MinuteSort with Flat Datacenter Storage

Johnson Apacible, Rich Draves, Jeremy Elson, Jinliang Fan,
Owen Hofmann, Jon Howell, Ed Nightingale, Reuben Olinsky, Yutaka Suzue
Microsoft Research

May 15, 2012

1 Overview

We have built a new high-performance distributed blob storage system, called *Flat Datacenter Storage* (FDS). Our MinuteSort entry is a relatively simple Daytona-class sort application that uses FDS for storage. We also have an Indy mode that is identical to the Daytona version except that input sampling is disabled and a uniform key distribution is assumed.

- In Indy mode, FDS sorted **1,470 GB**[1] in 59.4 s.[2,3]
- In Daytona mode, FDS sorted **1,401 GB** in 59.0 s.

The sorts were accomplished using a heterogeneous cluster consisting of 256 computers and 1,033 disks, divided broadly into two classes: storage nodes and compute nodes. Notably, **no compute node in our system uses local storage for data**; we believe FDS is the first system with competitive sort performance that uses *remote storage*. Because files are all remote, our 1,470 GB runs actually transmitted 4.4 TB over the network in under a minute. No strong assumptions are made around key or record lengths; keys and records of other lengths can be handled with only a performance-neutral configuration change.[4]

Our cluster's computers are a mix of HP and Dell systems with a range of RAM (24GB to 96GB) and CPUs (2 to 12 cores). The compute nodes had a single local disk, used only for operating system files. Storage nodes had one operating system disk plus between 5 and 16 FDS data disks used for sort data. These disks had a range of capabilities: primarily (78%) 10K RPM dual-port SAS disks, but with some (22%) 7,200 RPM SATA disks. Each computer is networked using 10G (10 gigabit/sec) Ethernet connected in a full-bisection-bandwidth CLOS network [5]. The interconnect is 22 × Blade G8264 switches, each of which has 64×10G Ethernet ports. Most computers had a dual-port 10G NIC (an Intel X520 or HP NC522SFP), though some of the storage nodes used only a single port. All computers run Windows Server 2008 R2 SP1.

The entire input file for each sort was stored in FDS as a single logical blob. The output was partitioned across 256 blobs. In Daytona mode, bucket boundaries were computed dynamically by scanning approximately 1.5 million keys from 256 locations selected uniformly at random across the input file.

Our results are the median of 15 runs. Each run consisted of a setup phase, during which `gensort` was used to create an appropriately-sized blob; the sort phase; and a validation phase, in which `valsort` checked the sorted results. Since `gensort` and `valsort` were not widely parallelized, the 15-run experiment took several hours. (Our system met the Daytona requirement that the system run for more than one hour without failure.)

In the remainder of this paper, we will describe FDS (Section 2), the network it runs on (Section 2.3), the sort application built on top of it (Section 3), and more details of our cluster's hardware configuration (Section 4). FDS itself has no sort-specific optimizations in it. Our sort application uses the same simple API into FDS that any other application might use; other applications can be expected to achieve similar performance.

2 Flat Datacenter Storage

Flat Datacenter Storage (FDS) is a new blob storage system developed at Microsoft Research with performance and scalability as its primary goals. In FDS, *all storage is remote*. That is, data is stored on dedicated storage nodes, called *tractservers*. Logically, a tractserver is a network front-end to a single disk; machines with multiple disks have one tractserver running per disk. User code does not run on tractservers; applications can only retrieve data from or write data to tractservers over the network. In FDS, there is no such thing as a local file (other than each system's operating system disk). This organization is in sharp contrast to existing systems such as MapReduce [3], Hadoop [1], and Cosmos [2], all of

[1] As is standard for MinuteSort, 1 GB = 10^9 bytes.

[2] FDS includes a job control service that copies binaries out to the cluster and executes processes. In conformance with the sort benchmark rules, our reported times were measured by this service, not by the sort process itself.

[3] Times reported are the median of 15 executions; details in Section 4.

[4] Variable length keys and records are also easily supported, though we have not implemented such support.

which are built from first principles to do processing on data stored locally—sending the computation to the data as a way to avoid sending the data over the network. FDS *always* sends data over the network.

