BYTE BARRIER

New Era of "Little Data" emerges.
Shall it be used for mere foppish etchings of felines?

WINTER'S HARDEST STORM HITS WEST

Blizzard Sweeps Eastward, Cutting Communication, Affecting Industry.

INTERNATIONAL LOAN FOR GERMANY GETS O. K.

NEW JERSEY SEEKS TO END DRY LAW

Asks That the Prohibition Amendment Be Declared Unconstitutional.

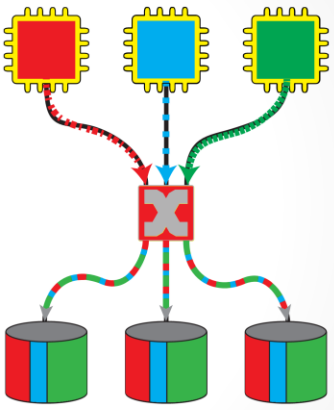
In that case, I think we all would have stopped making headlines a long time ago. Because, from a SYSTEMS perspective, I think little-data really is a solved problem. We already HAVE the perfect little-data computer, and it's right there on Page 1, above the fold:

A single machine, that has a bunch of processors, and a bunch of disks, connected by something like a RAID controller.

For I/O intensive workloads, that computer really is ideal.

Writing

- Fine-grained write striping → statistical multiplexing → high disk utilization
- Good performance and disk efficiency



When applications WRITE, the RAID controller splits those writes up pretty finely and stripes them over all the disks.

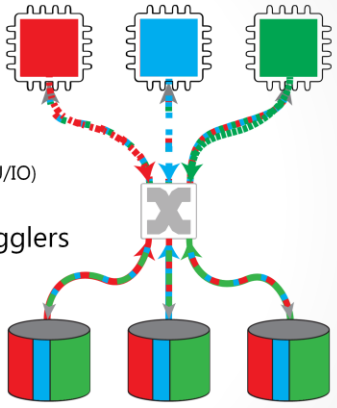
You might have a small number of writers writing a lot, or a large number of writers writing a little bit, or even both kinds writing at the same time.

But the lulls in one writer are filled in by the bursts in another, meaning we get good statistical multiplexing.

All the disks stay busy, and high utilization means we're extracting all the performance we can out of our hardware.

Reading

- High utilization (for tasks with balanced CPU/IO)
- Easy to write software
- Dynamic work allocation → no stragglers

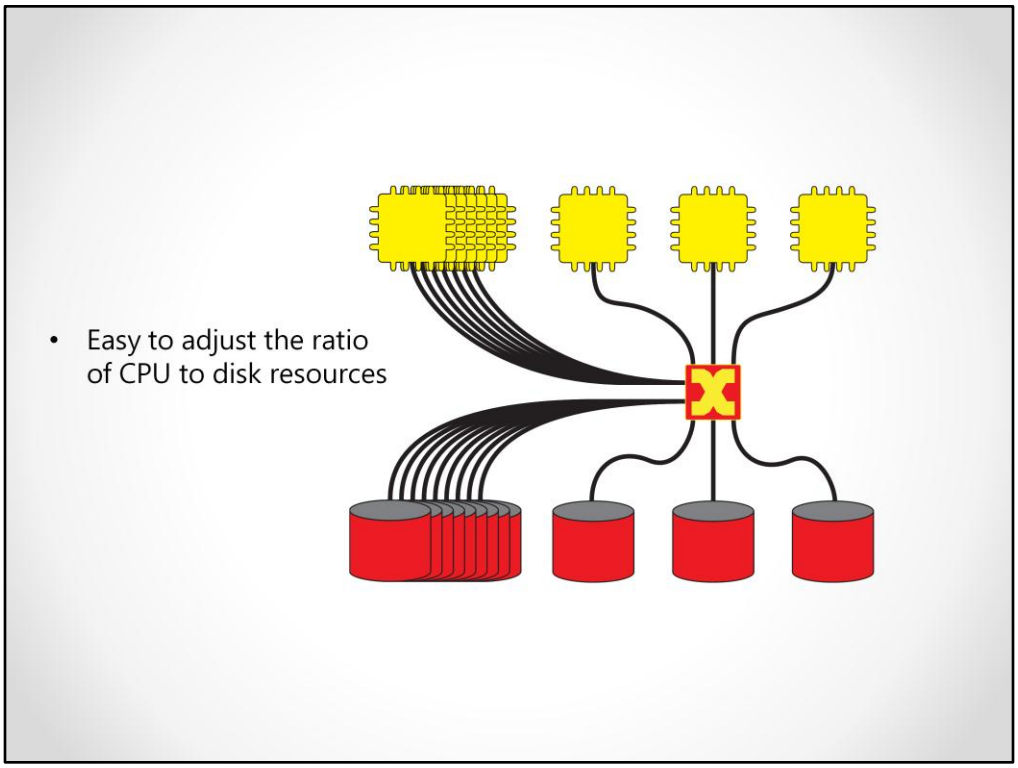


Reads can also exploit the striped writes.

(***) Again, we can get full performance out of the disks. Even if some processors consume data slowly, and others consume it quickly, all the disks stay busy, which is what we want.

(***) It's easy to write software for this computer, too. It doesn't matter how many physical disks there are; programmers can pretend there's just one big one. And files written by any process can be read by any other without caring about locality.

(***) Also, if we're trying to attack a large problem in parallel – for example, trying to parse a giant log file – the input doesn't need to be partitioned in advance. All the workers drain a global pool of work, and when it's exhausted, they all finish at about the same time. This prevents stragglers and means the job finishes sooner. We call this dynamic work allocation.

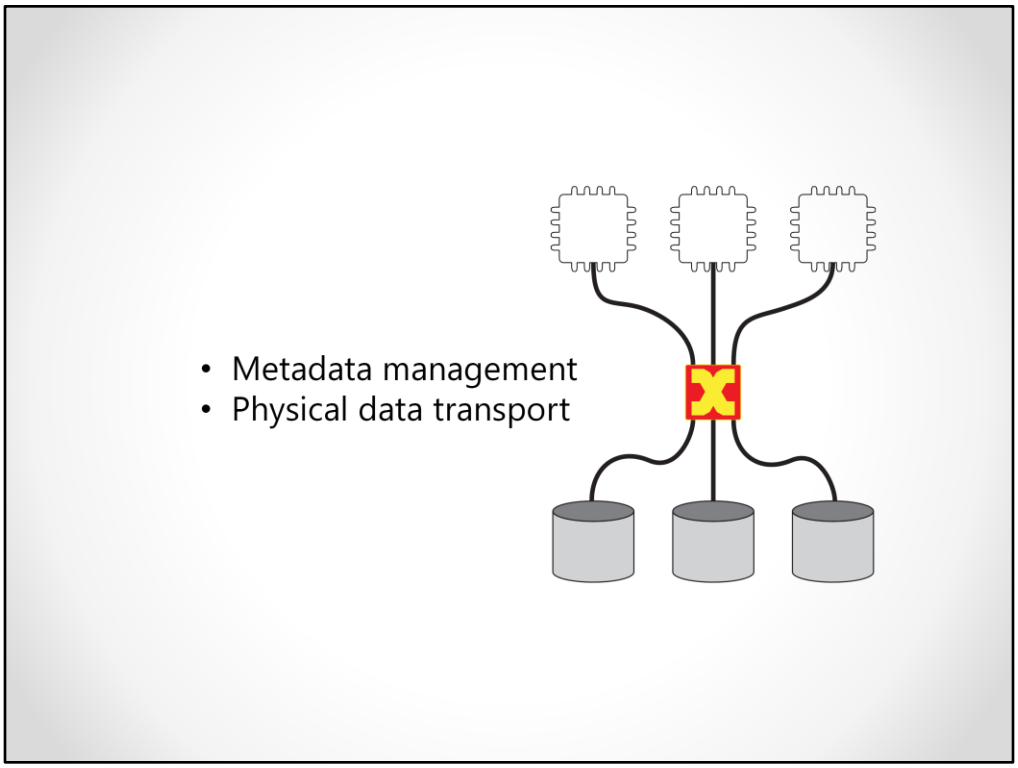


Another benefit of this computer is that it's (***) easy to adjust the ratio of processors to disks just by adding more of whichever one you need. You can fit the machine's resources to match its expected workload.

OK, so, what's the problem?

Back in the real world of *big* data, the problem is that this machine doesn't scale. We can add a few *dozen* processors and disks, but not *thousands*.

Why not?

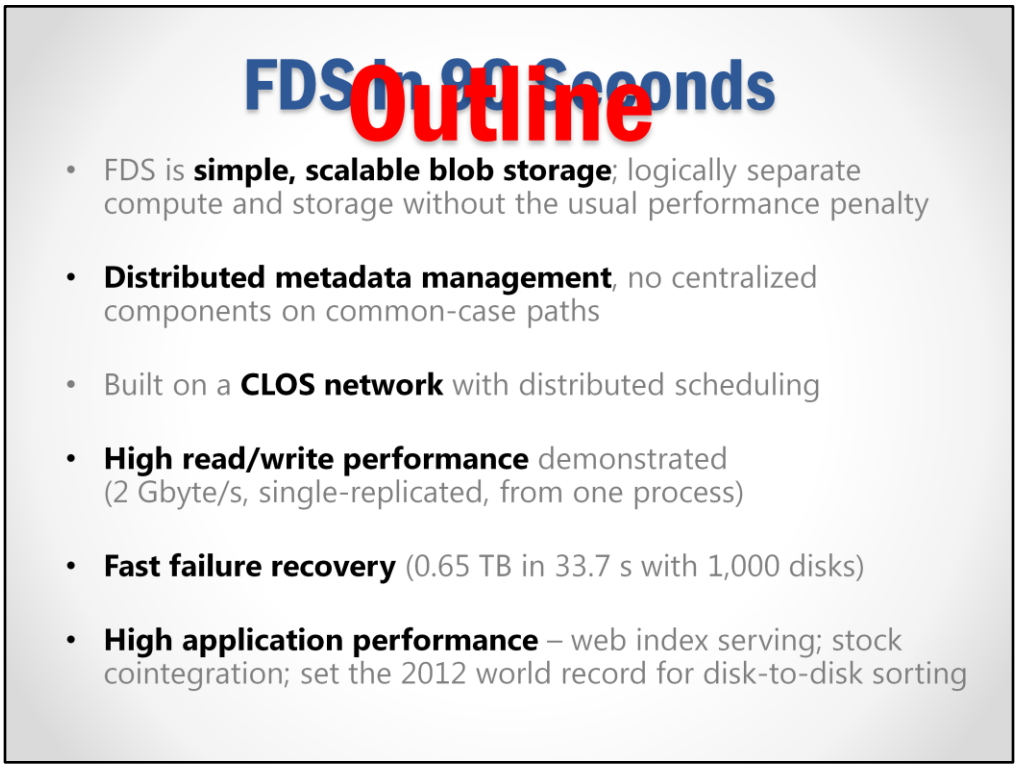


Let's take a look at this thing in the middle here. That's really where the magic happens. Roughly, it's doing two things.

