Flat Datacenter Storage

JEREMY ELSON AND EDMUND B. NIGHTINGALE

Jeremy Elson received his PhD from UCLA in 2003. He has worked in wireless sensor networks, time synchronization, online mapmaking, CAPTCHAs, and distributed storage. He also enjoys riding bicycles, flying airplanes, and DIY electronics.
jelson@microsoft.com

Ed Nightingale has worked at Microsoft since graduating with a PhD from the University of Michigan in 2007. Ed's favorite research areas include operating systems and distributed systems. Outside of work, Ed enjoys bicycling, reading, and attempting to keep up with his children.
ed.nightingale@microsoft.com

There's been an explosion of interest in Big Data—the tools and techniques for handling very large data sets. Flat Datacenter Storage (FDS) is a new storage project at Microsoft Research. We've built a blob store meant for Big Data, which scales to tens of thousands of disks, makes efficient use of hardware, and is fault tolerant, but still maintains the conceptual simplicity and flexibility of a small computer.

To make this idea concrete, consider the problem of "little data." From a systems perspective, little data is essentially a solved problem. The perfect little-data computer has been around for years: a single machine with multiple processors and disks interconnected by something like a RAID controller. For I/O-intensive workloads, such a computer is ideal. When applications write, the RAID controller splits the writes up and stripes them over all the disks. There might be a small number of writers writing a lot, or a large number of writers writing a little bit, or a mix of both. The lulls in one writer are filled in by the bursts in another, giving us good statistical multiplexing. All the disks stay busy, and high utilization means we're extracting all the performance we can from our hardware. Reads can also exploit the striped writes. Even if some processes consume data slowly and others consume it quickly, all the disks stay busy, which is what we want.

Writing software for this computer is easy, too. How many physical disks there are doesn't matter; programmers can pretend there's just one big one. Files written by any process can be read by any other without caring about locality. 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; 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 the little-data computer is that it's easy to adjust the ratio of processors to disks by adding more of whichever is needed. An administrator can buy machine resources to match the expected workload, fully and efficiently making use of the hardware budget.

This machine has one major drawback: it doesn't scale. We can add a few dozen processors and disks, but not thousands. The limitation lies in the fact that such a system relies on a single, centralized I/O controller. Roughly, the controller is doing two things:

- It manages metadata. When a process writes, the controller decides how the write should be striped, and records enough state so that reads can find the data later.
- It physically routes the data between disks to processors—actually transporting the bits.

In FDS, we've built a blob store that fully distributes both of these tasks. This means we can build a cluster that has the essential properties of the ideal little-data machine, but can scale to the size of a datacenter. To maintain conceptual simplicity, computation and storage are logically separate. There is no affinity, meaning any processor can access all data in the system uniformly—that's why we call it "flat;" however, it still achieves very high I/O performance that has come to be expected only from systems that couple storage and computation together, such as MapReduce, Dryad, and Hadoop.

We've developed 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 read or write loop can achieve more than 2 GBps all the way to the remote disk platters. In other words, FDS applications can write 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 seven 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 wrote a straightforward sort application on top of FDS that beat the world record for disk-to-disk sorting in 2012. Our general-purpose remote blob store beat previous implementations that exploited local disks. We've also experimented with applications from other domains, including stock market analysis and serving an index of the Web.

The Basics

In FDS, all blobs are identified with a simple GUID. Each blob contains 0 or more allocation units we call *tracts*. Tracts are numbered sequentially, starting from 0 (Figure 1).

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

The programming interface is simple; it has only about a dozen calls, such as *CreateBlob, ReadTract,* and *WriteTract*. The interface is designed to be asynchronous, meaning that the functions don't block, but rather call a callback when they're done. A typical high-throughput FDS application will start out by issuing a few dozen reads or writes in parallel, then issue more as the earlier ones complete. We call applications using the FDS API the *FDS clients*.

In addition to clients, there are two other types of actors in FDS. The first is the *tractserver*, lightweight software that sits between a raw disk and the network, accepting commands from the network such as "read a tract" and "write a tract."

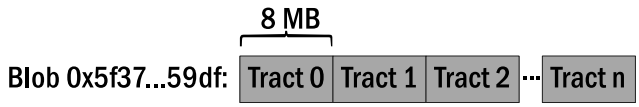

Figure 1: Blobs and tracts

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

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.