FDS mitigates the cost of data transport in two ways. First, we give each storage node network bandwidth that matches its storage bandwidth. SAS disks have read performance of about 120MByte/sec, or about 1 gigabit/sec, so in our FDS cluster a storage node is always provisioned with at least as many gigabits of network bandwidth as it has disks. Second, we connect the storage nodes to compute nodes using a full bisection bandwidth network—specifically, a CLOS network topology, as used in projects such as Monsoon [5].

The combination of these two factors produces an uncongested path from remote disks to CPUs, giving the system an aggregate I/O bandwidth essentially equivalent to a system such as MapReduce that uses local storage. There is, of course, a latency cost. However, FDS by its nature allows *any compute node to access any data with equal throughput.* MapReduce, in contrast, places a significant burden on developers, forcing them to reason about locality, and imposing a high cost for off-rack data transport. Sorting inherently breaks data parallelism; no initial data placement obviates the need for data movement. This makes it a poor match for MapReduce- and Hadoop-class systems and explains the large efficiency gap between FDS MinuteSort performance and that of the current Daytona MinuteSort record-holder: Yahoo!'s Hadoop system sorted 500GB in 59 seconds, and 1,000GB in 62 seconds, using 1,406 machines and 5,624 disks.

FDS supports data replication for failure recovery. However, to expose maximum disk bandwidth for the sort benchmark, we used an unreplicated cluster: both the original input and final output were single-replicated.

2.1 The FDS API

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, limited in size only by 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. Tracts in FDS are 8MB (2^{23} bytes). This size was selected, in part, because it amortizes the cost of disk seek over enough data to make the seek cost negligible. With disks we tested, reading and writing 8MB units in random locations on the disk achieved similar performance to sequential read and write performance in the same areas of the disk. (Regardless of access pattern, outer disk tracks exhibit better performance than inner tracks due to the greater linear speed of the disk platter past the head.)

The FDS API is relatively narrow and straightforward. It primarily consists of:

- `CreateBlob(UINT128 blobGuid)` (returns a blob handle)
- `OpenBlob(UINT128 blobGuid)` (returns a blob handle)
- `CloseBlob(UINT128 blobGuid)`
- `DeleteBlob(UINT128 blobGuid)`

Blob handles have the following members:

- `GetBlobSize()`
- `ExtendBlobSize(UINT64 numberOfTracts)`[5]
- `WriteTract(UINT64 tractNumber, BYTE *buf, UINT32 length)`
- `ReadTract(UINT64 tractNumber, BYTE *buf, UINT32 length)`
- `GetSimultaneousLimit()`

These are simplified versions of the actual function prototypes; parameters and return values related to error handling have been elided. In addition, each function takes a callback function (and associated context pointer). All calls in FDS, including `ReadTract` and `WriteTract`, are non-blocking. The FDS client-side library invokes the application's callback when the operation completes. The callback functions are required to be re-entrant, as each call is performed on its own thread and callback calls may overlap—that is, Tract 2 may arrive before the user's callback has finished processing Tract 1. Tracts reads are not guaranteed to arrive in order of issue.

Such a non-blocking API is required to achieve good performance. A hypothetical blocking version of ReadTract would encourage applications to issue requests serially, leaving a node's link mostly idle. By spreading a blob's tracts over many tractservers (as described in Section 2.2) and issuing many requests in parallel, many tractservers can begin reading data off disk and transferring it back to a processing node in parallel. In addition, deep read-ahead allows a tract to be read off disk into the tractserver's cache while the previous one is being transferred over the network. The FDS API `GetSimultaneousLimit()` tells the application how many reads and writes it should have outstanding at once. Typical FDS applications initialize a

[5] Blob metadata specifies precise file length in byte increments.

semaphore with this value, `down` the semaphore before issuing each read (e.g., in a loop over every tract in a blob), and `up` it in the read completion callback.

2.2 Data Placement

A key issue in parallel storage systems is data placement and rendezvous, that is: how does a writer decide which server to use when writing data? How does a reader find data that has been previously written by a writer?