(***) The first is metadata management – when a process writes, that thing decides how the write should be striped, and keeps enough state around so that reads can find the data later.

(***) Second, it's responsible for physically routing data from disks to processors – actually transporting the bits.

In FDS, we've built a blob storage system that fully distributes both of these tasks. This means we can build a cluster that has the essential properties of the ideal machine I described, but potentially can scale to the size of a datacenter. And in the next twenty minutes, I'll describe how.



Here's the 90 second overview.

(***) FDS is a simple, scalable blob store. Compute and storage are logically separate, and there's no affinity, meaning any processor can access all data in the system uniformly. That's why we call it "**flat**". We've combined that conceptual simplicity with the very high I/O performance that you've come to expect only from systems that *couple* storage and computation together, like MapReduce, Dryad and Hadoop.

(***) FDS has a novel way of distributing metadata. In fact, the common case read and write paths go through no centralized components at all.

(***) We get the bandwidth we need from full-bisection-bandwidth CLOS networks, using novel techniques to schedule traffic.

(***) With FDS, we've demonstrated very high read and write performance – in a single-replicated cluster, a single process in tight read or write loop can achieve more than 2 gigabytes per second all the way to the remote disk platters. To give you a sense of scale, that means we're writing to REMOTE disks faster than many systems can write LOCALLY, to a RAID array.

(***) Disks can also talk TO EACH OTHER at high speed, meaning FDS can recover from failed disks very quickly. For example, in one test with a 1,000-disk cluster, we killed a machine with 7 disks holding a total of about two-thirds of a terabyte; FDS brought the lost

data back to full replication in 34 seconds.

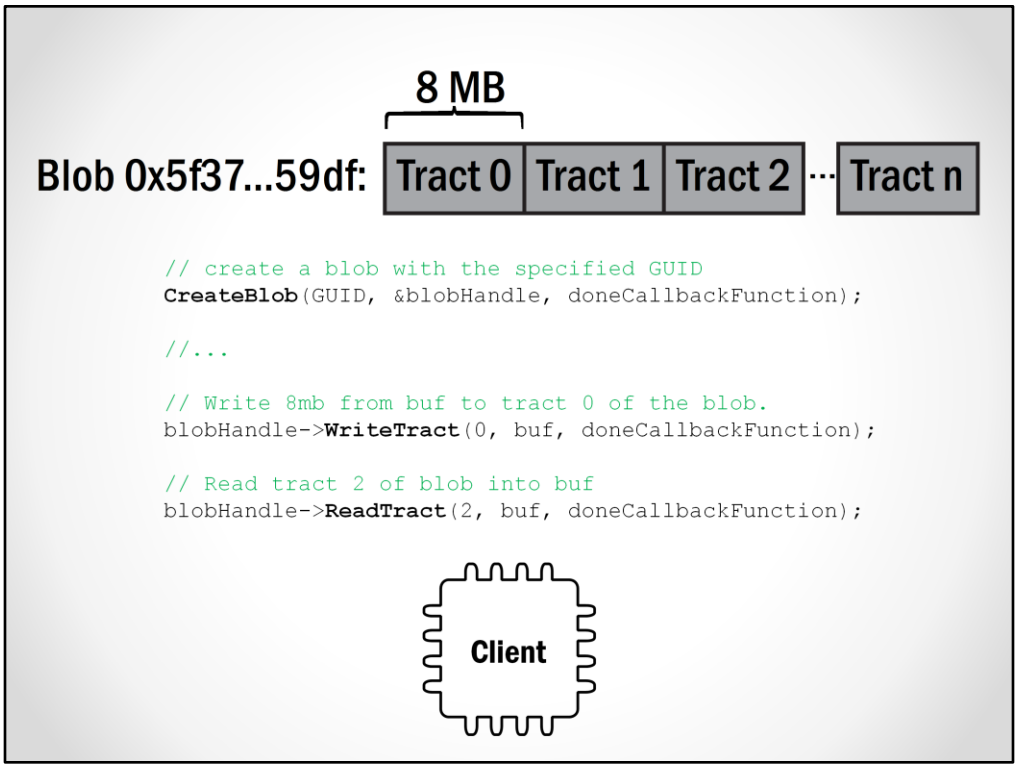
(***) Finally, we've shown that FDS can make APPLICATIONS very fast. We describe several applications from diverse domains in the paper, but in this talk I'll just be talking about one: we wrote a straightforward sort application on top of FDS which beat the 2012 world record for disk-to-disk sorting. Our general purpose remote blob store beat previous implementations that exploited local disks.

(***) This summary is also going to serve as an outline of my talk, which I will now give again, but …. Slower.

Outline

- FDS is **simple, scalable blob storage**; logically separate compute and storage without the usual performance penalty
- **Distributed metadata management**, no centralized components on common-case paths
- Built on a **CLOS network** with distributed scheduling
- **High read/write performance** demonstrated (2 Gbyte/s, single-replicated, from one process)
- **Fast failure recovery** (0.6 TB in 33.7 s with 1,000 disks)
- **High application performance** – set the 2012 world record for disk-to-disk sorting

So let's get started, and talk about the architecture in general.



First, let's talk about how the system looks to applications.

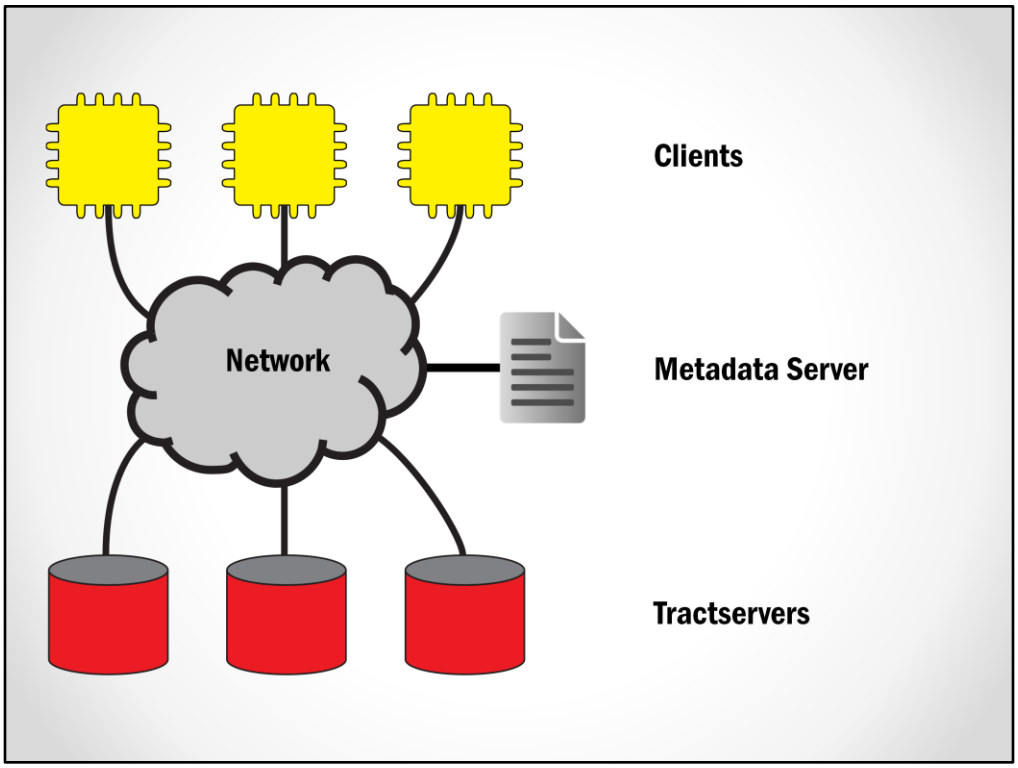
In FDS, all blobs are identified with a simple GUID.

Each blob contains 0 or more allocation units we call TRACTS, which are numbered sequentially, starting from 0.

All tracts in a system are the same size. In most of our clusters, a tract is 8 megabytes; I'll tell you a little later why we picked that number. A tract is the basic unit of reading and writing in FDS.

(***) The interface is pretty simple. It has only about a dozen calls, such as CreateBlob, ReadTract and WriteTract.
It's designed to be asynchronous, meaning that the functions don't block, but instead call a callback when they're done. A typical high-throughput FDS application will issue a few dozen reads or writes at a time, and issue more as the earlier ones complete.

We call applications using the FDS API "FDS **clients**."



In addition to clients, there are two other types of actors in FDS.

The first is what we call a **tractserver**. A tractserver is a simple piece of software that sits between a raw disk and the network, accepting commands from the network such as “read tract” and “write tract”.

There’s also a special node called the metadata server which coordinates the cluster and helps clients rendezvous with tractservers.

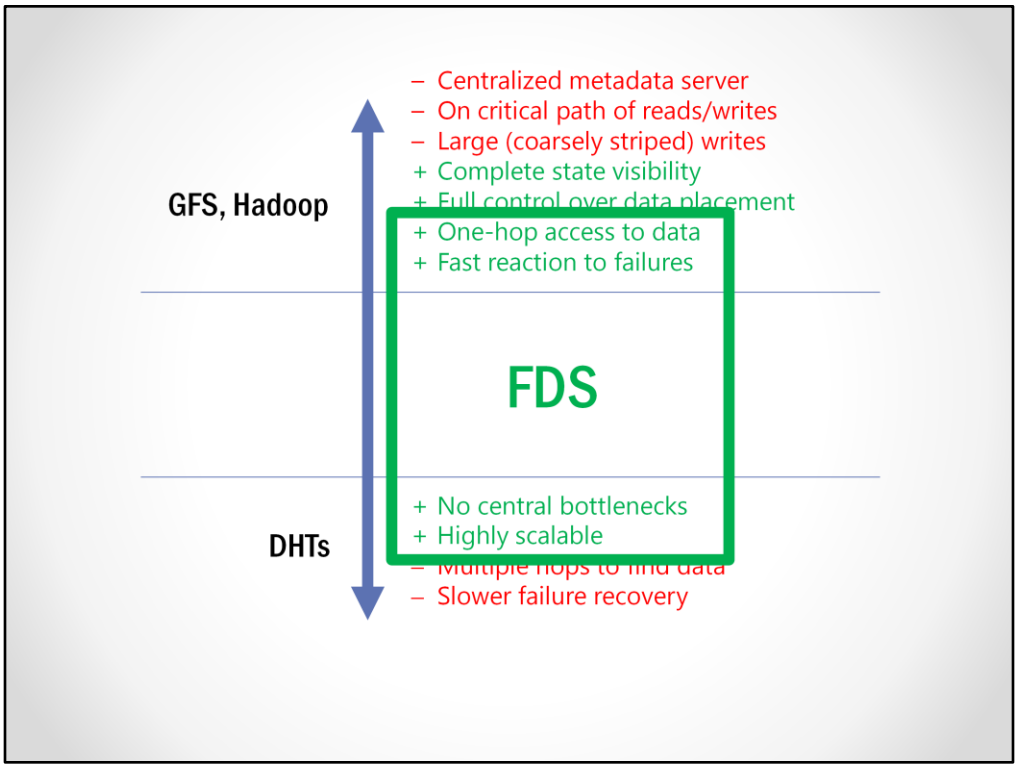
As we saw on the previous slide, the existence of tractservers and the metadata server is invisible to programmers. The API just talks about blobs and tract numbers. Underneath, our library contacts the metadata server as necessary and sends read and write messages over the network to tractservers.