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 essentially every read or write, clients consult a metadata server that has canonical information about the placement of all data in the system. This gives administrators excellent visibility and control; however, it is also a centralized bottleneck that has exerted pressure on these systems to increase the size of writes. For example, GFS uses 64 megabyte extents, nearly an order of magnitude larger than FDS tracts. This makes it harder to do fine-grained load balancing like the ideal little-data computer does.

On the other end of the spectrum are distributed hash tables. They're fully decentralized, but all reads and writes typically require multiple trips over the network before they find data. Additionally, 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 limited. When a client first starts, the metadata server sends some state to the client. For now, think of this state 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 IP addresses 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 spread load *uniformly*. To achieve high performance, FDS clients have lots of tract reads and writes outstanding simultaneously. The oracle needs to ensure (or, at least, make it likely) that all of those operations are being serviced by different tractservers. We don't want all the requests going to just one disk if we have ten of them.

Locator	Replica 1	Replica 2	Replica 3
0	A	B	C
1	A	D	F
2	A	C	G
3	D	E	G
4	B	C	F
...	...	...	...
1,526	LM	TH	JE

Figure 2: An example tract locator table. Each letter represents a disk.

Once a client has this oracle, reads and writes all happen without contacting the metadata server again. Because 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 large-scale systems, giving us really good statistical multiplexing of the disks—just like the little-data computer.

This technique gives us the flexibility to make writes as small as we need to. For throughput-sensitive applications, we use 8 MB tracts: large enough to amortize seeks and make random reading and writing almost as fast as doing so sequentially. We have also experimented with *seek-bound* workloads, where we reduced the tract size all the way down to 64 KB. That's hard with a centralized metadata server but no problem with our oracle.

So, what is this oracle? Simply, it is a table of all the disks in the system, collected centrally by the metadata server. We call this table the *tract locator table*, or TLT. The table has as many columns as there are replicas; the example in Figure 2 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^2; we'll see why a little later.

For each read or write operation, the client finds a row in this table by taking the blob GUID and tract number and deterministically transforming them into a row index:

```
Table_Index=(Hash(Blob_GUID) + Tract_Number) mod TLT_Length
```

As long as readers and writers are using consistent versions of the table, the mappings they get will also be consistent. (We describe how we achieve consistent table versioning in our full paper [3].) 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.

A critical property of this table is that it only contains disks, not tracts. In other words, reads and writes don't change the table. This means clients can retrieve it from the metadata server once, then never contact the metadata server again. The TLT only changes when a disk fails or is added.

There's another clever thing we can do with the tract locator table: use it to fully distribute the *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. 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, imagine that 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.

Unlike data writes, which go directly from the client to all replicas, metadata operations in multiply replicated systems go to only one tractserver—the one 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; however, requests that need to be serialized because they refer to the same blob will always end up at the same tractserver, thus maintaining correctness.

Networking

So far, we've assumed that there was an uncongested path from tractservers to clients. We now turn to the question of how to build such a network.

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

There has been a recent surge of research in the networking community in Clos networks [2]. Clos networks more or less do for networks what RAID did for disks: by connecting up a large number of low-cost, commodity routers and doing some clever routing, building full bisection bandwidth networks at the scale of a datacenter is now economical.

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 typical machine, all stuck behind a single one-gigabit link. For applications that have to move data, such as a 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. Of course, adding bandwidth has a cost; depending on the size of the network, we estimate about 30% more per machine. But as we'll explain a little later, we get a lot more than a 30% increase in performance for that investment.

We've gone through several generations of testbeds; the largest has 250 machines and about 1,500 disks. They're all connected using 14 top-of-rack routers and eight spine routers. Each router has 64 10-gigabit ports. The top-of-rack routers split their 64 ports into two halves: 32 ports connect to computers (clients or tractservers) and 32 connect to spine routers. There is a 40 Gbps connection between each top-of-rack and spine router—four 10 Gbps ports bonded together. In aggregate, this gives us more than 4.5 terabits of bisection bandwidth.