Many existing systems solve this problem centrally using a metadata server. 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 on the critical path for all reads and writes, making it a centralized bottleneck. Since it is also central point of failure, such metadata servers are typically implemented using a replicated state machine using a consensus protocol.

In FDS, we took a different approach. While we do have a metadata server, its role during normal operations is simple and limited: collect a list of the system's active tractservers and distribute information about them to clients.[6] We call this list the *tract locator table*, or TLT. For clarity of explanation, assume for now that the TLT is simply a list of the tractservers in the system. The order is random, but consistent; that is, the metadata server randomizes the server list once at cluster initialization, then gives out identical lists to every client.

When a client starts, it first retrieves the TLT from the metadata server. When it wants to read or write a tract, it first computes a *tract locator*. The simplest tract locator is the sum of the 128-bit blob GUID to be read and the 64-bit tract number to be read, modulo the number of entries in the TLT. Indexing the tract locator into the TLT yields the tractserver to which that tract read or write should be issued. The read and write requests contain the full blob GUID and tract number and (in the case of writes) the data payload. Readers can find the data written by earlier writers because the process of finding a tractserver is deterministic: as long as they have the same TLT as the writer when it wrote the tract, a reader's TLT lookup will point to the same tractserver as the writer's did.

[6]The FDS metadata server is also involved in failure recovery when a tractserver fails. Replication and failure recovery in FDS are beyond the scope of this paper.

Note that the TLT *does not* contain complete information about the location of individual tracts in the system. It only contains information about *classes* of tracts. Since it changes only in response to cluster reconfiguration—not individual reads and writes—it can be cached by clients for a long time.[7] Its size (in the simple implementation described so far) is proportional to the number of tractservers in the system (hundreds, or thousands), not the number of tracts stored (millions or billions). In addition, because the tractservers remember their position in the table, the metadata *stores no durable state*; in case of a metadata server failure, the TLT is reconstructed by contacting each tractserver.[8] The TLT is not modified by reads and writes; the only way to determine if a tract exists is to contact the tractserver responsible for it.

Per-blob metadata, such as blob length and permissions, are stored in a special tract ("tract -1") of each blob. Clients find a blob's metadata using the same method for finding data, using the TLT. Thus per-blob metadata management is as distributed as blob data storage. Distributed metadata is a particular advantage for atomic blob operations that require serialization to avoid inconsistency (e.g. `CreateBlob`, `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 to be serviced in parallel by independent tractservers.

2.2.1 Avoiding client convoys

One disadvantage of the simple scheme as described is that there are likely to be unwanted correlations introduced when multiple clients are running in parallel. For example, consider 100 clients, each of which begins a long series of sequential writes to a blob whose GUID is simply the node's ID (from 0 to 99). The FDS simultaneous read limit might be 50. In this case, even if the FDS cluster has thousands of tractservers, the 100 clients will end up using only 150 of those servers in lockstep, despite running on a system that has sufficient capacity for dozens of tractservers to be serving each client. For this reason, the actual tract locator we use is the tract number plus the SHA-1 hash of the blob's GUID. Hashing the GUID has the effect of deterministically "randomizing" each blob's starting point in the table, ensuring clients better exploit the available parallelism.

While a TLT could be a simple permutation of the server list, the actual algorithm we use in FDS is slightly

[7]In the event of disk failure, the cache is invalidated; details are beyond the scope of this paper.

[8]Details of FDS' method for recovering from metadata server failure are beyond the scope of this paper.

more complex. A simple permutation would still cause unwanted correlations, e.g., every client would write to tractserver 54 only after writing to tractserver 191. If one tractserver falls behind, readers will bunch up in its queue and become synchronized. To prevent this, our TLT is actually a concatenation of 20 independent permutations of the server list. It is important to concatenate independent permutations rather than shuffle 20 copies of the list to ensure that a writer that stops writing midway through the list will use each tractserver for the same number of tract writes. Uniform utilization is a key strategy to prevent stragglers in applications that do long sequential reads and writes.

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.