An important question is, “how does the client know which tractserver should be used to read or write a tract?”

Outline

- FDS is **simple, scalable blob storage**; logically separate compute and storage without the usual performance penalty
- **Distributed metadata management**, no centralized components on common-case paths
- Built on a **CLOS network** with distributed scheduling
- **High read/write performance** demonstrated (2 Gbyte/s, single-replicated, from one process)
- **Fast failure recovery** (0.6 TB in 33.7 s with 1,000 disks)
- **High application performance** – set the 2012 world record for disk-to-disk sorting

This brings us to the next part of my talk – metadata management.

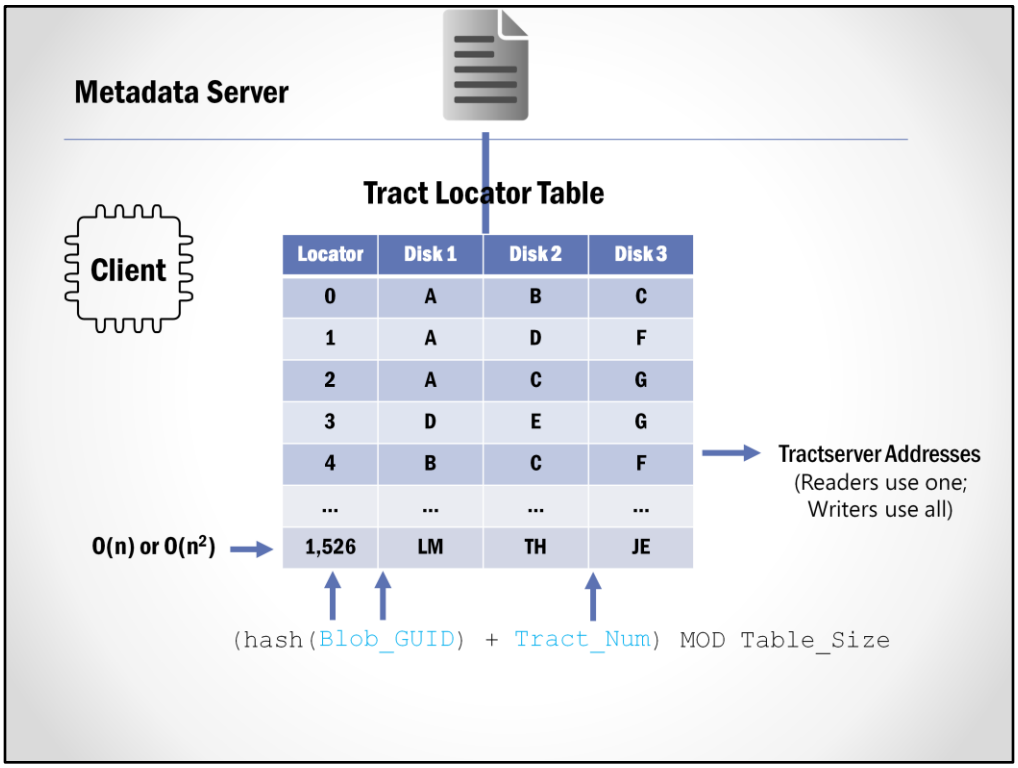


To understand how FDS handles metadata, it's useful to consider the spectrum of solutions in other systems.

On one extreme, we have systems like GFS and Hadoop that manage metadata centrally. On basically every read or write, clients consult a metadata server that has canonical information about the placement of all data in the system. This gives you really good visibility and control.
But it's also a centralized bottleneck that's exerted pressure on these systems to increase the size of writes – for example, GFS uses 64 megabyte extents, almost an order of magnitude larger than FDS. This makes it harder to do fine-grained load balancing like our ideal little-data computer does.

On the other end, we have distributed hash tables. They're fully decentralized, meaning there's no bottleneck, but all reads and writes typically require multiple trips over the network before they find data.
In addition, failure recovery is relatively slow because recovery is a localized operation among nearby neighbors in the ring.

In FDS we tried to find a spot in between that gives us some of the best properties of both extremes:
One-hop access to data and fast failure recovery without any centralized bottlenecks in common-case paths.



FDS does have a centralized metadata server, but its role is very limited. When a client first starts, the metadata server sends some state to the client – for now, think of it as an oracle.

When a client wants to read or write a tract, the underlying FDS library has two pieces of information:
The blob's GUID and the tract number.

The client library feeds those into the oracle and gets out the address of the tractservers responsible for replicas of that tract. In a system with more than one replica, reads go to ONE replica at random, and writes go to all of them.

The oracle's mapping of tracts to tractservers needs two important properties.

First, it needs to be consistent: a client READING a tract needs to get the same answer as the writer got when it wrote that tract.

Second, it has to be pseudo-random. As I mentioned earlier, clients have lots of tract reads and writes outstanding simultaneously.

The oracle needs to ensure that all of those operations are being serviced by different tractservers. We don't want all the requests going to just 1 disk if we have 10 of them.

Once a client has this oracle, reads and writes ALL happen without contacting the metadata

server again. Since reads and writes don't generate metadata server traffic, we can afford to do a LARGE number of SMALL reads and writes that all go to different spindles, even in very large-scale systems, giving us really good statistical multiplexing of the disks – just like the little-data computer.

And we really have the flexibility to make writes as small as we need to. For THROUGHPUT-sensitive applications, we use 8 megabyte tracts because 8 megs is large enough to amortize seeks, meaning reading and writing randomly goes almost as fast as sequentially. But, we've also done experiments with seek-bound workloads, where we pushed the tract size all the way down to 64K. That's very hard with a centralized metadata server but no problem with this oracle.

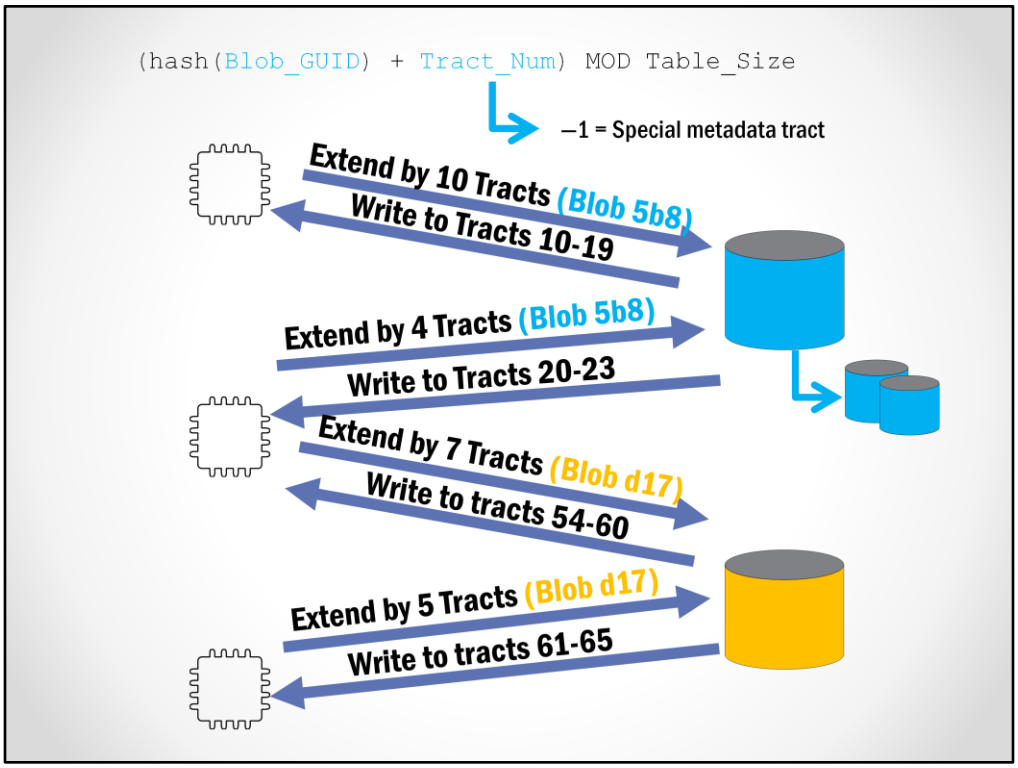
By now you're probably wondering what this oracle is.

(***) What it is is a table of all the disks in the system collected centrally by the metadata server. We call this table the **tract locator table**. The table has as many columns as there are replicas; this example shows a triple-replicated system. In SINGLE-replicated systems, the number of rows in this table grows linearly with the number of disks in the system. (***) In MULTIPLY-replicated systems, it grows as n-squared; we'll see why a little later.

Okay, so, how does a client use this table? It takes the blob GUID and tract number and runs them through a deterministic function that yields a row index. As long as readers and writers are using consistent versions of the table, the mappings they get will also be consistent. In the paper, we describe the details of how we do consistent table versioning.

(((We hash the blob's GUID so that independent clients start at random places in the table, even if the GUIDs //themselves are not randomly distributed.)))

Something very important about this table is that it only contains disks, not tracts. In other words, reads and writes don't change the table. So, to a first approximation, clients can retrieve it from the metadata server once, then never contact the metadata server again. The only time the table changes is when a disk fails or is added.



There's another clever thing we can do with the tract locator table, and that's distribute per-blob metadata, such as each blob's length and permission bits. We store this in tract "-1". Clients find the metadata tract the same way that they find regular data, just by plugging -1 into the tract locator formula we saw previously. This means that the metadata is spread pseudo-randomly across all tractservers in the system, just like the regular data.

Tractservers have support for consistent metadata updates. For example, let's say several writers are trying to append to the same blob. In FDS, each executes an FDS function called "extend blob." This is a request for a range of tract numbers that can be written without conflict. The tractserver serializes the requests and returns a unique range to each client. This is how FDS supports atomic append.

In systems with more than one replica, the requests go to the tractserver in the FIRST column of the table. THAT server does a two-phase commit to the others before returning a result to the client.

