Flat Datacenter Storage

Edmund B. Nightingale, Jeremy Elson,
Jinliang Fan, Owen Hofmann[*] Jon Howell, and Yutaka Suzue

Microsoft Research University of Texas at Austin[*]

Abstract

Flat Datacenter Storage (FDS) is a high-performance, fault-tolerant, large-scale, locality-oblivious blob store. Using a novel combination of full bisection bandwidth networks, data and metadata striping, and flow control, FDS multiplexes an application's large-scale I/O across the available throughput and latency budget of *every* disk in a cluster. FDS therefore makes many optimizations around data locality unnecessary. Disks also communicate with each other at their full bandwidth, making recovery from disk failures extremely fast. FDS is designed for datacenter scale, fully distributing metadata operations that might otherwise become a bottleneck.

FDS applications achieve single-process read and write performance of more than 2 GB/s. We measure recovery of 92 GB data lost to disk failure in 6.2 s and recovery from a total machine failure with 655 GB of data in 33.7 s. Application performance is also high: we describe our FDS-based sort application which set the 2012 world record for disk-to-disk sorting.

1 Introduction

A shared and centralized model of storage is one of simplicity. Consider a centralized file server in a small computer science department. Data stored by any computer can be retrieved by any other. This conceptual simplicity makes it easy to use: computation can happen on any computer, even in parallel, without regard to first putting data in the right place. As a result, applications are much less complex than they would be without a shared filesystem.

At the scale of large, big-data clusters that routinely exceed thousands of computers, this "flat" model of storage is still highly desirable. While some blob storage systems such as Amazon S3 [10] provide one, they come with a significant performance penalty because networks at datacenter scales have historically been *oversubscribed*. Individual machines were typically attached in a tree topology [12]; for cost efficiency, links near the root had significantly less capacity than the aggregate capacity below them. Core oversubscription ratios of hundreds to one were common, which meant communication was fast within a rack, slow off-rack, and worse still for nodes whose nearest ancestor was the root.

The datacenter bandwidth shortage has had unfortunate and far-reaching consequences in systems where performance is paramount. Software developers accustomed to treating the network as an abstraction are forced to think in terms of "rack locality." New programming models (e.g., MapReduce [13], Hadoop [1], and Dryad [19]) emerged to help exploit locality, preferentially moving computation to data rather than vice-versa. They effectively expose a cluster's aggregate disk throughput for tasks with high reduction factors (searching for a rare string), but many important computations (sort, distributed join, matrix operations) fundamentally require data movement and are still not well served by systems whose performance is locality-dependent. Software must also be expressed in a data-parallel style, which is unnatural for many tasks.

Counterintuitively, locality constraints can sometimes even *hinder* efficient resource utilization. One example is stragglers: if data is singly replicated, a single unexpectedly slow machine can preclude an entire job's timely completion even while most of the resources are idle. The preference for local disks also serves as a barrier to quickly retasking nodes: since a CPU is only useful for processing the data resident there, retasking requires expensive data movement. If the resident data is not needed, the CPUs may not be usable. In addition, because a node has a fixed ratio of CPUs to disks, tasks that run there will nearly always leave either CPUs or disks partially idle, depending on the resource ratio required by the task.

The root of this cascade of consequences was the locality constraint, itself rooted in the datacenter bandwidth shortage. When bandwidth was scarce, these sacrifices were necessary to achieve the best performance. However, recently developed CLOS networks [16, 15, 24]—large numbers of small commodity switches with redundant interconnections—have made it economical to build non-oversubscribed full bisection bandwidth networks at the scale of a datacenter for the first time. Flat Datacenter Storage (FDS) is a datacenter storage system designed from first principles under the formerly unrealistic assumption that datacenter bandwidth is abundant.

[*] Work completed during a Microsoft Research internship.

Unconventionally for a system of this scale, FDS returns to the flat storage model: all compute nodes can access all storage with equal throughput. Even when computation is co-located with storage, all storage is treated as remote; in FDS, there are no "local" disks. By spreading data over disks uniformly at a relatively fine grain (§2.2), FDS statistically multiplexes workloads over all of the disks in a cluster. FDS effectively eliminates the need to imbue locality constraints into storage systems, schedulers, or programming models.

Despite the conceptual simplicity afforded by the flat storage model, FDS achieves cluster-wide I/O performance on par with systems that exploit locality. Single-process read and write performance exceeds 2 GB/s (§5.2), and FDS dramatically accelerates data movement workloads. For example, our sorting application beat a world record for disk-to-disk sort performance (§6.1) by a factor of 2.8 while using about 1/5 as many disks. It is the first system in the competition's history to do so without exploiting locality.

A consequence of our design is that disk-to-disk bandwidth is also extremely high, facilitating fast recovery from disk and machine failures. In our 1,000 disk cluster, FDS recovers 92 GB lost from a failed disk in 6.2 seconds. Recovery of 655 GB lost from a failed 7-disk machine takes 33.7 seconds (§5.3).

FDS is more efficient for many workloads because every job can use the cluster's I/O bandwidth and CPU resources in exactly the ratio required. FDS therefore moves away from conflating high performance with data-parallel programming. Further, since data locality is immaterial to compute nodes, they are easily and quickly retasked. This can even be done at a fine grain within a single task; as demonstrated in §2.4, *dynamic work allocation* retasks at the granularity of individual data reads to dramatically reduce the effect of stragglers.

Much past research has been directed towards solving the individual problems brought about by the need for locality. FDS, in contrast, is a clean redesign from which many of these solutions fall out naturally. Our goal is to move datacenters back to a flat storage model so that these benefits may be widely realized.

The remainder of this paper is organized as follows. Section 2 gives an overview of the design of FDS. Section 3 describes FDS' strategy for replication and failure recovery. Section 4 explores our network in greater detail and describes our novel congestion avoidance strategy needed in full bisection bandwidth networks. Section 5 presents microbenchmarks. Section 6 reviews how FDS can speed up real workloads, including sort and serving a web index. We review related work in Section 7 and conclude in Section 8.

2 Design Overview

FDS' main goal is to expose all of a cluster's disk bandwidth to applications. Blobs are divided into *tracts* (§2.1), parallelizing both storage of data (§2.2) and handling of metadata (§2.3) across all disks. We exploit our locality-oblivious storage to dynamically assign work to workers, preventing stragglers (§2.4). FDS provides the best of both worlds: a scale-out system with aggregate I/O throughput equal to systems that exploit local disks combined with the conceptual simplicity and flexibility of a logically centralized storage array.

For simplicity we will first describe the system without regard to fault tolerance. §3 will describe replication, failure recovery, and cluster growth. In addition, this section assumes the network core is never congested; our network is described in §4.

2.1 Blobs and Tracts

In FDS, data is logically stored in *blobs*. A blob is a byte sequence named with a 128-bit GUID. The GUID can either be selected by the application or assigned randomly by the system. Blobs can be any length up to the system's storage capacity. Reads from and writes to a blob are done in units called *tracts*. Each tract within a blob is numbered sequentially starting from 0. Blobs and tracts are mutable; nothing prevents a client from overwriting a previously-written tract with new data.

Tracts are sized such that random and sequential access achieves nearly the same throughput. In our cluster, tracts are 8 MB (§5.1). The tract size is set when the cluster is created based upon cluster hardware. For example, if flash were used instead of disks, the tract size could be made far smaller (e.g., 64 kB).

Every disk is managed by a process called a *tractserver* that services read and write requests that arrive over the network from clients. Tractservers do not use a file system. Instead, they lay out tracts directly to disk by using the raw disk interface. Since there are only about 10^6 tracts per disk (for a 1 TB disk), all tracts' metadata is cached in memory, eliminating many disk accesses.

Tractservers and their network protocol are not exposed directly to FDS applications. Instead, these details are hidden in a client library with a narrow and straightforward interface. Figure 1 shows a simplified version of it; some parameters and return values have been elided. In addition to the listed parameters, each function takes a callback function and an associated context pointer. All calls in FDS are non-blocking; the library invokes the application's callback when the operation completes.