Bringing all these optimizations together, our metadata scheme has a number of nice properties:

- The metadata server is only in the critical path when a client process starts. This is the key factor that allows us to practically keep tract sizes as low as 8MB. (Systems such as GFS [4] require larger extents partially to reduce load on the metadata server.)
- The TLT can be cached long-term since it only changes 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 only metadata about the hardware configuration, not about files. Since it is not used heavily, the metadata server can be both lightweight and simple.
- Since TLT contains random permutations of the list of tractservers, sequential reads and writes by independent clients are highly likely to utilize all tractservers uniformly.
- The TLTs independent permutations prevent clients from organizing into synchronized convoys.

2.3 Network

Each computer is networked using two 10 Gbps Ethernet ports interconnected with a full bisection bandwidth CLOS network [5].

The network has two layers, "spine" routers and "TORs" (Top-Of-Rack routers), each of which is a 64×10Gbps Blade G8264 switch.

Our network has eight spines and fourteen TORs. Each TOR has a 40Gbps connection to each spine, giving it 320 Gbps total bandwidth to the spine. The other 320 Gbps of TOR bandwidth attach to NICs.

The TORs load-balance traffic to the spine using ECMP (equal-cost multipath routing), selecting a spine route for each TCP flow based on the hash of the TCP destination. This gives the network a full bisection bandwidth without the need for global scheduling, but the guarantee is only stochastic across multiple flows. FDS nodes, by design, send a large number of short flows to a broad set of destinations; this satisfies the stochastic requirement, balancing traffic from a TOR to the spine across the eight spine switches. The switches are running the manufacturer's OS (BladeOS v6.8.4); FDS' only requirement is topology.

The NICs are configured with large-send offload, receive-side scaling, and 9K (jumbo) Ethernet frames. The host OS is configured with a reduced MinRTO [6] to quickly recover from loss.

By design, at peak load, all the FDS nodes send at the same time. Because the flows are short and bursty, TCP's bandwidth allocation algorithms perform poorly. The result is collisions, high packet loss, and a devastating effect on performance. Because the network has full bisection bandwidth, however, these collisions mostly occur at the receiver, giving the FDS software an opportunity to prevent them.

FDS does so with a hybrid request-to-send/clear-to-send (RTS/CTS) flow-scheduling system. Large messages are queued at the sender, and the receiver is notified with an RTS. The receiver limits the number of CTSs 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.

With each node reading 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; but our applications use the preferred zero-copy versions.)

The standard Windows C runtime library's `malloc` services large memory allocations by passing them through to the operating system's underlying `VirtualAlloc()`. FDS does frequent large memory allocations so the default general-purpose behavior incurs substantial page fault costs. Thus for large mallocs, FDS exclusively uses application buffer pools.

3 Sort algorithm

The sort application consists of one distinguished head process and n worker processes. The input is given in a

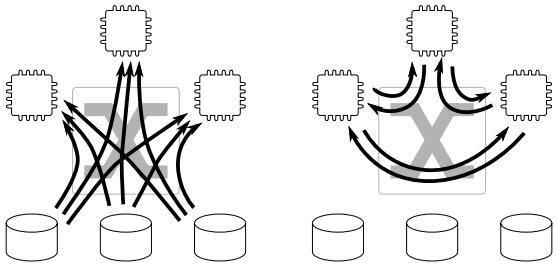

Figure 1: *Read phase.* In its reader role, each sort process (a) streams tracts, gathered by FDS from many tractservers, from its part of the input file. It bins the records, and (b) transmits the bins to other sort processes acting the the bucket-receiver role.

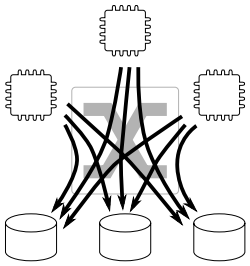

Figure 2: *Write phase.* After completing its sort, each bucket-receiver streams tracts to its output file; these tracts are scattered by FDS across the tractservers.

single FDS blob (file), from which each worker reads a separate subset of tracts. Figures 1a, 1b, and 2 show a coarse overview of the sort algorithm.