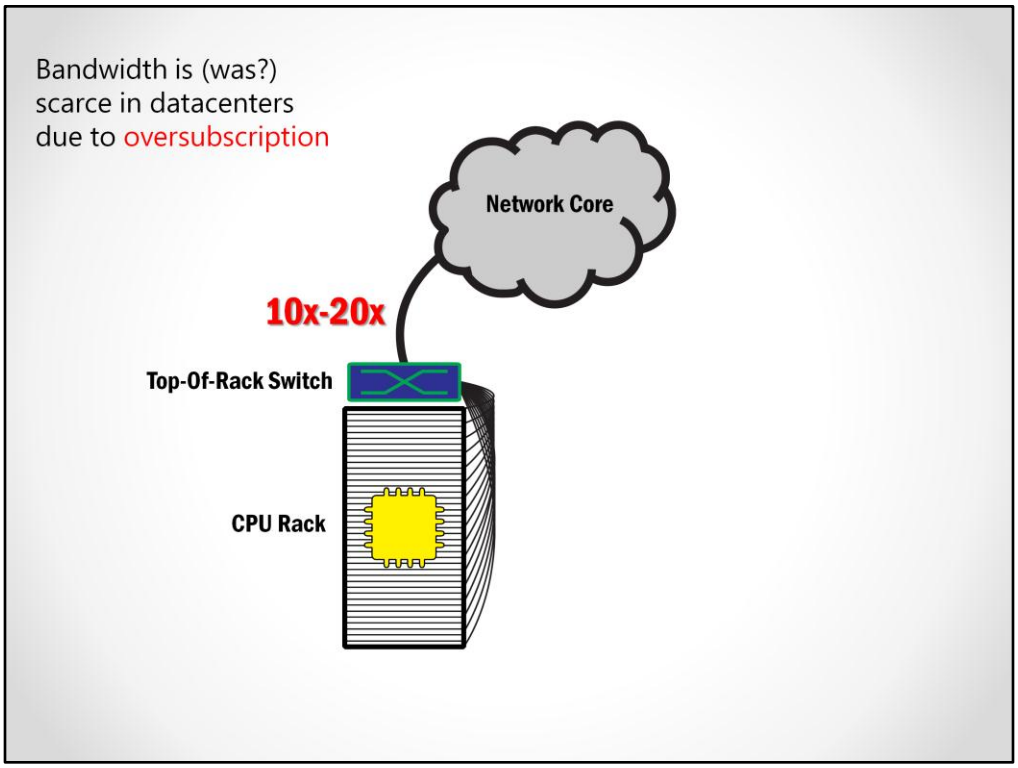
Because we're using the tract locator table to determine which tractserver owns each blob's metadata, different blobs will most likely have their metadata operations served by DIFFERENT tractservers. The metadata traffic is spread across every server in the system. But requests that need to be serialized because they refer to the same blob will always end up at the same tractserver, which maintains correctness.

Outline

- FDS is **simple, scalable blob storage**; logically separate compute and storage without the usual performance penalty
- **Distributed metadata management**, no centralized components on common-case paths
- Built on a **CLOS network** with distributed scheduling
- **High read/write performance** demonstrated (2 Gbyte/s, single-replicated, from one process)
- **Fast failure recovery** (0.6 TB in 33.7 s with 1,000 disks)
- **High application performance** – set the 2012 world record for disk-to-disk sorting

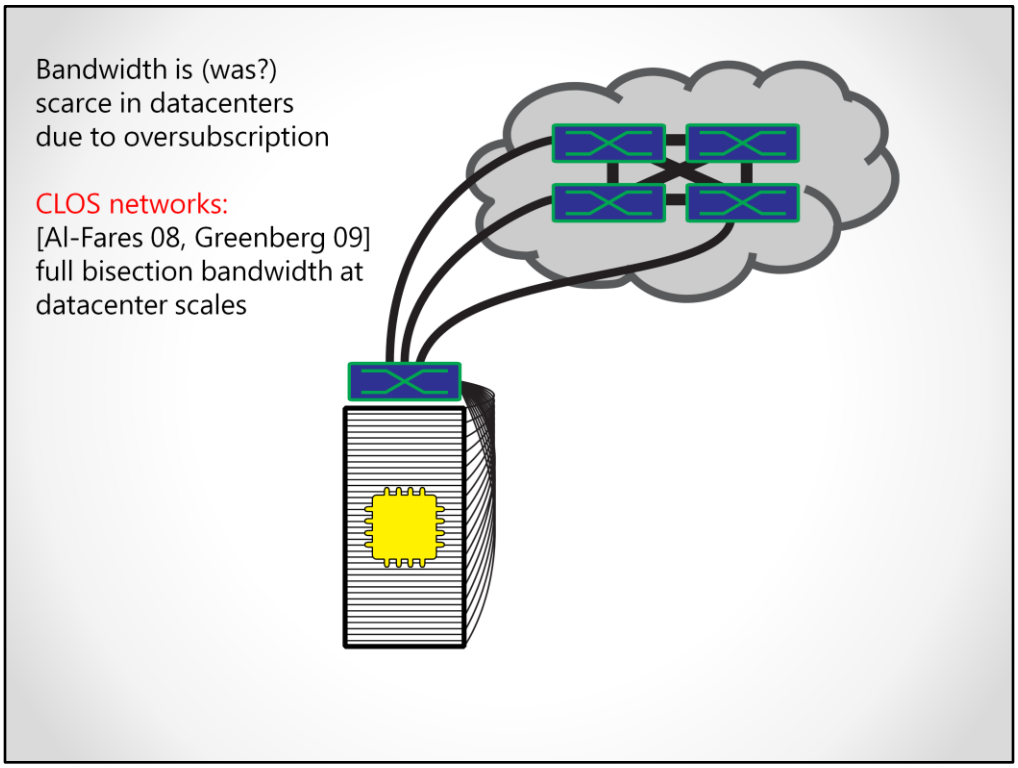
Okay, now that we've talked about how FDS handles metadata, let's talk about networking.

Up until now, I've assumed that there was an uncongested path from tractservers to clients. Let me now describe how we build such a network.



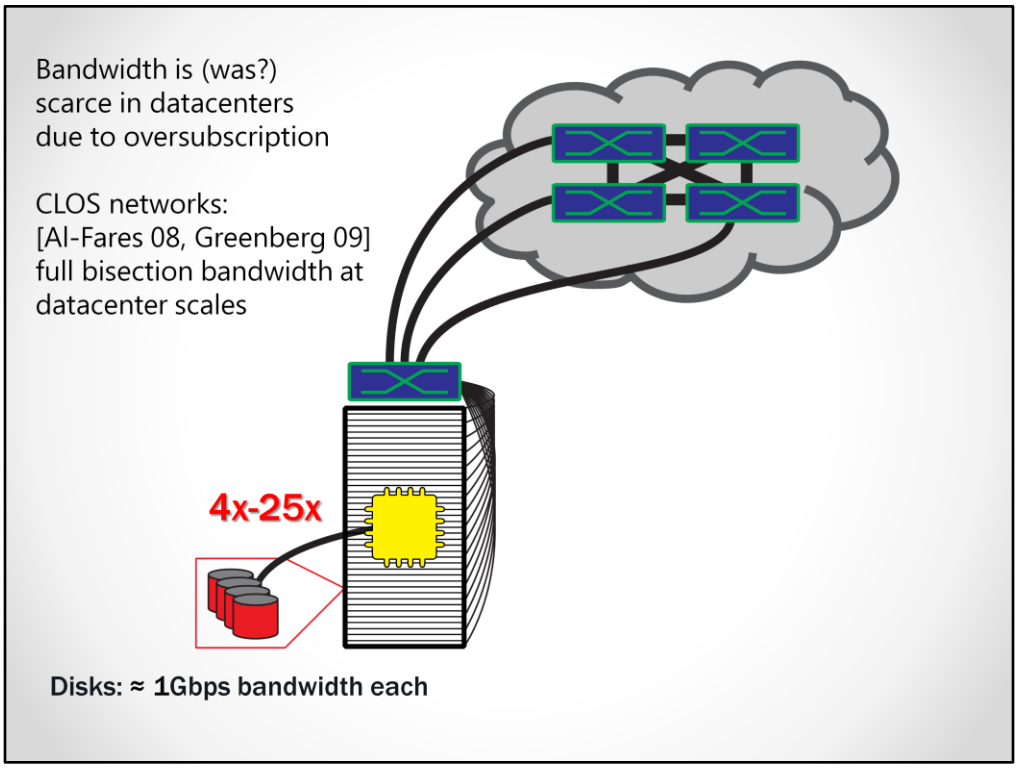
First, a little background.

Until recently, the standard way to build a datacenter was with significant oversubscription: a top-of-rack switch might have 40 gigabits of bandwidth down to servers in the rack, but only 2 or 4 gigabits going up to the network core. In other words, the link to the core was oversubscribed a factor of 10 or 20. This, of course, was done to save money.



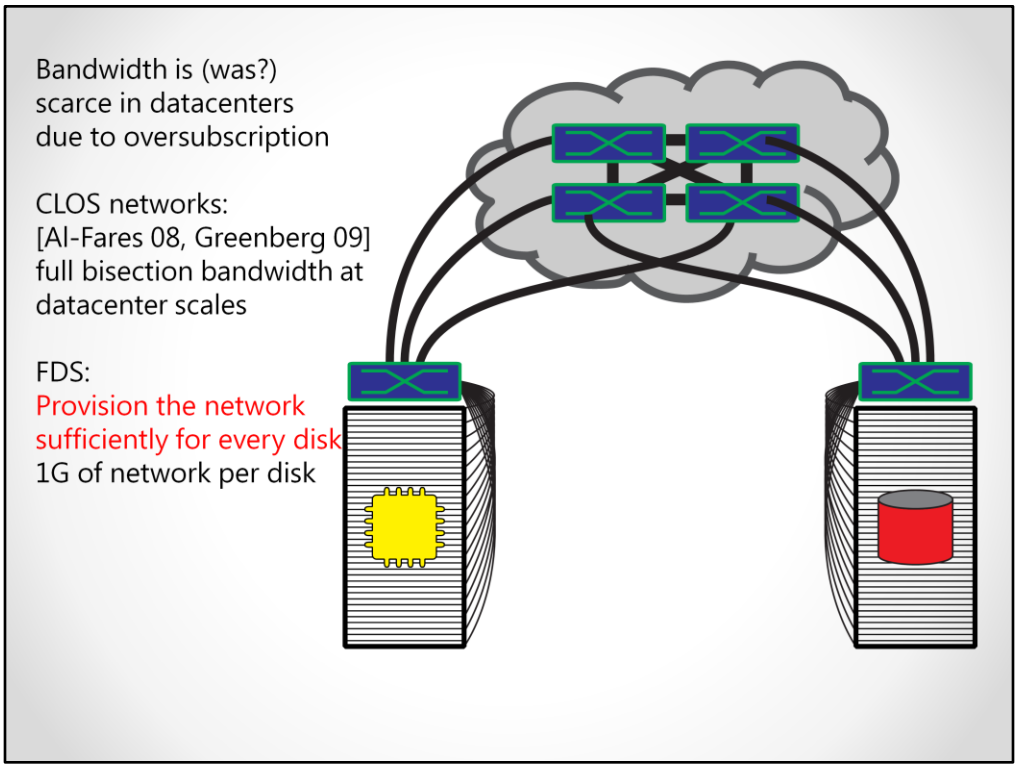
But, in the last few years, there's been an explosion of research in the networking community in what are called CLOS networks. CLOS networks more-or-less do for networks what RAID did for disks: by connecting up a large number of low-cost, commodity switches and doing some clever routing, it's now economical, for the first time, to build full-bisection-bandwidth networks at the scale of a datacenter.