Unfortunately, just adding all this bandwidth doesn't automatically produce a storage system with good performance. 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 such as ECMP (equal-cost multipath routing) 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. In FDS, our data layout is designed to spread data uniformly across disks partly because of the network load that such an access pattern generates. FDS clients use a large number of very short-lived

flows to a wide set of pseudo-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. Good traffic shaping is still necessary to prevent catastrophic collisions at receivers—a condition known as *incast* [6].

What's particularly unfortunate is that these two constraints are in tension. Clos networks need *short* flows to achieve good load balancing, but TCP needs *long* flows for its bandwidth allocation algorithm to find an equilibrium that prevents collisions.

In FDS, we ended up doing our own application-layer bandwidth allocation using a hybrid request-to-send/clear-to-send (RTS/CTS) scheme reminiscent of that found in wireless networks. Large messages are queued at the sender, and the receiver is notified with an RTS. The receiver limits the number of CTSes it allows outstanding, thus limiting the number of senders competing for its receive bandwidth. Small messages, such as control messages and RTS/CTS, are delivered over a different TCP flow from the large messages, reducing latency by enabling them to bypass long queues. FDS network message sizes are bimodal: large messages are almost all about 8 MB, and most other messages are 1 KB or smaller.

Microbenchmarks

Our full paper [3] has a more thorough evaluation of FDS. Here, we'll describe one set of microbenchmarks: testing the speed of simple test clients that read from or wrote to a fixed number of tractservers. We varied the number of clients and measured their aggregate bandwidth. The clients each had a single 10 Gbps Ethernet connection. The tractservers had either one or two, depending on how many disks were in the server.

Figure 3 shows results from a single-replicated cluster. Note the x-axis is logarithmic. 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 MBps per client and write bandwidth goes up by 1,150 MBps per client. Writers saturated about 90% of their theoretical network bandwidth, and readers saturated about 74%.

Two different cluster configurations are depicted: 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

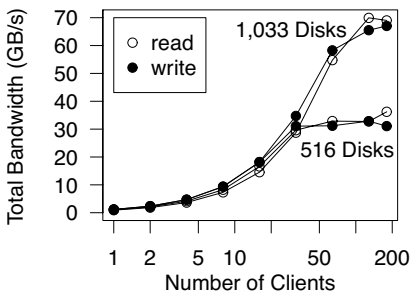

Figure 3: Sequential reading and writing in a single-replicated cluster

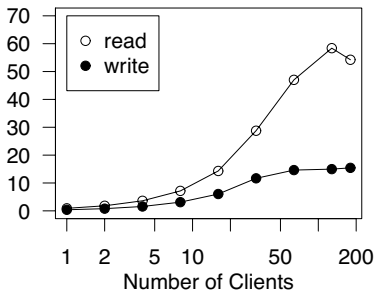

Figure 4: Sequential reading and writing in a triple-replicated cluster

than disk bandwidth. Because disks were the bottleneck, aggregate bandwidth kept going up until we'd saturated the disks, then leveled off.

We also tested clients that had 20 Gbps of network bandwidth instead of 10. These clients were able to read and write at over 2 GBps. In other words, writing remotely over the network all the way to disk platters, these FDS clients were faster than many systems can write to a local RAID. Decoupling storage and computation does not have to mean giving up performance.

Figure 4 shows a similar test against a triple-replicated cluster instead of a single-replicated cluster. Read bandwidth is about the same, but as expected, writes saturate the disks much sooner because clients have to write three times as much data (once for each replica). The aggregate write bandwidth is about one-third of the read bandwidth in all cases.

Failure Recovery

The way that data is organized in a blob store has a dramatic effect on recovery performance. The simplest method of replication is unfortunately also the slowest: mirroring. Disks can be organized into pairs or triples that are always kept identical. When a disk fails, an exact copy of the failed disk is created using an empty spare disk and a replica that's still alive. This is slow because it's constrained by the speed of a single disk. Filling a one terabyte disk takes at least several hours, and such slow recovery decreases durability because it lengthens the window of vulnerability to additional failures.

We can do better. In FDS, when a disk fails, our goal is not to reconstruct an exact duplicate of the failed disk. Instead, we ensure that *somewhere* in the system, extra copies of the lost data get made, returning us to the state where there are three copies of all data.