In the first *read phase*, each sort process performs two tasks simultaneously. First (Figure 1a), each sort process reads tracts from its assigned region of the input file. Simultaneously (Figure 1b), as each tract arrives (in arbitrary order), the sort process shuffles the tract's records into output bins according to the bucketing table. As each bin fills, the process sends the bin to the appropriate "bucket-receiver", some destination sort process. The buckets form an ordered partition of the keyspace. During the read phase, each process acts in both roles (reader-binner and bucket-receiver) simultaneously.

Each bin is 8MB, the same size as an FDS tract, to make network scheduling more consistent across the aggregate system. We have not determined whether performance is sensitive to this value.

In the second phase, the *write phase* (Figure 2), each sort process sorts its bucket and writes it to a separate output blob (file). The sorted result is distributed among n files.

3.1 Setup

When the sort processes start, each one fetches the FDS TLT from the central metadata server. Each process opens the blob and learns its length. Proceeding simultaneously, each process deterministically computes a global partitioning of the blob extent, into:

- one *initial assignment* for each sort process: the first k tracts are an initial work assignment for process 0; the next k for process 1, and so on.
- *dynamic work queue*: all remaining tracts comprise the dynamic work queue

Each process computes the same sets. Every process notes its own initial assignment; the head process also records the dynamic work queue, which it will dole out later to the other processes.

The head process' first task is to read 1.5 million records: 6000 records from each of 256 tracts selected uniformly at random from the input blob. From the samples it computes the key distribution and hence the assignment of key ranges to buckets. The assignment is weighted to account for statically-observed imbalances in hardware nodes' write speeds, to avoid stragglers in the write phase. When the head process has measured the entire sampling set, it unicasts the computed distribution to all the other sort processes.

To maximize utilization, the head process' sampling task commences simultaneously with the bulk reading work of the other processes (Section 3.2). Until the bucket distribution becomes available, the other processes buffer the data they have started reading.

3.2 Read phase

At system start, every sort process immediately begins issuing reads for tracts in its assignment (the head process continues reading its assignment after it completes reading its sampling set). Every tract read is buffered in the sort process' memory until the process learns the bucket partitioning from the head process. At that point, the process creates one bin for each remote bucket, and begins *binning* the read tract data into those bins. As each bin fills, it is sent to the remote sort process responsible for the bucket, and a fresh bin begun.

3.2.1 Disk stragglers

One potential source of stragglers in the read phase is slow disks. The sort application typically does not see the effect of slow disks because FDS biases the distribution of tracts to tractservers according to disk speed (see Section 2.2).

3.2.2 Reader-binner network stragglers

Another source of stragglers are slow read processes. A read process may be slow because it is running on a machine with poor network read performance, although in our record run we chose to avoid this cause by choice of process-to-machine assignments. Read processes may also be slow due to being unlucky in disk read queues, unlucky in network queues, or a process may be so unlucky as to need to send a disproportionately large number of bins to a single bucket, and hence unable to exploit network cross-section bandwidth. In theory, with large n, these sources of unluckiness regress to the mean; in practice, we saw as much as 15 seconds of straggle from unexplained sources.

Fortunately, all sources of straggle in the read phase, regardless of source, are addressed by a single mechanism, *dynamic work allocation*. As each sort process nears completion of its read assignments, the process queries the head process for additional assignments; the head process doles out assignments from the dynamic work queue. The dynamic work-assignment scheme maximizes use of sort processes of varying performance while preventing stragglers. Unlike sort systems that read from local disks, FDS' system organization makes it possible to dynamically allocate binning work: any sort process can read any tract from the input file, without locality constraints.

The choice of size of work allocation affects performance. Small assignments are beneficial because they reduce the impact of stragglers: a slow machine given a small assignment introduces only a small amount of delay before the read phase ends. On the other hand, handing out many small assignments increases the burden on the head process. Our system uses a "Zeno allocator": the allocation size is the amount of remaining work divided by the number of sort processes, constrained by maximum and minimum bounds. During the early part of the read phase, maximum-size allocations are handed out. As the phase nears its end, as the remaining work queue drains, smaller and smaller shares are handed out, until the tail of the queue is drained with minimum-size allocations. The maximum bound presents very little load on the head process, but prevents giving out all the work up front; that is, it keeps allocation dynamic. The minimum bound is chosen to avoid overwhelming the head process at the tail of the phase.