The application's callback functions must be reentrant; they are called from the library's threadpool and may overlap. Tract reads are not guaranteed to arrive in order of issue. Writes are not guaranteed to be committed in order of issue. Applications with ordering require-

Getting access to a blob
`CreateBlob(UINT128 blobGuid)`
`OpenBlob(UINT128 blobGuid)`
`CloseBlob(UINT128 blobGuid)`
`DeleteBlob(UINT128 blobGuid)`
Interacting with a blob
`GetBlobSize()`
`ExtendBlobSize(UINT64 numberOfTracts)`
`WriteTract(UINT64 tractNumber, BYTE *buf)`
`ReadTract(UINT64 tractNumber, BYTE *buf)`
`GetSimultaneousLimit()`

Figure 1: FDS API

ments are responsible for issuing operations after previous acknowledgments have been received, rather than concurrently. FDS guarantees atomicity: a write is either committed or failed completely.

The non-blocking API helps applications achieve good performance. By spreading a blob's tracts over many tractservers (§2.2) and issuing many requests in parallel, many tractservers can begin reading tracts off disk and transferring them back to the client simultaneously. In addition, deep read-ahead allows a tract to be read off disk into the tractserver's cache while the previous one is transferred over the network. The FDS API `GetSimultaneousLimit()` tells the application how many reads and writes to issue concurrently. A typical simultaneous limit is 50 tracts, though the exact value depends on the client's bandwidth.

2.2 Deterministic data placement

A key issue in parallel storage systems is data placement and rendezvous, that is: how does a writer know where to send data? How does a reader find data that has been previously written?

Many systems solve this problem using a metadata server that stores the location of data blocks [14, 30]. Writers contact the metadata server to find out where to write a new block; the metadata server picks a data server, durably stores that decision and returns it to the writer. Readers contact the metadata server to find out which servers store the extent to be read. This method has the advantage of allowing maximum flexibility of data placement and visibility into the system's state. However, it has drawbacks: the metadata server is a central point of failure, usually implemented as a replicated state machine, that is on the critical path for all reads and writes.

In FDS, we took a different approach. FDS uses a metadata server, but its role during normal operations is simple and limited: collect a list of the system's active tractservers and distribute it to clients. We call this list the *tract locator table*, or TLT. In a single-replicated system, each TLT entry contains the address of a single tractserver. With k-way replication, each entry has k tractservers; see §3.3.

To read or write tract number i from a blob with GUID g, a client first selects an entry in the TLT by computing an index into it called the *tract locator*, designed to both be deterministic and produce uniform disk utilization:

$$\text{Tract\_Locator} = (\text{Hash}(g) + i) \bmod \text{TLT\_Length}$$

Hashing the GUID "randomizes" each blob's starting point in the table, ensuring clients better exploit the available parallelism whether or not the GUIDs themselves are assigned randomly. FDS uses SHA-1 for this hash.

Adding the tract number *outside* the hash ensures that large blobs use all entries in the TLT uniformly. An early (discarded) locator equation used $\text{Hash}(g+i)$. This effectively selected a TLT entry independently at random for each tract, producing a binomial rather than a uniform distribution of tracts on tractservers. As the number of tractservers increased, so did the occupancy deviation between the most-filled and least-filled disk. In the rejected design, writing 1 TB of 8 MB tracts to 1,000 tractservers was expected to write between 92 and 161 tracts to each ($\mu = 125$; $\sigma = 11.2$). The one tractserver with 29% more data than average was an unwanted straggler.

Once clients find the proper tractserver address in the TLT, they send read and write requests containing the blob GUID, tract number, and (in the case of writes) the data payload. Readers and writers rendezvous because tractserver lookup is deterministic: as long as a reader has the same TLT the writer had when it wrote a tract, a reader's TLT lookup will point to the same tractserver.

In a single-replicated system, the TLT is constructed by concatenating m random permutations of the tractserver list. Using only a single permutation can lead to unwanted client convoys. Sequential reads from a blob use TLT entries sequentially, so clients that bunch up in the queue of a slow tractserver will move in lockstep through the TLT, overloading some tractservers while many others are idle. Setting $m > 1$ ensures that after being delayed in a slow queue, clients will fan out to m other tractservers for their next operation. Our system uses $m = 20$, but we have not tested the system's sensitivity to this parameter.

In the case of non-uniform disk speeds, the TLT is weighted so that different tractservers appear in proportion to the measured speed of the disk.

To be clear, the TLT *does not* contain complete information about the location of individual tracts in the system. It is not modified by tract reads and writes. The only way to determine if a tract exists is to contact the tractserver that *would be* responsible for the tract if it does exist. Since the TLT changes only in response to cluster reconfiguration or failures it can be cached by clients for a long time. Its size in a single-replicated system is proportional to the number of tractservers in the system (hundreds, or thousands), not the number of tracts stored (millions or billions).

When the system is initialized, tractservers locally store their position in the TLT. This means the metadata server does not need to store durable state, simplifying its implementation. In case of a metadata server failure, the TLT is reconstructed by collecting the table assignments from each tractserver.

To summarize, our metadata scheme has a number of nice properties:

- The metadata server is in the critical path only when a client process starts. This is the key factor that allows us to practically keep tract sizes arbitrarily small. Systems such as GFS [14] require larger chunks partially to reduce load on the metadata server.
- The TLT can be cached long-term since it changes only on cluster configuration, not each read and write, eliminating all traffic to the metadata server in a running system under normal conditions.
- The metadata server stores metadata only about the hardware configuration, not about blobs. Since traffic to it is low, its implementation is simple and lightweight.
- Since the TLT contains random permutations of the list of tractservers, sequential reads and writes by independent clients are highly likely to utilize all tractservers uniformly and are unlikely to organize into synchronized convoys.

Our design is enabled by running on a full bisection bandwidth network. The locality-oblivious uniform access pattern would cause crippling congestion on a traditional network with hierarchical oversubscription.

2.3 Per-Blob Metadata

Each blob has metadata such as its length. FDS stores it in each blob's special metadata tract ("tract -1"). Clients find a blob's metadata on a tractserver using the same TLT used to find regular data. Distributed metadata is a particular advantage for atomic blob operations that require serialization to avoid inconsistency (e.g. `CreateBlob`, `DeleteBlob` and `ExtendBlobSize`). Even if thousands of clients are requesting atomic operations on blobs simultaneously, operations that can be parallelized (by virtue of referring to different blobs) are likely serviced in parallel by independent tractservers.

When a blob is created, the tractserver responsible for its metadata tract creates that tract on disk and initializes the blob's size to 0. When a blob is deleted, that tractserver deletes the metadata. A scrubber application scans each tractserver for orphaned tracts with no associated metadata, making them eligible for garbage collection.

Newly created blobs have a length of 0 tracts. Applications must *extend* a blob before writing past the end of it. The extend operation is atomic, is safe to execute concurrently with other clients, and returns the new size of the blob as a result of the client's call. A separate API tells the client the blob's current size. Extend operations for a blob are sent to the tractserver that owns that blob's metadata tract. The tractserver serializes it, atomically updates the metadata, and returns the new size to each caller. If all writers follow this pattern, the extend operation provides a range of tracts the caller may write without risk of conflict. Therefore, the extend API is functionally equivalent to the Google File System's "atomic append." Space is allocated lazily on tractservers, so tracts claimed but not used do not waste storage.

2.4 Dynamic Work Allocation

A result that flows naturally from FDS is that the assignment of work to worker can be done at very short timescales. This enables FDS to mitigate stragglers—a significant bottleneck in large systems because a task is not complete until its slowest worker is complete [5].

Hadoop- and MapReduce-style clusters that primarily process data locally are very sensitive to machines that are slow due to factors such as misbehaving hardware, jobs running concurrently, hotspots in the network, and non-uniformity in the input. If a node falls behind, there are not many options for recovery other than restarting its computation elsewhere [5]. The straggler period can also represent a great loss in efficiency if most resources are idle while waiting for a slow task to complete.