We exploit our fine-grained striping of data across disks, and lay out data so that when a disk fails, there isn't just a single disk that contains backup copies of that disk's data. Instead, the n disks that remain will each have about $1/n$th of the data lost. Every disk sends a copy of its small part of the lost data to some other disk that has some free space.

Because we have a full bisection bandwidth network, all the disks can do this in parallel, making failure recovery fast. In fact, because *every* disk is participating in recovery, FDS has a nice scaling property: as a cluster gets larger, recovery goes faster. This is just the opposite of systems that use simple mirroring, where larger volumes require *longer* recovery times.

Implementing this scheme using the tract locator table is relatively straightforward. We construct a table such 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^2. We can optionally add more columns for more durability, but to get the fastest recovery speed, we never need more than n^2 rows.

When a disk fails, 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. (Additional details are described in our paper [3].)

Disk Count	100		1,000		
Disks Failed	1	1	1	1	7
Total Stored (TB)	4.7	9.2	47	92	92
GB/disk	47	92	47	92	92
GB Recovered	47	92	47	92	655
Recovery Time (s)	19.2±0.7	50.5±16.0	3.3±0.6	6.2±0.4	33.7±1.5

Table 1: Mean and standard deviation of recovery time after disk failure in a triple-replicated cluster. The high variance in one experiment is due to a single 80 sec. run.

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 (Table 1).

In our largest test, we used a 1,000-disk cluster and killed a machine with seven disks holding a total of 655 GB. All the lost data was recovered in 34 seconds.

More interesting is that every time we made the cluster larger, we got about another 40 MBps per disk of aggregate recovery speed. That's less than half the speed of a disk, but keep in mind 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 TB disk out of a 3,000-disk cluster, we'd recover all the data in less than 20 seconds.

MinuteSort

We've built several big-data applications on top of FDS from domains that include stock market analysis and serving an index of the Web. These applications are described in our recent paper on FDS [3]. In this article, we'll focus on just one: We set two world records in 2012 for disk-to-disk sorting using a small FDS application.

MinuteSort is a test devised by a group led by the late Jim Gray [1]. The question is: given 60 seconds, how much randomly distributed data can be shuffled into sorted order? Because the test was meant as an I/O test, the rules specify the data must start and end in stable storage. We competed in two divisions: one for general-purpose systems, and one for purpose-built systems that were allowed to exploit the specifics of the benchmark.

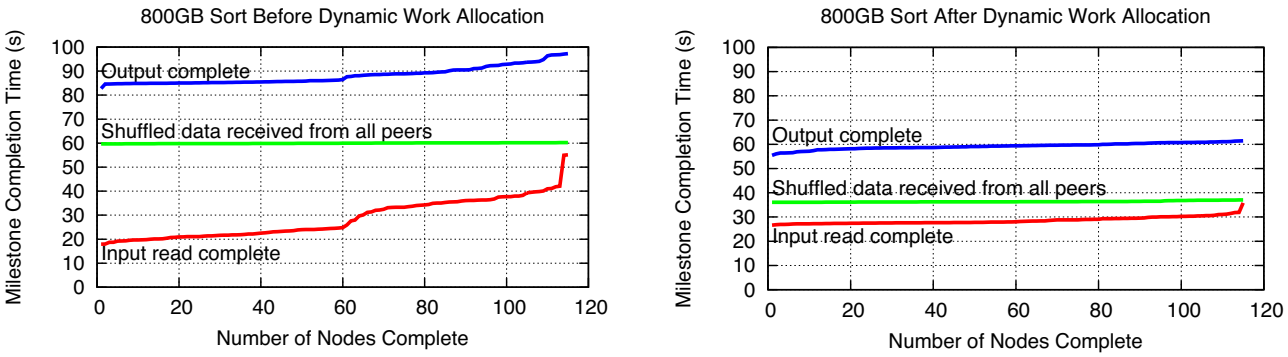

Figure 5: Visualization of the time to reach three milestones in the completion of a sort. The results are shown before (left) and after (right) implementation of dynamic work allocation. Both experiments depict 115 nodes sorting 800 GB.