3.2.3 Bucket-receiver network stragglers

Sort processes acting the bucket-receiver role can generate stragglers for the same reasons as in the reader-binner role. Stable causes (machines that consistently receive slowly) could be addressed by static bucket distribution bias; instead, we simply did not run sort processes on slow machines. Even unluckiness is addressed to some extent by the dynamic work allocation: bucketers using the network heavily use more of the NIC read bandwidth, and hence starve the reader-binner role, but that starvation is made mostly harmless by dynamic work allocation.

3.2.4 Read phase termination

At the end of the first phase, the head sort process hands out the last track assignment. As each sort process returns for additional assignments, it learns that the dynamic work queue is empty. From that point, the process completes its remaining read assignments, binning records and transmitting bins to remote buckets. When a process finishes binning its last tract, it sends a "last-bin" message to every other bucket.

3.2.5 Saturating network read bandwidth

Note that during the read phase, every sort process is reading from the network twice as much data as it writes: one read from disk and a second read of bins incoming to its bucket, versus one write for outgoing buckets.

Therefore, it is important to maximally utilize all reader processes' NIC read bandwidth. The startup phase is designed to maximize read bandwidth utilization: each process begins reading data immediately, even before it knows where the data should go. This strategy immediately saturates the available read bandwidth with FDS file read activity. As the phase progresses, bin transfers come to occupy about half of the read bandwidth. At the tail of the phase, as the file reads end, buffered bins can absorb the remaining read bandwidth.

It may seem surprising that we initially create big backlogs of buffered data at the beginning of the phase. However, the network write bandwidth required to drain those buffers is abundant, and while the read bandwidth required to absorb them into buckets is indeed the scarce resource, that resource is being well-utilized throughout the phase.

3.3 Write phase

The "last-bin" messages form a barrier. Once each sort process receives n last-bin messages, one from every other sort process, it knows that no further records are destined for its bucket; it may begin writing the sorted data in its bucket back to disk (Figure 2).

3.3.1 Overlapped sort

A naïve writer implementation would wait for the barrier, quicksort all of the data in its bucket, then stream the sorted bucket to its place on disk. Quicksorting the

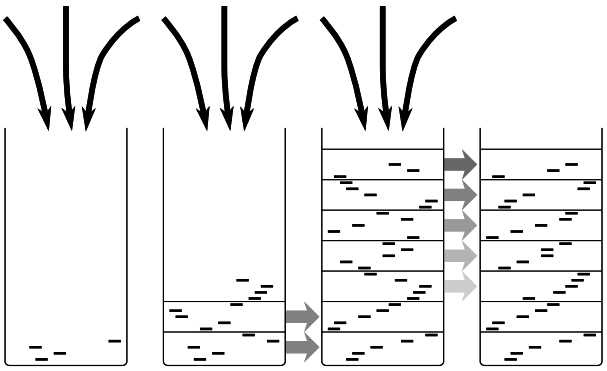

Figure 3: Incoming bins pour into the bucket. As each subbucket is filled, an asynchronous thread quicksorts it while the bucket continues to fill.

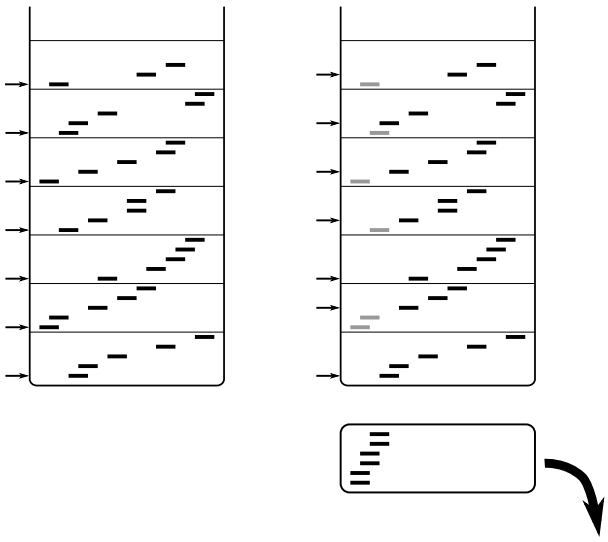

Figure 4: A merge pass repeatedly selects the lowest-valued record from among all subbuckets, creating tract-sized output buffers to write to the file system.