In FDS, since storage and compute are no longer colocated, the assignment of work to worker can be done dynamically, at fine granularity, *during* task execution. The best practice for FDS applications is to centrally (or, at large scale, hierarchically) give small units of work to each worker as it nears completion of its previous unit. This self-clocking system ensures that the maximum dispersion in completion times across the cluster is only the time required for the slowest worker to complete a single unit. Such a scheme is not practical in systems where the assignment of work to workers is fixed in advance by the requirement that data be resident at a particular worker before the job begins.

In many applications, the effect is significant. For example, in our sort application (§6.1), elimination of stragglers in the reading phase accounted for a 1/3 reduction in total job runtime.

3 Replication and Failure Recovery

Thus far, we have described FDS as single-replicated and thus not resilient to disk failures. To improve durability and availability, FDS supports higher levels of replication. When a disk fails, redundant copies of the lost data are used to restore the data to full replication.

The use of full bisection bandwidth networks means that, with appropriate data layout (§3.3), FDS can perform failure recovery dramatically faster than many other systems. In an *n*-disk cluster where one disk fails, roughly $1/n$th of the replicated data will be found on all *n* of the other disks. All remaining disks send the under-replicated data to each other *in parallel*, restoring the cluster to full replication very quickly. Performance is bounded only by the aggregate disk and network bandwidth. Thus, as the size of the cluster grows failure recovery gets *faster*. As we will see in §5.3, our cluster of about 1,000 disks recovers 92 GB of data lost from a single disk in only 6.2 s, and 655 GB lost from 7 disks on a failed machine in 33.7 s. Though the broad approach of FDS' failure recovery is similar to RAMCloud [26], RAMCloud recovers data to DRAM and uses replication only for fault tolerance. FDS uses replication both for availability and fault tolerance while recovering data *back to stable storage*. Such fast failure recovery significantly improves durability because it reduces the window of vulnerability during which additional failures can cause unrecoverable data loss.

3.1 Replication

As described in §2.2, each entry of the TLT in an *n*-way replicated cluster contains *n* tractservers. (Construction of such a TLT is described in §3.3.) When an application writes a tract, the client library finds the appropriate row of the TLT and sends the write to *every* tractserver it contains. Reads select a single tractserver at random. Applications are notified that their writes have completed only after the client library receives write acknowledgments from all replicas.

Replication also requires changes to `CreateBlob`, `ExtendBlobSize`, and `DeleteBlob`. Each mutates the state of the metadata tract and must guarantee that updates are serialized. Clients send these operations only to the tractserver acting as the primary replica, marked as such in the TLT. When a tractserver receives one of these operations, it executes a two-phase commit with the other replicas. The primary replica does not commit the change until all other replicas have completed successfully. Should the prepare fail, the operation is aborted.

FDS also supports per-blob *variable replication*, for example, to single-replicate intermediate computations for write performance, triple-replicate archival data for durability, and over-replicate popular blobs to increase read bandwidth. The maximum possible replication level is determined when the cluster is created and drives the number of tractservers listed in each TLT entry. Each blob's actual replication level is stored in the blob's metadata tract and retrieved when a blob is opened. For an *n*-way replicated blob, the client uses only the first *n* tractservers in each TLT entry.

3.2 Failure recovery

We begin with the simplest failure recovery case: the failure of a single tractserver. Later sections will describe concurrent tractserver failures, support for failure domains, and metadata server failures.

As described earlier, each row of the TLT lists several tractservers. However, each row also has a *version number*, canonically assigned by the metadata server. When a tractserver is assigned to a row and column of the TLT, it is also given the row's current version number.

Tractservers send heartbeat messages to the metadata server. When the metadata server detects a tractserver timeout, it declares the tractserver dead. Then, it:

- invalidates the current TLT by incrementing the version number of each row in which the failed tractserver appears;
- picks random tractservers to fill in the empty spaces in the TLT where the dead tractserver appeared;
- sends updated TLT assignments to every server affected by the changes; and
- waits for each tractserver to ack the new TLT assignments, and then begins to give out the new TLT to clients when queried for it.

When a tractserver receives an assignment of a new entry in the TLT, it contacts the other replicas and begins copying previously written tracts. When a failure occurs, clients must wait only for the TLT to be updated; operations can continue while re-replication is still in progress.

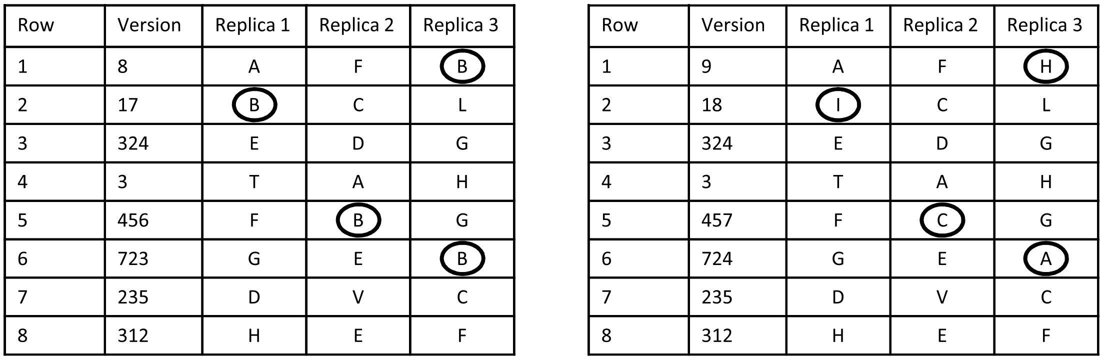
All operations are tagged with the client's TLT version. If a client attempts an operation using a stale TLT entry, the tractserver detects the inconsistency and rejects the operation. This prompts the client to retrieve an updated TLT from the metadata server. Client operations to tractservers not affected by the failure proceed as usual.

Table versioning prevents a tractserver that failed and then returned, e.g., due to a transient network outage, from continuing to interact with applications as soon as the client receives a new TLT. Further, any attempt by a failed tractserver to contact the metadata server will result in the metadata server ordering its termination.

After a tractserver failure, the TLT immediately converges to provide applications the current location to read or write data. This convergence property differentiates it from other hash-based approaches, such as those used within distributed hash tables, which may cause requests to be routed multiple times through the network before determining an up-to-date location for data.

3.2.1 Additional failure scenarios

Thus far, we have considered only the case where a single tract server fails. We now extend our description to concurrent and cascading tractserver failures as well as metadata server failures.



Row	Version	Replica 1	Replica 2	Replica 3
1	8	A	F	B
2	17	B	C	L
3	324	E	D	G
4	3	T	A	H
5	456	F	B	G
6	723	G	E	B
7	235	D	V	C
8	312	H	E	F

Row	Version	Replica 1	Replica 2	Replica 3
1	9	A	F	H
2	18	I	C	L
3	324	E	D	G
4	3	T	A	H
5	457	F	C	G
6	724	G	E	A
7	235	D	V	C
8	312	H	E	F

Figure 2: A Tract Locator Table before (*left*) and after (*right*) Disk B fails. B's appearances in the table are replaced with different disks and the version numbers of affected rows are incremented. All the disks in rows that contained B participate in failure recovery in parallel.

When multiple tractservers fail, the metadata server's only new task is to fill more slots in the TLT. Similarly, if failure recovery is underway and additional tractservers fail, the metadata server executes another round of the protocol by filling in empty table entries and incrementing their version. Data loss occurs when all the tractservers within a row fail within the recovery window.

Though not yet implemented, transient failures can be handled gracefully with *partial failure recovery*. A tractserver failure triggers replication of lost data as usual. If the tractserver later returns to service, the metadata server has two options: complete failure recovery as if the disk had never returned or use other replicas to recover the writes the returning tractserver missed while it was away. The metadata server will choose the option that requires copying less data. If it completes failure recovery, the returning tractserver is added to the empty disk pool. Otherwise, the tractserver resumes its positions in the TLT, failure recovery is halted, and copying of missed writes begins. Tractservers identify missed writes by examining the version of the TLT with which each tract was written. To further mitigate the effects of transient faults, the metadata server could separate the replacement of a failed server in the TLT from the initiation of the data movement required for failure recovery.