In the general-purpose division, the previous record, which stood for three years, was set by Yahoo! using a large Hadoop cluster [4] consisting of about 1,400 machines, and about 5,600 disks. With FDS, using less than one-fifth of the computers and disks, we nearly tripled the amount of data sorted, which multiplies out to a 15x improvement in disk efficiency. The gain 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% more money per machine for more bandwidth, and harnessed that bandwidth using the techniques described earlier. 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 TritonSort [5]. They wrote a tightly integrated and optimized sort application that did a beautiful job of 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%. What distinguishes our sort is that it was just a small application sitting on top of FDS, a general-purpose blob store with no sort-specific optimizations.

Dynamic work allocation was a key technique for making our sort fast. We noted earlier that one advantage of ignoring locality constraints—as in the little-data computer—is that all workers can draw work from a global pool, preventing stragglers. Early versions of our sort didn't use dynamic work allocation; we just divided the input file evenly among all the nodes.

As seen in the time diagram in Figure 5 (left), 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. Initially, the red (lowest line) stage was 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 critical because there was a global barrier between the red stage and the green stage (middle line in graph).

We knew the problem did not lie in the hardware because different nodes were the stragglers in each experiment. We concluded that we had built a complex distributed system with a great deal of randomness; a few nodes would always get unlucky. We switched to using dynamic work allocation. In Figure 5 (right), each node would initially process a tiny part of the input. When it was almost done, it would ask the head sort node for more work. This dramatically reduced stragglers, making the whole job faster. A worker that finished early would get more work assigned and unlucky nodes would not. This was entirely enabled by the fact that FDS uses a global store; clients can read any part of the input they want, so shuffling the assignments around at the last second really has no cost.

Conclusion

FDS gives us 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 discard the locality constraints.

This also means we can build clusters with very high utilization; we can buy as many disks as we need for I/O bandwidth, and as many CPUs as we need for processing power. Individual applications can use resources in whatever ratio they need. We

do have to invest more money in the network. In exchange, we 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, many data scientists have the mindset that certain kinds of high-bandwidth applications must fit into a rack if they're going to be fast, but a rack just isn't big enough for many big-data applications. With FDS, we've shown a path around that constraint. FDS doesn't just make today's applications faster. FDS may let us imagine new *kinds* of applications, too.

Acknowledgments

Dave Maltz built our first Clos networks and taught us how to build our own. Johnson Apacible, Rich Draves, and Reuben Olinsky were part of the sort record team. Trevor Eberl, Jamie Lee, Oleg Losinets, and Lucas Williamson provided systems support. Galen Hunt provided a continuous stream of optimism and general encouragement. We also thank Jim Larus for agreeing to fund our initial 14-machine cluster on nothing more than a whiteboard and a promise, allowing this work to flourish.

References

[1] D. Bitton, M. Brown, R. Catell, S. Ceri, T. Chou, D. DeWitt, D. Gawlick, H. Garcia-Molina, B. Good, J. Gray, P. Homan, B. Jolls, T. Lukes, E. Lazowska, J. Nauman, M. Pong, A. Spector, K. Trieber, H. Sammer, O. Serlin, M. Stonebraker, A. Reuter, and P. Weinberger, "A Measure of Transaction Processing Power," *Datamation*, vol. 31, no. 7 (April 1985), pp. 112–118.

[2] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Towards a Next Generation Data Center Architecture: Scalability and Commoditization," *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08 (ACM, 2008), pp. 57–62.

[3] E.B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, "Flat Datacenter Storage," Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12), October 2012.

[4] O. O'Malley and A.C. Murthy, "Winning a 60 Second Dash with a Yellow Elephant," 2009: http://sortbenchmark.org/Yahoo2009.pdf.

[5] A. Rasmussen, G. Porter, M. Conley, H. Madhyastha, R.N. Mysore, A. Pucher, and A. Vahdat, "TritonSort: A Balanced Large-Scale Sorting System," 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11), Boston, MA, April 2011.

[6] V. Vasudevan, H. Shah, A. Phanishayee, E. Krevat, D. Andersen, G. Ganger, and G. Gibson, "Solving TCP Incast in Cluster Storage Systems," 7th USENIX Conference on File and Storage Technologies (FAST '09), San Francisco, CA, February 2009.