In FDS, we take the idea a step further.



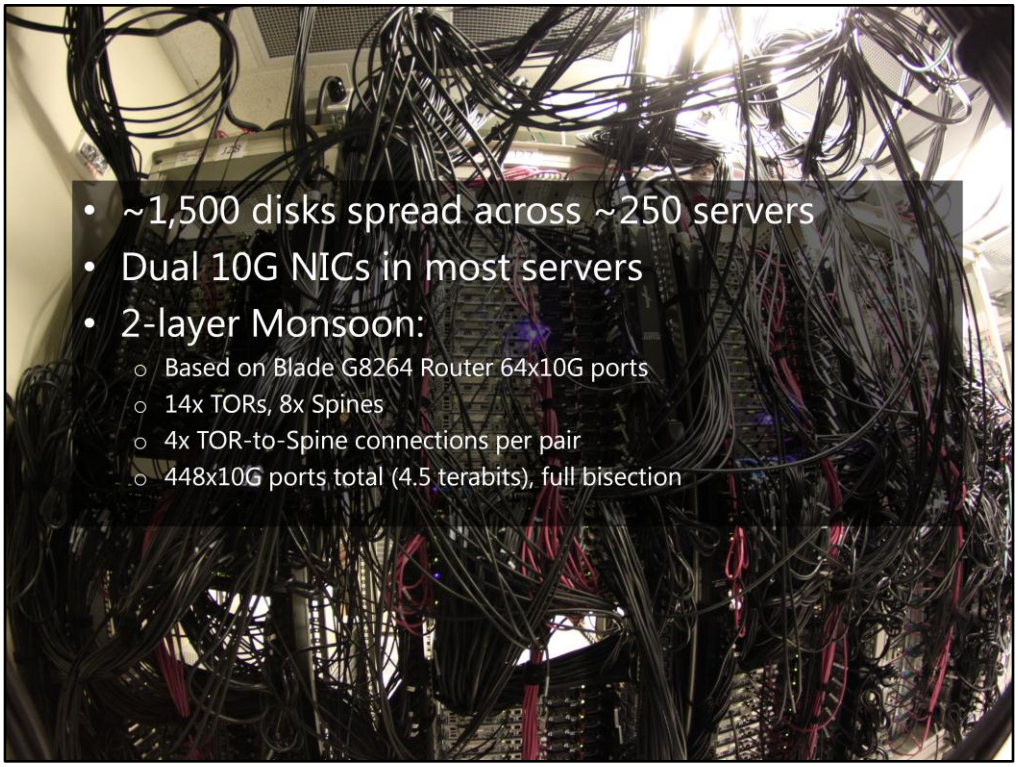
Even with CLOS networks, many computers in today's datacenters still have a bottleneck between disks and the network.
A typical disk can read or write at about a gigabit per second, but there are four, or 12, or even 25 disks in a machine, all stuck behind a single 1 gigabit link.

For applications that have to move data, such as sort or a distributed join, this is a big problem.



In FDS, we make sure all machines with disks have as much network bandwidth as they have disk bandwidth. For example, a machine with 10 disks needs a 10 gigabit NIC, and a machine with 20 disks needs two of them.

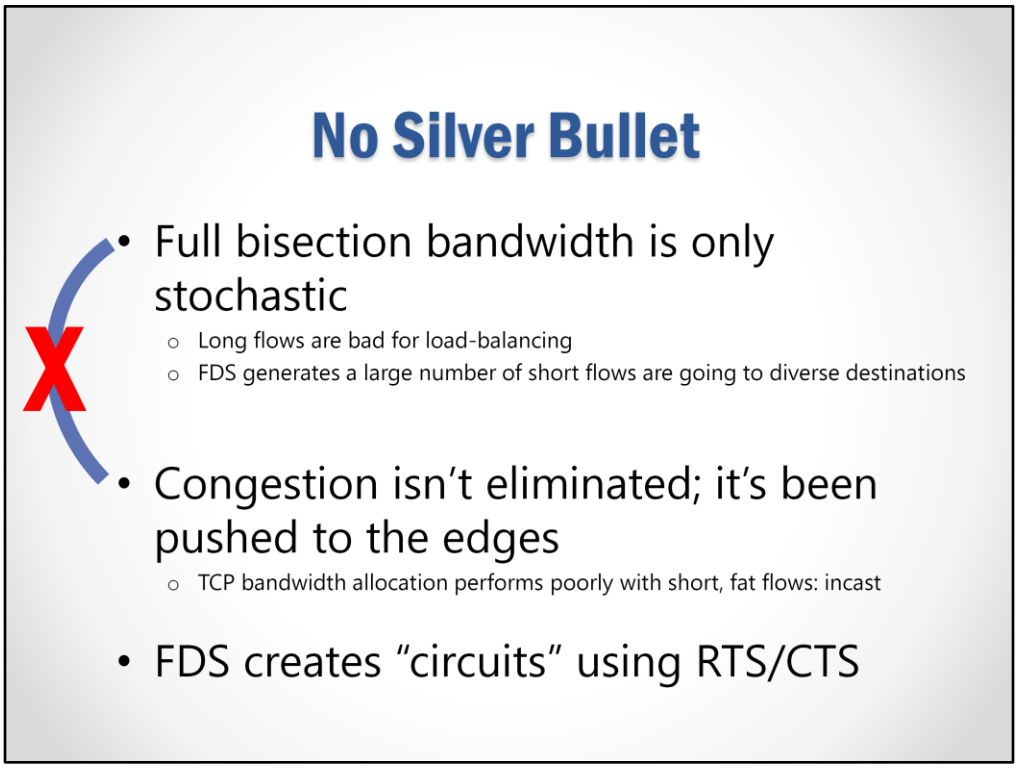
Adding all this bandwidth has a cost; depending on the size of the network, maybe about 30% more per machine. But as we'll see a little later, we get a lot more than a 30% increase in performance for that investment.



So, we really built this.

We've gone through several generations of testbeds, but our largest has 250 machines and about 1,500 disks.
They're all connected using 14 top-of-rack switches and 8 spine switches, giving it 4.5 terabits of bisection bandwidth.

We made a company that sells 10G network cables very happy.



Unfortunately, just adding all this bandwidth doesn't automatically give you good performance, like we thought it would when we were young and innocent at the beginning of the project.

Part of the problem is that in realistic conditions, datacenter CLOS networks don't guarantee full bisection bandwidth. They only make it stochastically *likely*. This is an artifact of routing algorithms that select a single, persistent path for each TCP flow to prevent packet reordering. As a result, CLOS networks have a well-known problem handling long, fat flows. We designed our data layout how we did partly because of the network load it generates. FDS generates a large number of very short-lived flows to a wide set of random destinations, which is the ideal case for a CLOS network.

A second problem is that even a perfect CLOS network doesn't actually eliminate congestion. It just pushes the congestion out to the edges. So good traffic shaping is still necessary.

But what's really nasty is that these two constraints are in tension. CLOS networks need really SHORT flows for load balancing, but TCP needs nice LONG flows for its bandwidth allocation algorithm to find an equilibrium.

We ended up doing our own application-layer bandwidth allocation using an RTS/CTS scheme which is described in the paper, along with a bunch of other tricks.

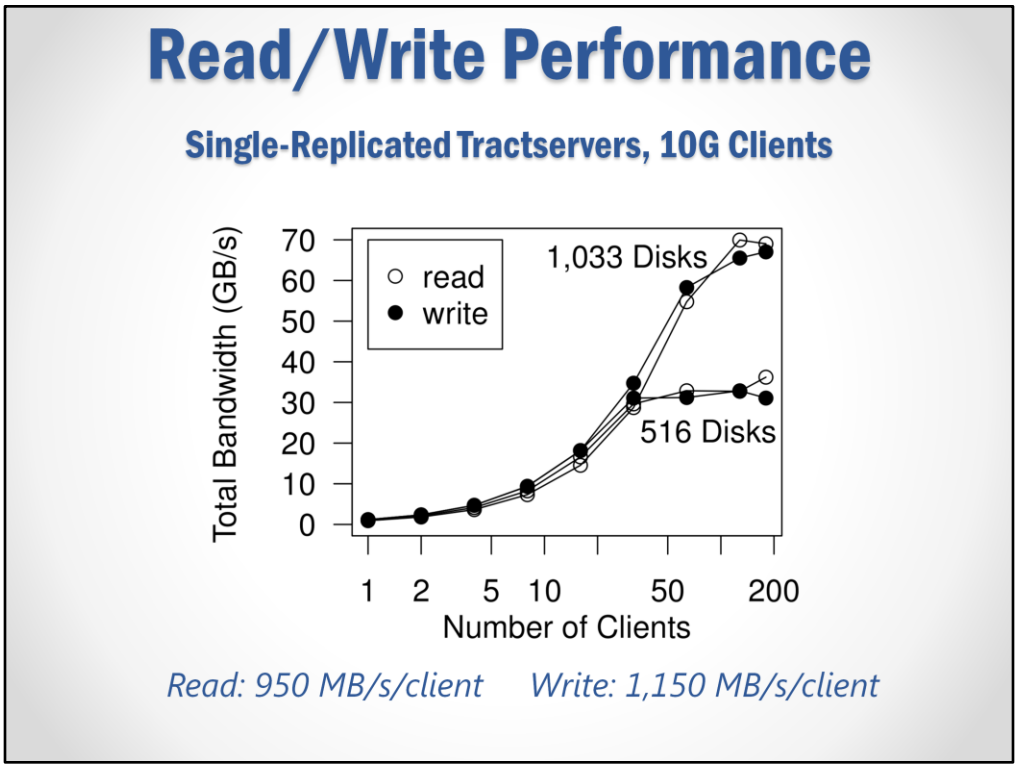
Outline

- FDS is **simple, scalable blob storage**; logically separate compute and storage without the usual performance penalty
- **Distributed metadata management**, no centralized components on common-case paths
- Built on a **CLOS network** with distributed scheduling
- **High read/write performance** demonstrated (2 Gbyte/s, single-replicated, from one process)
- **Fast failure recovery** (0.6 TB in 33.7 s with 1,000 disks)
- **High application performance** – set the 2012 world record for disk-to-disk sorting

Now, let me show you some microbenchmarks from one of our test clusters.