Network partitions complicate recovery from metadata server failures. A simple primary/backup scheme is not safe because two active metadata servers will cause cluster corruption. Our current solution is simple: only one metadata server is allowed to execute at a time. If it fails, an operator must ensure the old one is decommissioned and start a new one. We are experimenting with using Paxos leader election to safely make this process automatic and reduce the impact to availability. Once a new metadata server starts, tractservers contact it with their TLT assignments. After a timeout period, the new metadata server reconstructs the old TLT.

Tractservers that fail concurrently with a metadata server are detected by the new metadata server as missing TLT entries; the normal failure recovery protocol is executed. One strength our failure recovery protocol is its simplicity; all tractserver failure cases use the same protocol and exercise the same code path. This leads to small, simple, more obviously-correct code.

3.3 Replicated data layout

As mentioned previously, a k-way replicated system has k tractservers (disks) listed in each TLT entry. The selection of *which* k disks appear has an important impact on both durability and recovery speed.

Imagine first that we wish to double-replicate ($k = 2$) all data in a cluster with n disks. A simple TLT might have n rows with each row listing disks i and $i+1$. While data will be double-replicated, the cluster will not meet our goal of fast failure recovery: when a disk fails, its backup data is stored on only two other disks ($i+1$ and $i-1$). Recovery time will be limited by the bandwidth of just two disks. A cluster with 1 TB disks with 100 MB/s read performance would require 90 minutes for recovery. A second failure within that time would have roughly a $2/n$ chance of losing data permanently.[1]

A better TLT has $O(n^2)$ entries. Each possible *pair* of disks (ignoring failure domains; §3.3.1) appears in an entry of the TLT. Since the generation of tract locators is pseudo-random (§2.2), any data written to a disk will have high diversity in the location of its replica. When a disk fails, replicas of $1/n$th of its data resides on the other n disks in the cluster. When a disk fails, all n disks can exchange data in parallel over FDS' full bisection bandwidth network. Since all disks recover in parallel, larger clusters recover from disk loss more quickly.

While such a TLT recovers from single-disk failure quickly, a second failure while recovery is in progress is

[1] More precisely, $1-(\frac{n-1}{n})^2$

guaranteed to lose data. Since all pairs of disks appear as TLT entries, any pair of failures will lose the tracts whose TLT entry contained the pair of failed disks. Replicated FDS clusters therefore have a minimum replication level of 3. Perhaps counterintuitively, no level of replication ever needs a TLT larger than $O(n^2)$. For any replication level $k > 2$, FDS starts with the "all-pairs" TLT, then expands each entry with $k-2$ additional random disks (subject to failure domain constraints).

Constructing the replicated TLT this way has several important properties. First, performance during recovery still involves every disk in the cluster since every pair of disks is still represented in the TLT.

Second, a triple disk failure within the recovery window now has only about a $2/n$ chance[1] of causing permanent data loss. To understand why, imagine two disks fail. Find the entries in the TLT that contain those two disks. We expect to find 2 such entries. There is a $1/n$ chance that a third disk failure will match the random third disk in that TLT entry.

Finally, adding more replicas decreases the probability of data loss. Consider now a 4-way replicated cluster. Each entry in the $O(n^2)$-length TLT has *two* random disks added instead of one. 3 or fewer simultaneous failures are safe; 4 simultaneous failures have a $1/n^2$ chance of losing data. Similarly, 5-way replication means that 4 or fewer failures are safe and 5 simultaneous failures have a $1/n^3$ chance of loss.

One possible disadvantage to a TLT with $O(n^2)$ entries is its size. In our 1,000-disk cluster, the in-memory TLT is about 13 MB. However, on larger clusters, quadratic growth is cumbersome: 10,000 disks would require a 600 MB TLT.

We have two (unimplemented) strategies to mitigate TLT size. First, a tractserver can manage multiple disks; this reduces n by a factor of 5–10. Second, we can limit the number of disks that participate in failure recovery. An $O(n^2)$ TLT uses every disk for recovery, but 3,000 disks are expected to recover 1 TB in less than 20 s (§5.3). The marginal utility of involving more disks may be small. To build an n-disk cluster where m disks are involved in recovery, the TLT only needs $O(\frac{n}{m} \times m^2)$ entries. For 10,000 to 100,000 disks, this also reduces table size by a factor of 5–10. Using both optimizations, a 100,000 disk cluster's TLT would be a few dozen MB.

3.3.1 Failure domains

A *failure domain* is a set of machines that have a high probability of experiencing a correlated failure. Common failure domains include machines within a rack, since they will often share a single power source, or machines within a container, as they may share common cooling or power infrastructure.

FDS leaves it up to the administrator to define a failure domain policy for a cluster. Once that policy is defined, FDS adheres to that policy when constructing the tract locator table. FDS guarantees that none of the disks in a single row of the TLT share the same failure domain. This policy is also followed during failure recovery: when a disk is replaced, the new disk must be in a different failure domain than the other tractservers in that particular row.

3.4 Cluster growth

FDS supports the dynamic growth of a cluster through the addition of new disks and machines. For simplicity, we first consider cluster growth in the absence of failures.

Cluster growth adds both storage capacity and throughput to the system. The FDS metadata server rebalances the assignment of table entries so that both existing data and new workloads are uniformly distributed. When a tractserver is added to the cluster, TLT entries are taken away from existing tractservers and given to the new server. These assignments happen in two phases. First, the new tractserver is given the assignments but they are marked as "pending" and the TLT version for each entry is incremented. The new tractserver then begins copying data from other replicas. During this phase, clients write data to both the existing servers and the new server so that the new tractserver is kept up-to-date. Once the tractserver has completed copying old data, the metadata server 'commits' the TLT entry by incrementing its version and changing its assignment to the new tractserver. It also notifies the now replaced tractserver that it can safely garbage collect all tracts associated with that TLT entry.

If a new tractserver fails while its TLT entries are pending, the metadata server increments the TLT entry version and expunges it from the list of new tractservers. If an existing server fails, the failure recovery protocol executes. However, tractservers with pending TLT entries are not eligible to take over for failed servers as they are already busy copying data.

Variable replication complicates cluster growth because each replica will have different data depending upon the replication level of each blob written to it. Therefore, new tractservers must read from the existing tractserver whose TLT entry it is replacing. Should that server fail, the new tractserver may read from another tractserver "to the left" of its own entry, since it will have a superset of the data required. For example, the first replica in the TLT has all single-replicated tracts whereas the third replica has all triple-replicated tracts.

3.5 Consistency guarantees

The current protocol for replication depends upon the client to issue all writes to all replicas. This decision means that FDS provides weak consistency guarantees to clients. For example, if a client writes a tract to 1 of 3 replicas and then crashes, other clients reading different replicas of that tract will observe differing state. Weak consistency guarantees are not uncommon; for example, clients of the Google File System [14] must handle garbage entries in files. However, if strong consistency guarantees are desired, FDS could be modified to use chain replication [33] to provide strong consistency guarantees for all updates to individual tracts.

Tractservers may also be inconsistent during failure recovery. A tractserver recently assigned to a TLT entry will not have the same state as the entry's other replicas until data copying is complete. While in this state, tractservers reject read requests; clients use other replicas instead.