data sequentially (after read and before write) introduces an unacceptable delay: 16 seconds out of our 60-second budget. Therefore, it is vital to overlap partial sorts with the read phase.

Our first attempt to introduce overlap was to maintain buckets as a heap, heap-inserting each record as bins arrive. This strategy proved to introduce such a high CPU cost during the read phase that CPU became the bottleneck resource. We conjecture that this is because the heap traversals wreaked havoc on the CPU's memory cache, but we did not verify that hypothesis.

Figure 3 illustrates our approach. During the read phase at one process, incoming bins "pour" records into the bucket. The bucket is divided into "subbuckets" of 250MB (thin horizontal lines). As each subbucket fills, an asynchronous thread (grey arrows) quicksorts the subbucket.

Once the last subbucket is quicksorted, records from all of the subbuckets are merged into buffers that are issued as write requests to the output file for the bucket. Figure 4 illustrates the merge process: one pointer per subbucket tracks the next record in the subbucket. Output buffers are filled sequentially, one record at a time, by selecting the minimum record under one of the pointers.

This algorithm minimizes the latency between the phase barrier (when the last "last-bin" message arrives) and the point at which the process can issue its first output tract write to FDS storage. Small subbuckets are better for reducing the latency, as they reduce the size of the final quicksort, the only one on the critical path.

On the other hand, small subbuckets mean more subbuckets, increasing the amount of work performed for *every record* while merging subbuckets to write. At some threshold, subbuckets are so numerous that CPU time traversing the merge pointers becomes a write bottleneck.[9] The choice of 250MB subbuckets reflects an informal empirical sensitivity analysis that prevents sort process CPU from becoming a bottleneck during the write phase. At this size, we see about 700–1500ms latency due to quicksorting the last subbucket.[10]

3.3.2 Disk stragglers

As in the read phase, variations in tractserver disk performance may lead to stragglers. And as in the read phase, this problem is solved transparently in FDS by biasing the tract distribution using nominal disk speed.

3.4 Handling input larger than memory

The algorithm described above, like many algorithms used for MinuteSort, assumes that the entire input fits into the aggregate main memory of the sort nodes. A straightforward extension to a two-pass sort allows sorting of datasets of arbitrary size (up to the cluster's total disk capacity).

The main change required is an increase in the number of buckets so that the size of any single bucket does not exceed the memory on a compute node. After the input size is determined, the number of buckets is computed as the larger of the following two quantities:

[9] Our parameters give about 24 pointers per bucket, so we examine them linearly; perhaps CPU time could be reduced by maintaining the pointers in a heap. We did not evaluate this alternative.

[10] Perhaps dynamically reducing the subbucket size during the sort, analogous to the Zeno work allocation scheme above, could reduce the size of the last subbucket while keeping the number of subbuckets small. We did not evaluate this alternative.

- A number of buckets that ensures the size of any single bucket is smaller than a compute node's main memory (with high probability); and
- The number of sort processes. (Having fewer buckets than sort processes gives up parallelism.)

An intermediate blob is created for each bucket. During the first pass through the input, each sort node streams data in from its portion of the total input blob and writes each record to the appropriate intermediate blob. The memory used by each sort node in this step is the number of output buckets times the size of the per-bucket output buffer used to batch records into tracts before writing.

After this pass is complete, each bucket is sorted serially: a bucket's contents are read from a blob into a single sort node's memory, sorted, and written to an output blob. The hybrid qsort and mergesort algorithm shown in Figure 3 is still useful for this step. Bucket sorting is done in parallel by a large number of sort nodes. Dynamic allocation of buckets to sort nodes prevents stragglers.