In these experiments, we had test clients READING FROM or WRITING TO a fixed number of tractservers. We varied the number of clients and measured their aggregate bandwidth.

The clients each had a single 10 gig Ethernet connection, and the servers had either one or two, depending on how many disks were in the server.



The first result I'll show you is from a single-replicated cluster.

Note the X-axis here is logarithmic.

And I have the pleasure of showing you a graph that is delightfully uninteresting. The aggregate read and write bandwidth go up close to linearly with the number of clients, from 1 to 170. Read bandwidth goes up at about 950 megabytes per second per client and write bandwidth goes up by 1,150 megabytes per second per client. Writers saturated about 90% of their theoretical network bandwidth, and readers saturated about 74%.

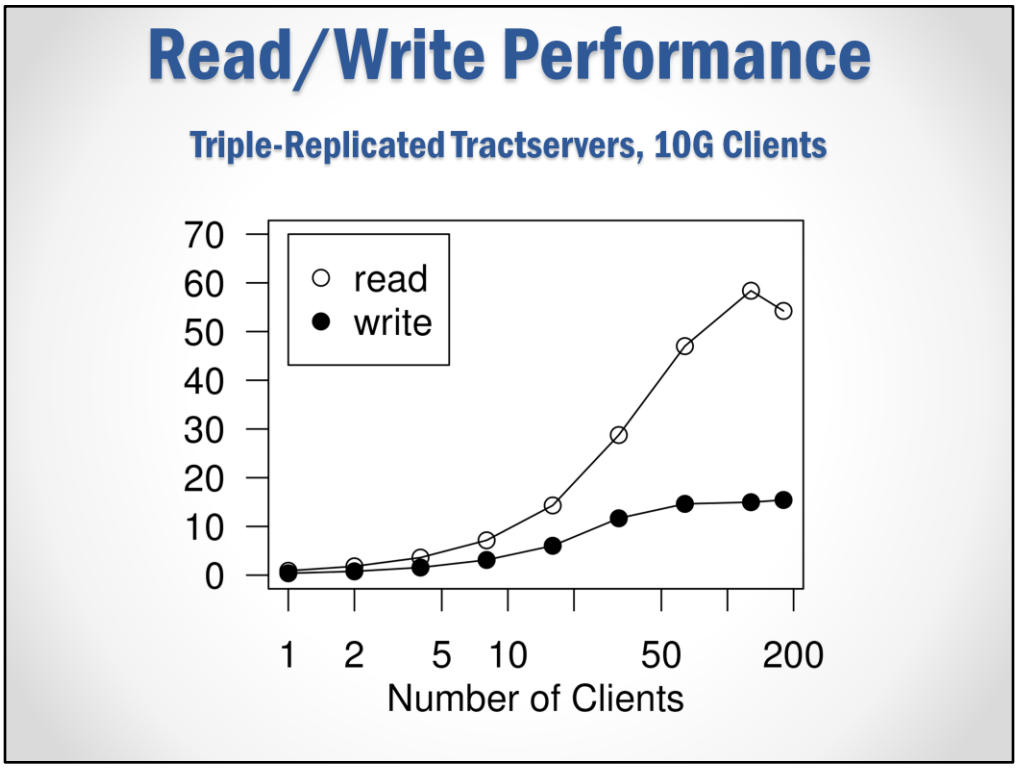
This graph shows two different cluster configurations: one used 1,033 disks, and the other used about half that.

In the 1,033 disk test, there was just as much disk bandwidth as there was client bandwidth, so performance kept going up as we added more clients.

In the 516 disk test, there was much MORE CLIENT bandwidth available than DISK bandwidth. Since disks were the bottleneck, aggregate bandwidth kept going up until we'd saturated the disks, then got flat.

It's not shown on this graph, but we also tested clients that had 20 gigabits of network bandwidth instead of ten. There, clients were able to read and write at over 2 gigabytes per second. That's writing remotely, over the network, all the way to disk platters, faster than a

lot of systems write to a local RAID. Decoupling storage and computation does not have to mean giving up performance!



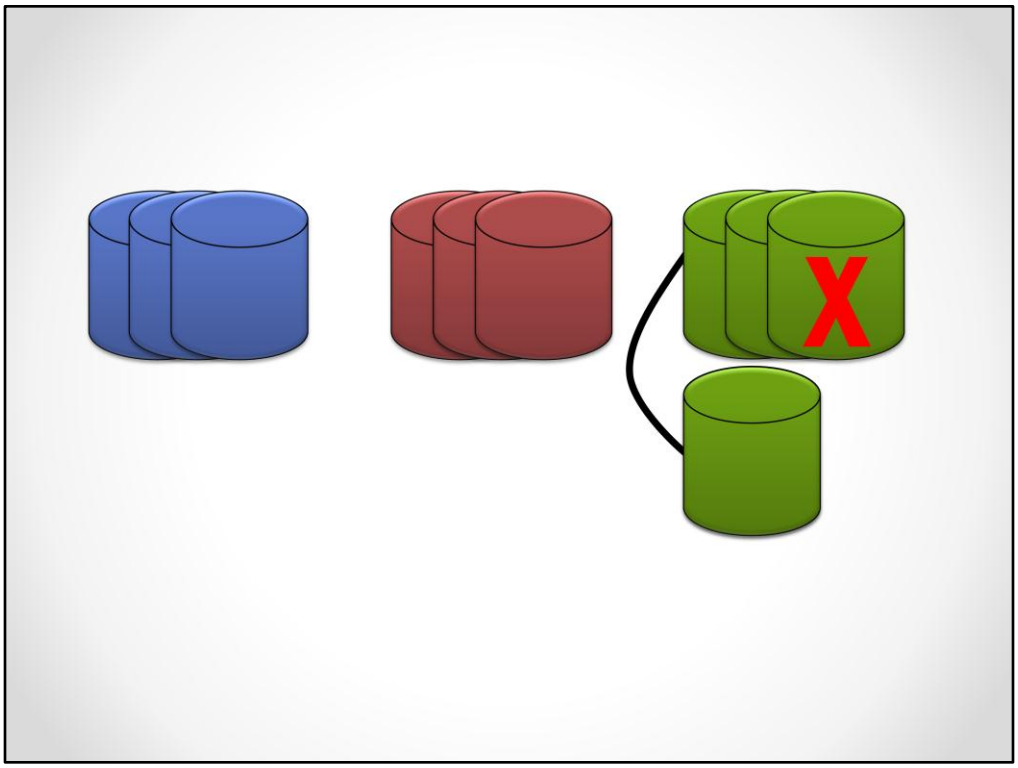
This test is similar to the previous one but we're now writing to a triple-replicated cluster instead of a single-replicated cluster.

Read bandwidth is just about the same, but as you'd expect, writes saturate the disks much sooner because we're writing everything three times. The aggregate write bandwidth is about one-third of the read bandwidth in all cases.

Outline

- FDS is **simple, scalable blob storage**; logically separate compute and storage without the usual performance penalty
- **Distributed metadata management**, no centralized components on common-case paths
- Built on a **CLOS network** with distributed scheduling
- **High read/write performance** demonstrated (2 Gbyte/s, single-replicated, from one process)
- **Fast failure recovery** (0.6 TB in 33.7 s with 1,000 disks)
- **High application performance** – set the 2012 world record for disk-to-disk sorting

I'd like to switch gears now, a little bit, and talk about failure recovery. I'll first describe how it works in FDS, and then show you some more performance numbers.

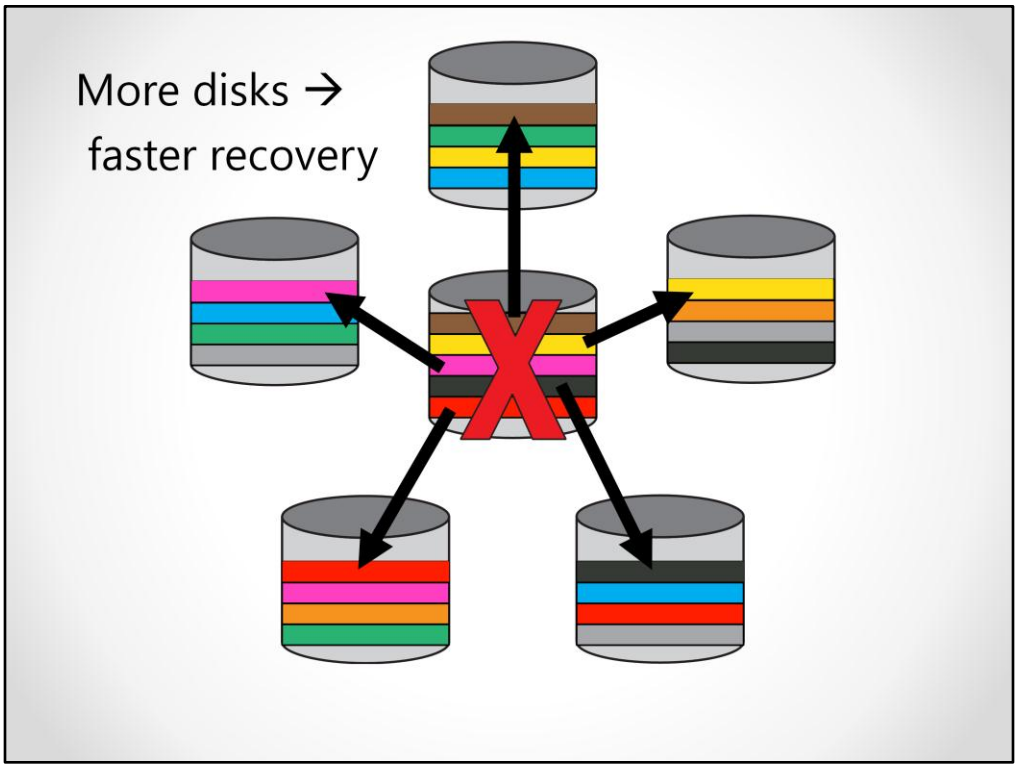


The way that data is organized in a blob store has a pretty dramatic effect on recovery performance.

The simplest method of replication is unfortunately also the slowest. You could organize disks into pairs or triples that are always kept identical. When a disk fails, you bring in a hot spare, and use a replica that's still alive to make an exact copy of the disk that died.

This is really slow because it's constrained by the speed of a single disk. Filling a 1 terabyte disk takes at least several hours, and such slow recovery actually DECREASES DURABILITY because it lengthens the window of vulnerability to additional failures.

But we can do better. In FDS, when a disk dies, our goal isn't to reconstruct an exact duplicate of the disk that died. We just want to make sure that somewhere in the system, extra copies of the lost data get made. It doesn't matter where. We just want to get back to a state where there are 3 copies of everything SOMEWHERE.