4 Networking

FDS' main goal is to expose all of a cluster's disk bandwidth to applications. FDS creates an uncongested path from disks to CPUs by:

- Giving each storage node *network bandwidth equal to its disk bandwidth*, preventing bottlenecks between the disk and network;
- Using a *full bisection bandwidth network*, preventing bottlenecks in the network core; and
- Giving compute nodes as much network bandwidth as they need I/O bandwidth, preventing bottlenecks at the client

Our FDS testbed uses a two-layer CLOS network [15, 16], which in its largest configuration consists of 8 "spine" routers and 14 "TORs" (Top-Of-Rack routers). Each router is a 64×10 Gbps Blade G8264. Each TOR has a 40 Gbps link (4 bonded 10 Gbps ports) to each spine router, giving it 320 Gbps total bandwidth to the spine layer. The other 320 Gbps of each TOR's bandwidth attaches to NICs. In total, this provides about 5.5 Tbps of bisection bandwidth for an infrastructure cost of about $250k. The routers are factory-standard, running the manufacturer's OS (BladeOS v6.8.4). We use BGP for route distribution with each TOR on its own IP subnet.

The TORs load-balance traffic to the spine using ECMP (equal-cost multipath routing), a standard router feature that selects a spine route for each TCP flow based on the hash of the TCP destination. This gives the network full bisection bandwidth without the need for global scheduling, and has an important advantage over round-robin route selection: it ensures packets within a flow are not reordered.

One drawback of ECMP is that full bisection bandwidth is not guaranteed, but only stochastically *likely* across multiple flows. Long-lived, high-bandwidth flows are known to be problematic with ECMP [3]. FDS, however, was designed to use a large number of short-lived (tract-sized) flows to a large, pseudo-random sequence of destinations. This was done in part to satisfy the stochastic requirement of ECMP.

Each computer in our cluster has a dual-port 10 Gbps NIC, primarily the Intel X520. One or both of these ports are connected to a TOR depending on the server's role (compute vs. storage) and the number of data disks it has. The NICs are configured with large-send offload, receive-side scaling, and 9 kB (jumbo) Ethernet frames. The TCP stack is configured with a reduced MinRTO to quickly recover from loss.

We have found it difficult to saturate a 10 G NIC using a single TCP flow: a single CPU core typically cannot keep up with a 10 Gbps NIC's interrupt load. Our operating system (Windows Server 2008 R2), in conjunction with the Intel NIC, uses RSS (Receive Side Scaling) to spread the interrupt load across cores. Similar to ECMP, RSS prevents in-flow packet reordering by hashing the TCP 4-tuple to select a core. As a result, multiple flows are needed to spread interrupt load. We need 5 flows per 10 Gbps port to reliably saturate the NIC. This is easily satisfied in FDS because its design dictates that many flows are active simultaneously.

At 20 Gbps, a zero-copy architecture is mandatory. FDS' data interfaces pass the zero-copy model all the way to the application. For clarity, Section 2.1 showed conventional one-copy versions of `WriteTract` and `ReadTract` interfaces; our applications actually use the preferred zero-copy versions. We also use buffer pools to avoid the large page fault penalty associated with frequent allocation of large buffers.

4.1 RTS/CTS

By design, at peak load, all FDS nodes simultaneously saturate their NICs with short, bursty flows. A disadvantage of short flows is that TCP's bandwidth allocation algorithms perform poorly. Under the high degree of fan-in seen during reads, high packet loss can occur as queues fill during bursts. The reaction of standard TCP to such losses can have a devastating effect on performance. This is sometimes called *incast* [34].

Schemes such as DCTCP [4] ameliorate incast in concert with routers' explicit congestion notification (ECN). However, because our network has full bisection bandwidth, collisions mostly occur *at the receiver*, not in the network core, providing us an opportunity to prevent them in the application layer. FDS does so with a request-to-send/clear-to-send (RTS/CTS) flow-scheduling system. Large sends are queued at the sender, and the receiver is notified with an RTS. The receiver limits the number of CTSs outstanding, thus limiting the number of senders competing for its receive bandwidth and colliding at its switch port.

RTS/CTS adds an RTT to each large message. This has the potential to reduce performance by preventing the network pipeline from ever filling. However, the FDS API encourages deep read-ahead and write-ahead, ensur-

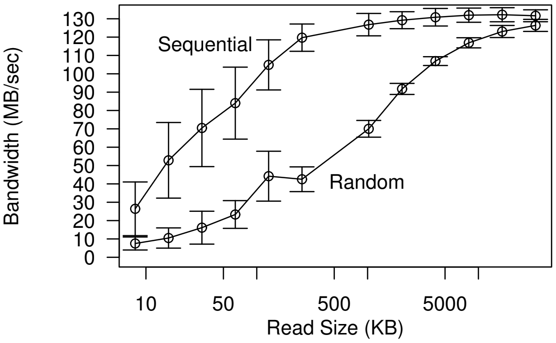


Figure 3: Performance of a single process reading to a single 10,000 RPM disk. Each point is the mean across 24 disks. Error bars show the standard deviation.

ing the FDS network library always has a large queue of future messages. This allows it to send an RTS for future messages in parallel with other data transfers, allowing the pipeline to fill.

Small and large messages are delivered using separate TCP flows, reducing the latency of control messages by enabling them to bypass long queues. FDS network message sizes are bimodal: large messages are almost all about 8 MB, and most others are a few kB or less.

5 Microbenchmarks

In this section and §6, we describe microbenchmarks and application benchmarks. All of our tests were conducted on a heterogeneous cluster of up to 256 HP, Dell and Silicon Mechanics servers acquired between 2008 and 2012. The machines had between 12 and 96 GB RAM, between 2 and 24 cores, and between 0 and 20 data disks (plus one OS disk). The disks were a combination of 147 GB and 300 GB 10,000 RPM 2.5" dual-port SAS drives; and 500 GB and 1 TB 7,200 RPM 3.5" SATA drives. Dynamic work allocation (§2.4) was key to efficiently using such a heterogeneous cluster. Since some machines were on loan to us during our quest to break the sort record (§6.1), some experiments used a subset of the cluster. In all but one (§6.3) of our experiments, computation and storage are located on different machines, which guarantees all experiments rely exclusively on remote access to storage. Unless otherwise noted, each test used an unreplicated cluster with CRC checks disabled. The network configuration is described in §4.

5.1 Raw Disk Performance

We start with a simple test: a single process writing to a single local disk. This benchmark establishes the baseline performance of our cluster's disks and shows how large random reads must be to amortize the cost of disk seeks. Like the tractserver, this benchmark uses the raw disk interface (that is, does not use NTFS). Results are shown in Figure 3. Each point is the median performance of 24 10,000 RPM SAS disks reading continuously using system calls ranging from 8KB to 32MB. Reads are sequential on the upper line, random on the lower line. Sequential performance peaks at about 131 MB/s. 8 MB random reads reach ≈117 MB/s, or ≈88% of sequential performance. Write performance, not shown, is similar to read performance.

5.2 Remote Reading and Writing

SimpleTestClient is the "hello world" FDS application. It uses the FDS client library to read and write blobs from and to tractservers. As in a real application, data are sent over the network and read from or written to disk. We tested its throughput and scalability by running successively larger numbers of SimpleTestClient instances concurrently against 1,033 tractservers. Each client instance had a single 10G NIC assigned to it. The tract size was 8 MB. For each configuration we adjusted the blob size so that the total read or write lasted between 5 and 10 minutes. We plot the number of clients against their aggregate throughput averaged over the entire experiment.

Figure 4a shows 1 to 180 clients reading and writing blobs sequentially against an unreplicated cluster. Two pairs of curves are plotted: a 516-disk and 1,033-disk cluster. At the left of the graph, the total tractserver bandwidth far exceeds the client bandwidth; performance is constrained by the number of clients. Throughput increases linearly at the rate of about 1,150 MB/s/client for writing and 950 MB/s/client for reading, roughly 90% and 74%, respectively, of the 10 Gbps interface. Reading tends to be slower than writing due to the difficulty of maintaining NIC saturation under fan-in (§4.1).

The right of the curves flatten as client bandwidth grows larger than tractserver bandwidth, which saturates them. Performance peaks at 32 GB/s in the 516-disk configuration and 67 GB/s in the 1,033-disk configuration, showing near-linear scalability. In both cases, FDS achieves remote reading and writing at roughly half the locally achievable disk throughput measured in §5.1. Though much better than many existing blob storage systems (see Table 2) there is room for improvement.