As with MinuteSort, the bucket boundaries can either be assumed uniform (for Indy) or based on input sampling (for Daytona). The amount of input sampling required for even buckets goes up as the number of buckets goes up; our algorithm selects a number of samples proportional to input size. As with our MinuteSort algorithm, a safety margin is desirable between expected bucket size and maximum possible bucket size to avoid overflow due to nonuniformity in the input.

4 System configuration

This section provides specific details of our system configuration for the reported result.

4.1 Computational hardware

During our record runs, the FDS cluster employed computational hardware as enumerated in Table 1. There's nothing profound or deliberate about our selection of hardware; we bodged it together from our own budget (bought), surplus machines (begged), and other groups' hardware pressed into service temporarily (borrowed). As we introduced a class of machines to the cluster, we characterized its performance, and assigned it a role to maximize benefit to the sort. Because FDS eliminates disk locality considerations, new machines can be without concern for intra-node resource balance.

The NICs consist of 205 Intel X520 dual-port 10Gbps Ethernet NIC cards and 50 HP NC522SFP dual-port 10Gbps Ethernet NIC cards. (When we say "NIC" in this paper, we are referring to one of the two ports on a dual-port NIC card.) Some machines can not do enough useful work to saturate both NICs, so the table only reports the number of "connected" NIC ports, those occupying ports in the TORs (Section 4.2).

4.2 Networking hardware

As described in Section 2.3, the compute hardware is interconnected by a CLOS-topology full-bandwidth interconnect. Twenty-two Blade Rackswitch G8264 64×10 Gbps Ethernet switches serve as eight spine switches and 14 TORs.

The computers are all also connected by a 1 Gbps management link, but no application or filesystem data traverses that link.

4.3 Formal report

Table 2 records the formal benchmark output of our test runs.

5 Summary

FDS is a general-purpose scalable parallel blob store that exploits a full-bandwidth interconnect to expose the entire cluster's disk bandwidth to remote clients. The sort performance results in this paper demonstrate the power of the architecture: in both Daytona and Indy sorts, the system reads the data remotely to the sort machines, sorts the data across the network, and writes it remotely back to storage.

Performant remote file access imparts a flexibility absent in contemporary distributed storage systems. Beyond sort, FDS supports a broad variety of scalable large-data applications. It does so without demanding that cluster nodes balance compute and disk performance; more importantly, it does so without demanding that applications observe locality constraints.

References

[1] APACHE. Hadoop, 2008. `http://hadoop.apache.org/`.

[2] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1265–1276.

[3] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 10–10.

[4] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

quantity	Machine class	connected 10 Gbps NICs	data disks	disk type	RAM GB	cores	source
metadata server							
1	HP DL 360	1			24	2	begged
tractservers							
21	HP DL 326	2	16	10K RPM SAS	48	8	bought, begged
10	HP DL 326	2	11	7,200 RPM SATA	12	4	bought
3	HP DL 326	2	10	7,200 RPM SATA	12	4	bought
1	HP DL 326	2	9	7,200 RPM SATA	12	4	bought
76	HP DL 360	1	5	10K RPM SAS	24	2	begged
24	HP DL 380	2	7	10K RPM SAS	24	4	begged
sort application							
14	Dell PowerEdge	2			24	8	bought
27	HP DL 316	2			48	8	borrowed
61	HP DL 326	2			48	8	borrowed
18	Silicon Mechanics	2			96	12	borrowed

Table 1: Enumeration of hardware resources

	Indy	Daytona
Data size	1470 GB	1401 GB
median	59392 ms	59045 ms
runs	15	15
min	57957 ms	57353 ms
max	66892 ms	63110 ms
checksum	1b617e245d3d92c55	1a187b1de8697149e
duplicate keys	0	0

Table 2: Formal benchmark data

[5] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication* (New York, NY, USA, 2009), SIGCOMM '09, ACM, pp. 51–62.

[6] MICROSOFT. MinRTO Hotfix. `http://support.microsoft.com/kb/2472264`.