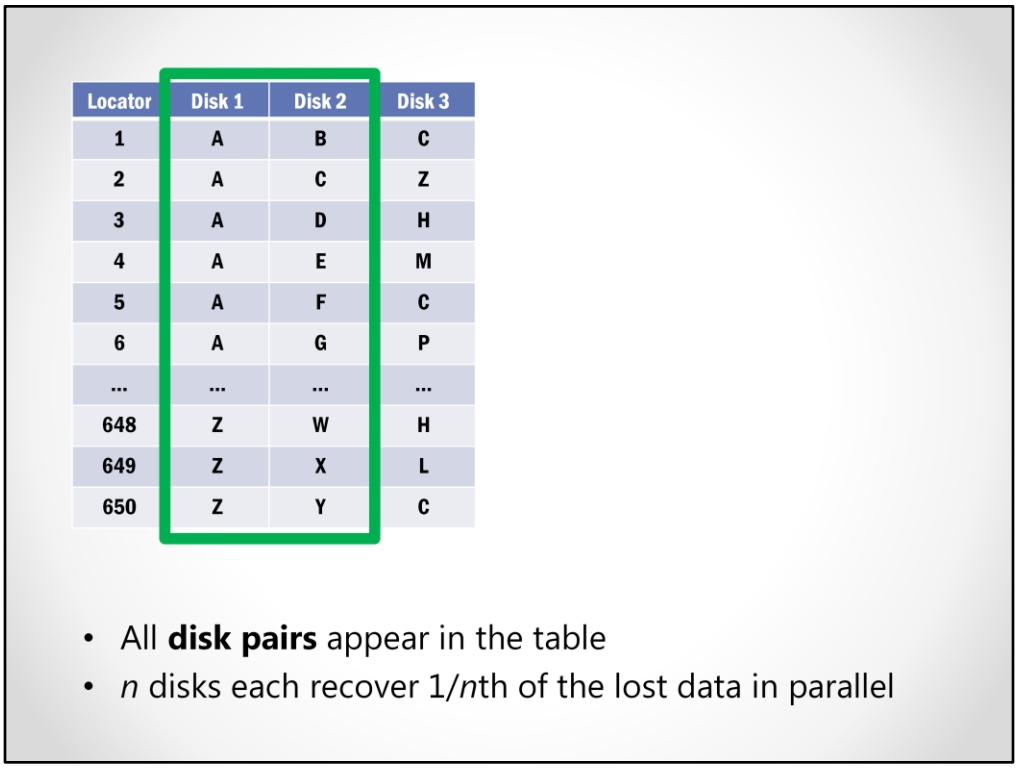
In FDS, failure recovery exploits the fine-grained blob striping I was talking about earlier.

We lay out data so that when a disk dies (***), there isn't just a single disk that contains backup copies of that disk's data. Instead, all n of the disks that are still alive (***) will have about 1/nth of the data that was lost.

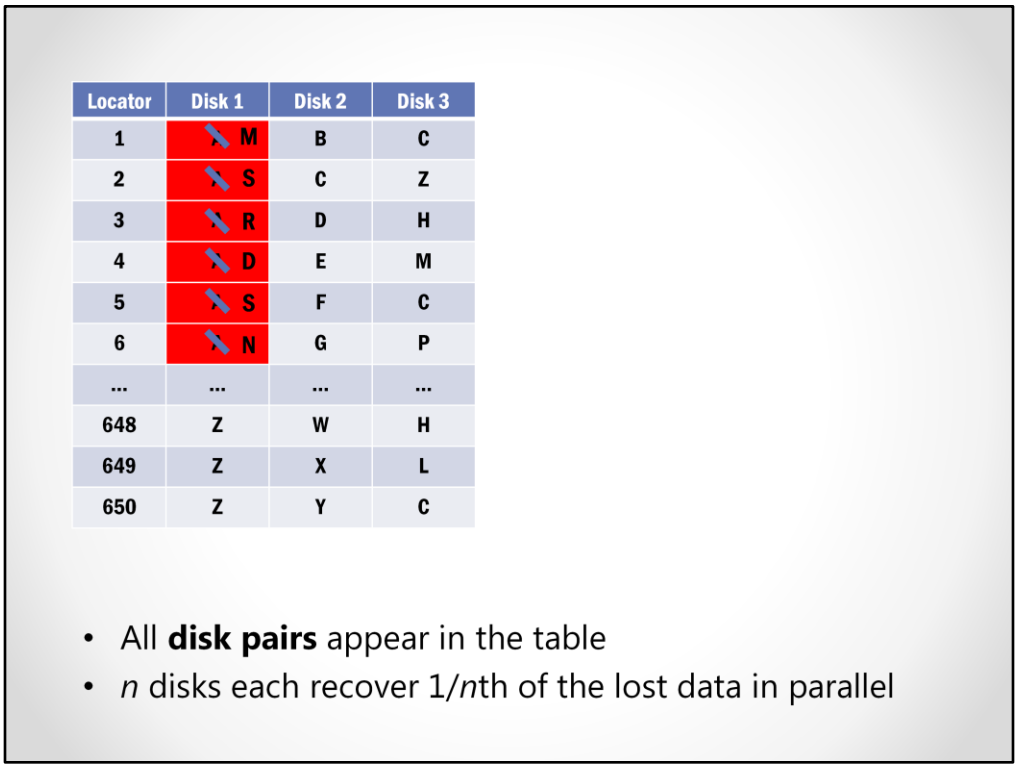
Every disk sends a copy of its small part of the lost data to some other disk that has some free space. (***)

Since we have a full bisection bandwidth network, all the disks can do this in parallel, and failure recovery goes really fast.

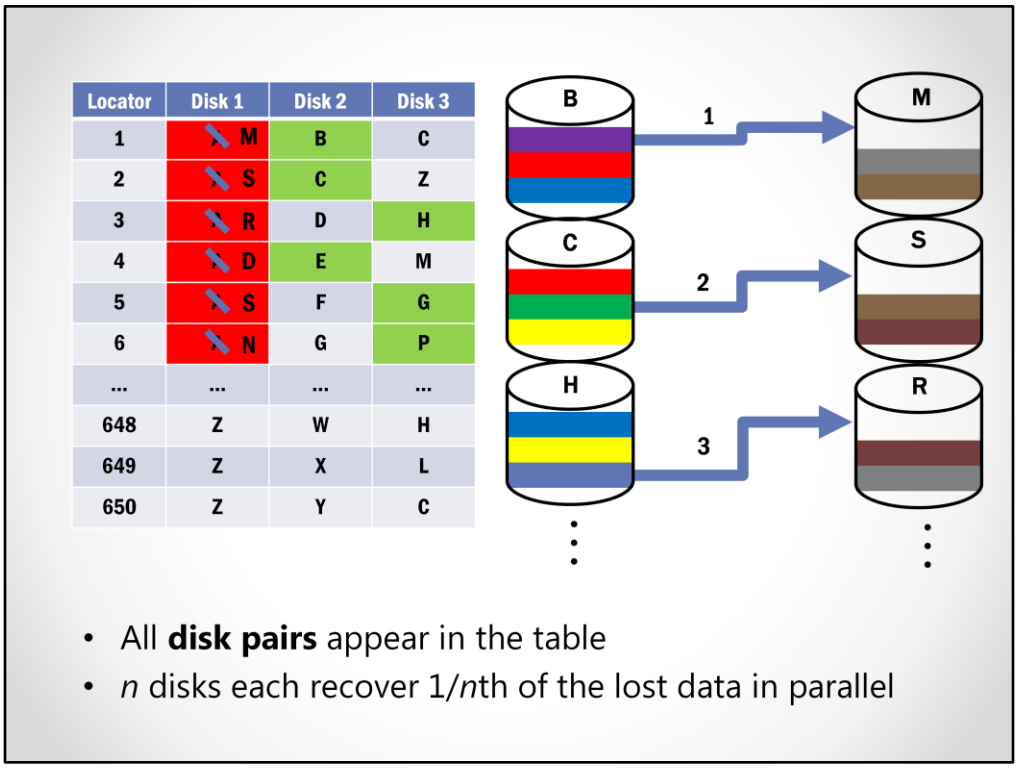
In fact, since EVERY disk is participating in the recovery, FDS has a really nice property: as a cluster gets larger, recovery actually goes faster. (***) THIS is sort of amazing, because it's just the opposite of something like a RAID, where larger volumes require LONGER recovery times.



It's pretty easy to implement this using the tract locator table. What we do is construct a table such that that every possible PAIR (***) of disks appears in a row of the table. This is why, in replicated clusters, the number of rows in the table grows as n-squared. We can add more COLUMNS for more durability if we WANT to, but to get the fastest recovery speed, we never need more than n-squared rows.



When a disk dies, such as Disk A in this example, the metadata server (***) first selects a random disk to replace the failed disk in every row of the table.



Then, it selects one of the remaining GOOD disks in each row to transfer the lost data to the replacement disk.

It sends a message to each disk to start the transfer.(***)

And, since we're on a full bisection bandwidth network, all the transfers can happen in parallel.

I'm leaving out a lot of details, but they're described in the paper.

Instead, let me show you some results.

Failure Recovery Results

Disks in Cluster	Disks Failed	Data Recovered	Time
100	1	47 GB	19.2 ± 0.7s
1,000	1	47 GB	3.3 ± 0.6s
1,000	1	92 GB	6.2 ± 6.2s
1,000	7	655 GB	33.7 ± 1.5s

- We recover at about 40 MB/s/disk + detection time
- 1 TB failure in a 3,000 disk cluster: ~17s

We tested failure recovery in a number of configurations, in clusters with both 100 and 1,000 disks, and killing both INDIVIDUAL disks and all the disks in a single machine at the same time.

Let me just highlight a couple of the results.

Failure Recovery Results

Disks in Cluster	Disks Failed	Data Recovered	Time
100	1	47 GB	19.2 ± 0.7s
1,000	1	47 GB	3.3 ± 0.6s
1,000	1	92 GB	6.2 ± 6.2s
1,000	7	655 GB	33.7 ± 1.5s

- We recover at about 40 MB/s/disk + detection time
- 1 TB failure in a 3,000 disk cluster: ~17s

In our largest test, we used a 1,000-disk cluster and killed a machine with 7 disks holding a total of about two-thirds of a terabyte of data. All the lost data was re-replicated in 34 seconds.

But, more interesting is that every time we made the cluster larger, we got about another 40 megabytes per second per disk of aggregate recovery speed. That's less than half the speed of a disk, but remember that's because every disk is simultaneously READING the data it's sending, and WRITING to its free space that some other disk is filling.

Extrapolating these numbers out, we estimate that if we lost a 1-terabyte disk out of a 3,000-disk cluster, we'd recover all the data in less than 20 seconds. And remember, that's not to memory, that's writing recovered data all the way out to disk.