A similar test using *random* reads and writes is shown in Figure 4b. As expected, the performance is substantially the same as the sequential read and write tests. This validates our design goal that for 8MB tracts, random and sequential I/O deliver the same performance.

Figure 4c shows the bandwidth of sequentially reading and writing clients against a 1,033 disk *triple*-replicated cluster. As expected, the write bandwidth is about one-third of the read bandwidth since clients must send three copies of each write. Scaling properties are similar to that seen in Figures 4a and 4b.

Finally, to test the maximum speed achievable by a single client process, we tested a single instance of Sim-

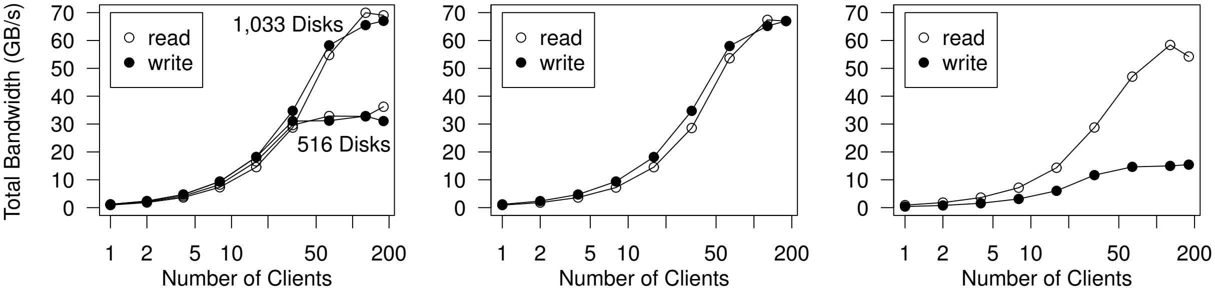


Figure 4: Mean aggregate throughput of 1 to 180 clients reading and writing 8 MB tracts on a 1,033-disk cluster. Standard deviation is less than 1% of the mean of each point. The *x* axes use logarithmic scales. *(a)* Sequential reading and writing in a single-replicated cluster. Results for a 516-disk cluster are also shown. *(b)* Random reading and writing in a single-replicated cluster. *(c)* Sequential reading and writing in a triple-replicated cluster.

Disk count	100		1,000		
Disks failed	1	1	1	1	7
Total (TB)	4.7	9.2	47	92	92
GB/disk	47	92	47	92	92
GB recov.	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 s run.

pleTestClient with 20 Gbps assigned to it, rather than 10 Gbps as in the previous tests. We wrote, then read, 10,000 tracts against a single-replicated cluster of 30 tractservers. Over 5 trials, SimpleTestClient achieved a mean (and standard deviation) write bandwidth of 2,187 ± 32 MB/s; for read, 2,045 ± 15 MB/s.

5.3 Failure Recovery

Table 1 shows the time taken to re-replicate lost data after one or more disk failures. Each experiment used a triple-replicated cluster of 100 or 1,000 146 GB, 10 K RPM SAS disks that contained enough data so that each tractserver held either 47 GB or 92 GB. We then killed a random tractserver process and measured the time until the cluster reported failure recovery was complete. Due to the random nature of the TLT, the exact amount of data recovered in each test varied slightly. We ran each test 5 times and we report the mean and standard deviation.

With 100 disks, FDS recovered 47 GB in 19.2 s and 92 GB in 50.5 s. Scaling the number of disks up by 10× to 1,000, recovery times improved by 6.8× to 3.3 s for 100 disks and by 8× to 6.2 s for 1,000 disks. Scaling is not quite linear due to the fixed cost of generating and distributing a new TLT.

We measured whole-machine failure by resetting a machine running 7 tractservers containing ≈655 GB of data. FDS recovered the lost data in 33.7 ± 1.5 s. Compared to single-disk failures, the whole-machine failure test recovered 7× the data in only 5× the time, as the fixed costs of recovery were amortized over more linear recovery time.

Recovery of 655 GB involves reading and writing a total of 1310 GB. Doing so in 33.7 s with 993 tractservers implies an average total read/write bandwidth of ≈40 MB/s/disk. Since the disks in these tests were nearly full, recovery wrote to the innermost, slowest disk tracks.

These results imply that a 1 TB disk in a 3,000 disk cluster could be recovered in ≈17 s. Such a small recovery window dramatically lessens the probability of data loss. Further, it lowers the impact of disk failures on availability and application performance. Though we are recovering to disk, our recovery time is comparable to the best known technique for recovering to RAM [26].

6 Applications

6.1 Sort

Sorting is an important primitive in many big-data applications. Its load pattern is similar to other common tasks such as distributed database joins and large matrix operations. This has made it an important benchmark since at least 1985 [9]. A group informally sponsored by SIGMOD curates an annual disk-to-disk sort performance competition with divisions for speed, cost efficiency, and energy efficiency [2]. Each has sub-divisions for general purpose applications (“Daytona”) and implementations that are allowed to exploit the specifics of the competition (“Indy”), such as assuming 100-byte records and uniformly distributed 10-byte keys.

In April 2012, our FDS-based sort application set the world record for sort in both the Indy and Daytona categories of MinuteSort [7], which measures the amount of data that can be sorted in 60 seconds. Using a cluster of 1,033 disks and 256 computers (136 for tractservers, 120 for the application), our Daytona-class app sorted

System	Computers	Data Disks	Sort Size	Time	Implied Disk Throughput
MinuteSort—Daytona class (general purpose)					
FDS, 2012	256	1,033	1,401 GB	59 s	46 MB/s
Yahoo!, Hadoop, 2009 [25]	1,408	5,632	500 GB	59 s	3 MB/s
Yahoo!, Hadoop, 2009 [25] (unofficial 1 TB run)	1,408	5,632	1,000 GB	62 s	5.7 MB/s
MinuteSort—Indy class (benchmark-specific optimizations allowed)					
FDS, 2012	256	1,033	1,470 GB	59.4 s	47.9 MB/s
UCSD TritonSort, 2011 [27]	66	1,056	1,353 GB	59.2 s	43.3 MB/s

Table 2: Comparison of FDS MinuteSort results with the previously standing records. In accordance with sort benchmark rules, all reported times are the median of 15 runs and 1 GB = 10^9 bytes.

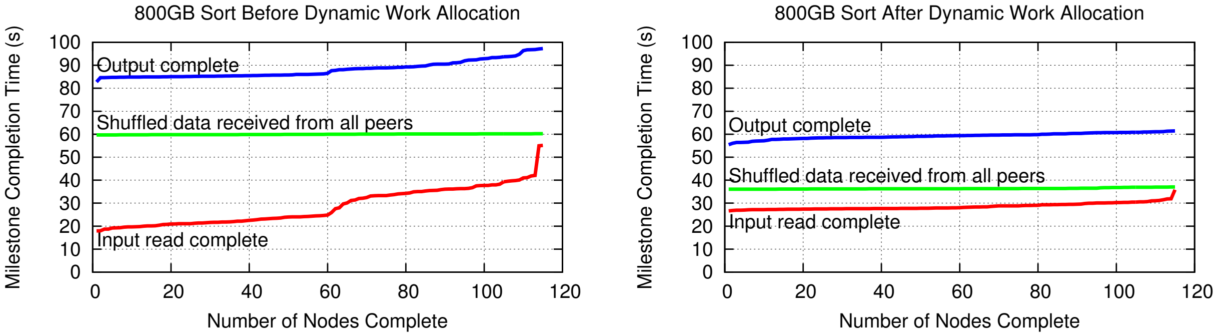


Figure 5: Visualization of the time to reach three milestones in the completion of a sort. The results are shown before (*left*) and after (*right*) implementation of dynamic work allocation. Both experiments depict 115 nodes sorting 800 GB.