What's really magical about these numbers is they are strikingly linear. Recovery time is a combination of some FIXED time to detect the failure plus some VARIABLE time for data transfer. In our experiments, the variable part INCREASES LINEARLY with the amount of data recovered, and DECREASES LINEARLY with the number of disks involved in the recovery. I think that getting such linear results in a system of this scale is really cool.

Outline

- FDS is **simple, scalable blob storage**; logically separate compute and storage without the usual performance penalty
- **Distributed metadata management**, no centralized components on common-case paths
- Built on a **CLOS network** with distributed scheduling
- **High read/write performance** demonstrated (2 Gbyte/s, single-replicated, from one process)
- **Fast failure recovery** (0.6 TB in 33.7 s with 1,000 disks)
- **High application performance** – set the 2012 world record for disk-to-disk sorting

Finally I'd like to discuss one application of FDS. We describe several in the paper from a few different domains, but I'm just going to highlight one: we set the 2012 world record for disk-to-disk sorting using a small FDS application.

Minute Sort

- Jim Gray's benchmark: How much data can you sort in 60 seconds?
  - Has real-world applicability: sort, arbitrary join, group by <any> column

System	Computers	Disks	Sort Size	Time	Disk Throughput
MSR FDS 2012	256	1,033	1,470 GB	59 s	46 MB/s
Yahoo! Hadoop 2009	1,408	5,632	500 GB	59 s	3 MB/s

15x efficiency improvement!

- Previous "no holds barred" record – UCSD (1,353 GB); FDS: 1,470 GB
  - Their purpose-built stack beat us on CPU efficiency, however
- Sort was "just an app" – FDS was not enlightened
  - Sent the data over the network thrice (read, bucket, write)
  - First system to hold the record without using local storage

MinuteSort is a test devised by a group led by the late Jim Gray about 25 years ago. The question is, given 60 seconds, how much randomly distributed data can you shuffle into sorted order? This was meant as an **I/O** test, so the rules specify the data must START and END in stable storage.

We competed in two divisions. One was for general-purpose systems, and one was for purpose-built systems that were allowed to exploit the specifics of the benchmark.

In the general purpose division, the previous record, which stood for three years, was set by Yahoo, using a large Hadoop cluster. By large, I mean about 1,400 machines, and about 5,600 disks. With FDS, with less than ONE-FIFTH of the computers and disks, we nearly TRIPLED the amount of data sorted … which multiplies out to a pretty remarkable 15x improvement in efficiency.

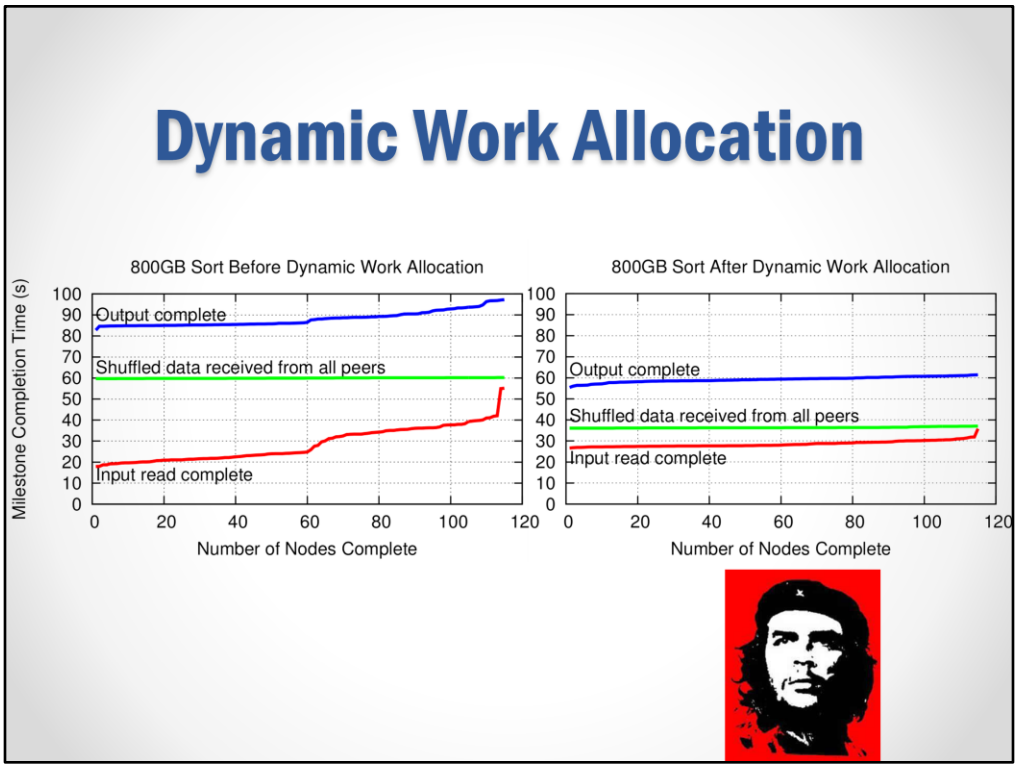
This all came from the fact that Yahoo's cluster, like MOST Hadoop-style clusters, had serious oversubscription both from disk to network, and from rack to network core.

We ATTACKED that bottleneck, by investing, on average, 30 percent more money per machine for more bandwidth, and HARNESSED that bandwidth using everything you heard in this talk. The result is that instead of a cluster having mostly IDLE disks, we built a cluster with disks working CONTINUOUSLY.

In the specially-optimized class, the record was set last year by UCSD's fantastic TritonSort.

They wrote a tightly integrated and optimized sort application that did a beautiful job of really squeezing everything they could out of their hardware. They used local storage, so they did beat us on CPU efficiency, but not on disk efficiency. In absolute terms, we set that record by about 8%.

But I think what really distinguishes our sort is that it REALLY WAS just a small app sitting on top of FDS. FDS is general-purpose, and had absolutely no sort-specific optimizations in it at all. Sort used no special calls into FDS, just plain old read and write. In fact, that meant we had to transport the data three times: the clients read data over the network from tractservers, then exchanged, then wrote back to the tractservers, but we beat both records anyway.



I also want to take a minute to show you the effect of dynamic work allocation. At the beginning of the talk I mentioned that one advantage of ignoring locality constraints, like in the little-data computer, is that all workers can draw work from a global pool, which prevents stragglers.

Early versions of our sort didn't use dynamic work allocation. We just divided the input file evenly among all the nodes.

As you can see with this time diagram, stragglers were a big problem. Each line represents one stage of the sort. A horizontal line would mean all nodes finished that stage at the same time, which would be ideal. But as you can see, the red stage was pretty far from ideal. About half the nodes would finish the stage within 25 seconds and a few would straggle along for another 30. This was bad because there was a global barrier between the red stage and the green stage.

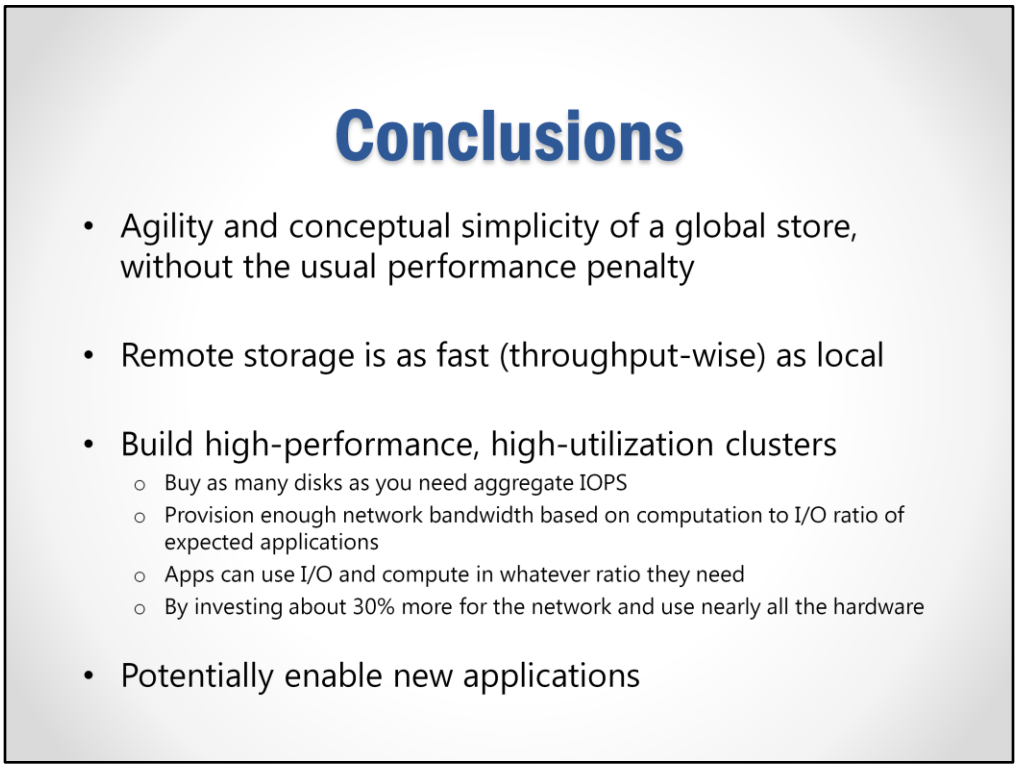
Now, we knew it wasn't a problem with the hardware because DIFFERENT nodes were coming in last each time. It was just a big distributed system with a lot of random things happening, and a few nodes would always get unlucky.

Finally we switched to using dynamic work allocation. These two tests were actually done on the same day with no other change.

In the version on the right, each node would process a tiny part of the input, and when it

was almost done, it would ask the head sort node for more. This was all done at the application layer – FDS didn't know anything about it. As you can see, it dramatically reduced stragglers, which made the whole job faster. It was a really nice self-clocking scheme. A worker that finished early would get lots more work assigned, and unlucky nodes wouldn't. And this was entirely enabled by the fact that FDS uses a global store. Clients can read any part of they input they want, so shuffling the assignments around at the last second really has no cost.

And, yes, I admit it: this is digital socialism.



In conclusion, FDS gives you the agility and conceptual simplicity of a global store, but without the usual performance penalty.

We can write to remote storage just as fast as other systems can write to local storage, but we're able to throw away the locality constraints.

This also means we can build clusters with very high utilization: you buy as many disks as you need I/O bandwidth, and as many CPUs as you need processing power. Individual applications can use resources in whatever ratio they need. We do have to invest more money in the network, but in exchange we really unlock the potential of all the other hardware we've paid for, both because we've opened the network bottleneck and because a global store gives us global statistical multiplexing.

Today, a lot of people have the mindset that certain kinds of high-bandwidth applications just HAVE to fit into a rack if they're going to be fast. And racks really aren't that big! I'd argue we've shown a path around that constraint. The bottom line with FDS is really not JUST that it can make today's applications go faster. I think it lets us imagine NEW KINDS of applications, too.

Thank you!