1,401 GB[2] in 59.4 s. This bested the standing Daytona record by a factor of 2.8x while using about 1/5 as many CPUs and disks. Our Indy-class app sorted 1,470 GB in 59.0 s, breaking the standing record by about 8%. Our Indy sort was identical to Daytona except for assuming a uniform key distribution instead of sampling the input. Our results and comparisons to the previous recordholders are shown in Table 2. The sort application consists of one head process and *n* worker processes. The input is given in a single FDS blob, from which each worker reads a separate subset of tracts. The sort occurs in three phases: input sampling, reading, and writing.

In the input sampling phase, the head process reads 1.5 million records: 6,000 from each of 256 tracts selected randomly from the input blob. It computes the key distribution and hence the assignment of key ranges to buckets. It then unicasts the computed distribution to all the other sort processes. In the Indy sort, this phase is skipped; the key distribution is assumed to be uniform and the bucket partitions are pre-computed.

In the read phase, each sort process performs three tasks simultaneously. First, each sort process reads tracts from its assigned region of the input. Simultaneously, as each tract arrives (in arbitrary order), the sort process shuffles the tract's records into output bins according to the bucket partitions received after the sampling phase. As each bin fills, the process sends the bin to the appropriate "bucket-receiver," a peer sort process. The buckets form an ordered partition of the keyspace. Finally, it receives bins from peer sort nodes.

In the write phase, each sort process sorts its bucket and writes it to a separate output blob. The sorted result is distributed among *n* blobs.

Sort relied heavily on dynamic work allocation (§2.4). At first, all workers were responsible for reading equal portions of the input; 1/3 of the total sort time was lost to stragglers, partially because of our cluster's heterogeneity. In later versions, the head node coordinated work assignments: nodes requested an input range as they neared completion of their previously assigned range.

Figure 5 shows time diagrams for a sort experiment before and after dynamic work allocation was implemented. (No other changes were made between the two experiments shown.) Each graph has three lines that depict the time at which each worker reached three milestones: completing the read phase, starting the write

[2] In this section only, we use the sort benchmark's definition of 1 GB = 10^9 bytes.

phase, and completing the write phase. Horizontal lines would be ideal, indicating every node reached a milestone simultaneously. There is a global barrier between the read and write phases; stragglers in the read phase cause significant performance loss as most nodes wait idly for the last readers to complete. Dynamic work allocation, enabled by FDS' freeing nodes from data locality, significantly improved overall sort time by reducing the time nodes were waiting at the barrier.

FDS is the first system in the history of the sort benchmark to set the record using general-purpose remote storage exclusively, without exploiting locality at all. Parallel sorts were previously accomplished by reading input data off a local disk, bucketing over the network, then writing to a local disk. In FDS, all storage is remote: our sort reads data from remote tractservers, buckets, then writes back to tractservers, sending data over the network three times. This demonstrates that FDS achieves world-class performance while still exposing a simple single-disk model that frees applications from the complexity associated with reasoning about locality.

In 2011, TritonSort [27] set the MinuteSort record with code carefully optimized for sorting. The FDS sort achieved similar per-disk bandwidth despite being built on top of a general blob store, accessing disks remotely. However, the FDS result did use approximately 4× as many computers. Part of this difference is superficial: tractservers were run on older machines, recycled from other projects; they did not have sufficient CPU or memory to act as sort clients. A single modern machine could act as both tractserver and client whereas our cluster required two. However, some of the difference is fundamental; it demonstrates the inefficiency of records traversing the network three times rather than once.

6.2 Cointegration

A colleague had an application that performs cointegration (a statistical technique) on stock market data to find correlations in price fluctuations. The raw input is a time-series of all stock trades. The application's first phase reads the time-series data and, for each ticker symbol it finds, generates an "order book," a list of trades for that stock on that day. The second phase compares pairs of order books to find correlations.

This application was originally implemented on a publicly available cloud-computing provider. Measurements showed it was I/O-bound, so he ported it to FDS. The public cloud implementation used one single-core VM for compute and the cloud service's blob store for storage. The FDS version used one 8-core machine for compute and 98 tractservers in an older version of our cluster that used 9 1 Gbps Ethernet interfaces per machine. Accounting for differences in hardware, the FDS version was at least 6× faster, as shown in Table 3.

		Time	Speedup	Speedup
System	Cores	(sec)	Per Node	Per Core
Public cloud	1	1,200	1	1
FDS	8	24	50x	6.25x

Table 3: Comparison of FDS to a public cloud for reading a single day of stock market data and generating one order book per stock symbol found.

A 6× speedup might actually under-state the improvement from FDS. On the cloud computing system, the VM's single core was underutilized. A multi-core VM would have been unlikely to significantly improve the speed since the bottleneck was I/O.

An interesting performance bottleneck emerged. Because this task had always been I/O-bound, the input was stored after zlib compression and decompressed on the fly. This effectively increased I/O throughput by the compression factor of 7×. In FDS, input tracts arrived so quickly that decompression was the bottleneck. 8 MB compressed tracts were, on average, arriving every 70 ms per 1 Gbps NIC. Zlib decompression took 218 ms, implying 3 cores were needed per NIC, but the machine had only 8 cores. Configuring zlib to favor speed rather than compression reduced decompression time to only 188 ms. Switching to a compression library optimized for decompression speed (XPress), compression ratio was reduced to 3:1 but tract decompression required only 62 ms. FDS transformed cointegration from an I/O-bound task to a compute-bound task.

6.3 Serving an index of the web

Search portals such as Bing use an index of the web to provide answers to search queries. In 2011, Bing was evaluating alternative architectures for serving the tail (i.e., infrequent) queries from the index. We ported one system to use FDS and measured its performance.

The tail index serving pipeline is composed, roughly, of two stages. First, for each term found in a user's search query, a list of the documents that contain it are retrieved from disk. The document list comes annotated with a static rank of each document's relevance with respect to that single search term. Second, these document lists are merged, ranked, and sent to a secondary ranker which computes each document's relevance to the total query based on document contents and other information. These document lists were "term-sharded," meaning each machine in the cluster was responsible for a subset of the terms found in all crawled web documents. Each machine spread its document lists across four local disks. The code had been carefully optimized.

Our effort to use FDS focused initially on a feasibility experiment that perturbed as little as possible in the index serving pipeline. Each rack of machines was converted into an FDS cluster. The index serving pipeline, also running on the same machines, was changed to use FDS as its "local" store instead of local physical disks. This had the effect of striping all shards' data across all disks in the rack.

We ran in-house performance regression tests that issued thousands of queries to the ranker pipeline and measured the number of queries per second (QPS) served while keeping 95% of service times below the maximum specified by the SLA. We measured performance using two configurations. The first consisted of an unmodified index serving pipeline on 40 machines, each of which had 4 disks and a 10 Gb/s network connection to a single switch. The second configuration used only 12 machines but ran using an FDS cluster with 48 tractservers.

The FDS version showed a dramatic QPS improvement of 2× despite using 1/3 the machines. While the median latency was essentially the same in both versions, the FDS version reduced the 95th percentile latency by 2.4×. We attribute the difference to better statistical multiplexing of disks. Queries require document lists that vary in size by several orders of magnitude. Using local disks, long reads delay other queries behind them. FDS, in contrast, was better at spreading document lists more uniformly across all available disks, making it less likely that single machine would fall behind.

The index serving pipeline proved to be a good match for our weak consistency model. While serving the index, the document lists are read-only. When the index is updated, each term's blob is written by a single writer, and the front-end does not try to read the new blob until the update is complete.

7 Related Work

We believe that FDS is the first high performance blob storage system designed for datacenter scale with a flat storage model.

The Google File System has a centralized master that keeps all metadata in memory [14]. This approach is limiting because as the contents of the store grow, the metadata server becomes a centralized scaling and performance bottleneck. In a recent interview, Google architects described the GFS metadata server as a limiting factor in terms of scale and performance [22]. Additionally, a desire to reduce the size of a chunk from 64 MB was limited by the proportional increase in the number of chunks in the system.

FDS uses deterministic placement to eliminate the scaling limitations of current blob store metadata servers. The tract locator table's size is determined by the number of machines in a cluster, rather than the size of its contents. Further, the TLT enables FDS to use an unlimited number of small, 8 MB tracts to stripe data across the cluster. Finally, FDS fully distributes the blob metadata into the cluster using metadata tracts, which further minimizes the need to centrally administer the store.

xFS [6] also proposed distributing file system metadata among storage nodes through distributed metadata managers. A replicated manager map determined which manager was responsible for the location of a particular file. FDS distributes its metadata among storage nodes through the metadata tract and uses deterministic placement to eliminate the need for an additional manager service to respond to requests for the location of individual files within the storage system.

DHTs like Chord [31] use techniques such as consistent hashing [20] to eliminating the need for centralized coordination when locating data. However, under churn, a request within a DHT might be routed to several different servers before finding an up-to-date location for data. FDS also uses hashing to implement deterministic placement, but the TLT directs clients to data without ambiguity. Failures spur the generation of a new table, providing an up-to-date location to clients. FDS takes advantage of deterministic placement to minimize the load on the FDS metadata server while relying on a small amount of state to ensure that the location of data is determined within a single network hop.

RAMCloud [26] provides fast recovery from failures by distributing data, in the form of log segments, across many machines and recovering those segments in parallel. However, RAMCloud recovers all data to main memory, while FDS implements fast failure recovery back to stable storage. Further, RAMCloud distributes data to disk purely for reasons of fault tolerance, while FDS replication is used both for fault tolerance and availability. Panasas [35] uses RAID 5 to stripe files, rather than blocks, across many servers to accelerate RAID recovery, while FDS ensures that all disks participate in recovery by striping tracts among all disks in the cluster.

Distributed file systems like Frangiapani [32], GPFS [28], AFS [18] and NFS [29] export a remote storage model across a shared, hierarchical name-space. These systems must contend with strong consistency guarantees, and the vagaries of remote, shared access to a POSIX compliant API. By focusing only on fast access to blob storage, FDS provides weak consistency guarantees with very high performance.

Among many others, systems such as Swift [11], Zebra [17], GPFS [28], Panasas [35], and Petal [21] stripe files, blocks, or logs across file servers to improve read and write throughput for traditional hierarchical file systems. FDS follows in the footsteps of these systems by using the tract locator table to guarantee a uniform distribution of disk accesses regardless of the pattern of requests issued by applications.

TritonSort [27] demonstrated the power of a balanced approach to storage and computation by breaking several of the world records in sorting. FDS demonstrates that this approach can be extended to a general purpose, locality oblivious system to break the sorting record without loss of performance.

PortLand [24] and VL2 [15] make it economically feasible to build datacenter-scale full bisection bandwidth networks. Other full bisection networks exist, such as Infiniband [23], but at a cost and scale that limited them to supercomputing and HPC environments.

Finally, River [8] used a distributed queue to dynamically adjust the assignment of work to applications nodes at run time in data flow computations. Similarly, FDS applications use dynamic work allocation to choose which node will consume a tract of data at runtime to adjust for performance faults and the varying performance capabilities within a heterogeneous cluster.

8 Conclusion

We have presented Flat Datacenter Storage, a datacenter-scale blob storage system that exposes the full bandwidth of its disks to all processors uniformly. It largely obviates the need for locality without sacrificing performance. Individual processes can read and write at near their NIC's rate—2 GBps or more. Aggregate client bandwidth scales nearly linearly and reaches 50% of the theoretical bandwidth of the underlying disks. Bandwidth capacity scales nearly linearly as disks are added.

This has a number of important consequences. First, recovery from failed disks can be done in seconds rather than hours; we recover 655 GB in a 1,000 disk cluster in only 33.5 s. Small recovery windows increase durability by decreasing the likelihood of complete data loss.

Second, FDS has implications for the structure of software. By exposing a cluster's full I/O bandwidth without locality constraints, FDS deconflates high I/O performance from data-parallel programming models such as MapReduce. Programmers can pick the most natural model for expressing computation without sacrificing performance.

Third, FDS has implications for the way clusters are built. Today's big-data clusters are often built with "one size fits all" machines that assume all applications have a similar balance of CPU to disk bandwidth requirements. With FDS, I/O and compute resources can be purchased separately, each independently upgradable depending on which resource is in shortage.

Finally, systems like FDS may pave the way for new kinds of applications. Large matrix operations, sorts, distributed joins, and all-to-all comparisons were largely off-limits to programmers working at datacenter scales: if it couldn't be done within a rack, it couldn't be done quickly. FDS makes these applications practical—potentially even enabling new kinds of science.

9 Acknowledgments

We thank our shepherd, Steve Gribble, and the anonymous reviewers for helpful comments and feedback. John Douceur, Jason Flinn and Eddie Kohler also provided insightful comments on early drafts. We are indebted to Barry Bond, who ported the cointegration application to FDS and has provided extensive guidance on performance-tuning Windows Server. 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. Li Lu helped to integrate FDS with the Bing index serving pipeline. 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] Apache Hadoop. `http://hadoop.apache.org`.

[2] MinuteSort Benchmark. `http://sortbenchmark.org`.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, pages 281–296. USENIX Association, 2010.

[4] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In S. Kalyanaraman, V. N. Padmanabhan, K. K. Ramakrishnan, R. Shorey, and G. M. Voelker, editors, *SIGCOMM*, pages 63–74. ACM, 2010.

[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '09)*, October 2010.

[6] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Trans. Comput. Syst.*, 14(1):41–79, Feb. 1996.

[7] J. Apacible, R. Draves, J. Elson, J. Fan, O. Hofmann, J. Howell, E. Nightingale, R. Olinksy, and Y. Suzue. MinuteSort with Flat Datacenter Storage. Technical report, Microsoft Research, http://sortbenchmark.org/FlatDatacenterStorage2012.pdf, 2012.

[8] R. H. Arpaci-Dusseau. Run-time adaptation in River. *ACM Trans. Comput. Syst.*, 21(1):36–86, Feb. 2003.

[9] 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*, 31(7):112–118, Apr. 1985.

[10] D. Borthakur. The Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[11] L.-F. Cabrera and D. D. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.

[12] Cisco Systems. Data center: Load balancing Data Center Services, 2004.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *The 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 137–150, December 2004.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, New York, NY, USA, 2003. ACM.

[15] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. *Commun. ACM*, 54(3):95–104, Mar. 2011.

[16] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *Proceedings of the ACM workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 57–62, New York, NY, USA, 2008. ACM.

[17] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 29–43, New York, NY, USA, 1993. ACM.

[18] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 Eurosys Conference*, pages 59–72, 2007.

[20] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31:1203–1213, May 1999.

[21] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural support for Programming Languages and Operating Systems*, volume 31, pages 84–92, New York, NY, USA, Sept. 1996. ACM.

[22] M. K. McKusick and S. Quinlan. GFS: Evolution on fast-forward. *acmqueue*, 7(7), August 2009.

[23] Mellanox. Building a Scalable Storage with InfiniBand, 2012.

[24] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM.

[25] O. O'Malley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. `http://sortbenchmark.org/Yahoo2009.pdf`, 2009.

[26] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 29–41, New York, NY, USA, 2011. ACM.

[27] A. Rasmussen, G. Porter, M. Conley, H. M. andRadhika Niranjan Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced large-scale sorting system. In *the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[28] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *the Conference on File and Storage Technologies (FAST '02)*, January 2002.

[29] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol. RFC 3530, Apr. 2003.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.

[31] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[32] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, New York, NY, USA, 1997. ACM.

[33] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *The 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 91–104, 2004.

[34] V. Vasudevan, H. Shah, A. Phanishayee, E. Krevat, D. Andersen, G. Ganger, and G. Gibson. Solving TCP incast in cluster storage systems. In *The 7th USENIX Conference on File and Storage Technologies (FAST '09)*, San Francisco, CA, February 2009.

[35] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small1, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *The 6th USENIX Conference on File and Storage Technologies (FAST '08)*, 2008.