DeepSort: Scalable Sorting with High Efficiency

Zheng Li[†] and Juhan Lee[‡]

***Abstract*—We designed a distributed sorting engine optimized for scalability and efficiency. In this report, we present the results for the following sort benchmarks: 1) Indy Gray Sort and Daytona Gray Sort; 2) Indy Minute Sort and Daytona Minute Sort. The submitted benchmark results are highlighted in Table I.**

I. Introduction

DeepSort is a scalable and efficiency-optimized distributed sorting engine. It performs out-of-place external sorting supporting general key-value record types. We engineered the software design for high performance based on two major design decisions:

- Parallelism at all sorting phases and components is maximized through properly exposing program concurrency to thousands of user-level light-weight threads. As a result, computation, communication, and storage tasks could be highly overlapped with low switching overhead. For instance, since hard drives are optimal for sequential access, the concurrency of storage access is limited while computation and communication threads are overlapped to take advantage of remaining resources. We use the Go programming language to orchestrate the massive parallel light-weight threads, i.e. Go routines. Individual sorting functions are implemented in C for high performance.
- Data movements in both hard drives and memories are minimized through an optimized data flow design that maximizes memory and cache usage. Specifically, we avoid using disks for data buffering as much as possible. For data buffered in memory, we separate key data that needs frequent manipulation from payload data that is mostly static. In this way, performance could be improved through minimal data movements for payloads and proper caching on the frequently accessed keys.

We deployed DeepSort on a commodity cluster with approximately 400 nodes. Each node has two 2.1GHz Intel Xeon hexa-core processors with 64GB memory, eight 7200rpm hard drives, and one 10Gbps Ethernet port. These servers are connected in a fat-tree network with 1.28:1 top-of-rack (TOR) to spine subscription ratio. We equipped these servers with CentOS 6.4 and *ext4* file systems. We carefully measured the performance of DeepSort on this platform based on Sort Benchmark FAQ [1] for both Gray Sort and Minute Sort. Our results show that the system can sort 3.7 TB of data within a minute based on Daytona Minute Sort rules. All submitted benchmark results are listed in Table I.

Contact: zheng.l@samsung.com

† Research Engineer, Cloud Research Lab, Samsung Research America Silicon Valley, San Jose, CA U.S.

‡ Vice President, Intelligence Solution Team, Samsung Software R&D Center, Korea

In this rest of the report, we explain the software design of DeepSort in Section II. The underlying hardware and operating system platforms are described in Section III, followed by the detailed experiment measurements in Section IV.

II. Design of DeepSort

The design of DeepSort targets high efficiency at a large scale. To reach this goal, we designed a fluent data flow that shares the limited memory space and minimizes data movement. DeepSort expresses data processing concurrency through light weight threading and optimize parallelism at all stages and components of the program. This section explains the implementation of our design philosophy and specific design choices in the distributed external sorting program.

A. Design Overview

The overall design is depicted in Figure 1 from the perspective of records to be sorted. A record is first fetched from the source disk to the memory, sorted and distributed to the destination node, merged with other records based on order, and finally written to the destination disk. For cases like Gray Sort, in which the amount of the data is larger than the aggregated capacity of memory, multiple rounds of intermediate sorts are executed. The final round merges spilled intermediate data from previous rounds. The input of the unsorted records is distributed evenly across nodes, and the output is also distributed based on key partitions. The concatenated output files are formed into a globally sorted list. RAID-6 erasure coding across nodes is used for replication in Daytona sort [1]. The data read procedure includes detecting possible failures and recovering data from erasure decoding. The data write procedure is followed by sending data to other nodes for erasure encoding.

Each process phase in Figure 1 is described below from source disks to destination disks. A table of notations is presented in Table II for ease of understanding. We leverage both data parallelism and task parallelism to exploit the concurrency in the distributed sorting. There is only one process per node with thousands of light weight threads to handle different functionalities concurrently. To illustrate the abundant parallelism, we count the number of light weight threads as in Table III. The light weight threading library, i.e. Go runtime, multiplexes them onto OS threads for high performance. Memory is shared among these threads to minimize IO operations and thus optimize the overall performance. As we maximize the memory usage by sharing the same address space, only pointers are passed between different phases except across nodes or rounds.

[1]Based on email communications with the SortBenchmark committee, such replication complies with the competition rules.

TABLE I
SUMMARY OF RESULTS

Category	Variant	Data Size	# of Nodes	Time (seconds)	Quantity of Interest
Gray Sort	Indy	109.728 TB	381	1650	3.99 TB/min
Gray Sort	Daytona	101.376 TB	384	2050	2.97 TB/min
Minute Sort	Indy	5249.6 GB	386	59.859	5249.6 GB
Minute Sort	Daytona	3686.4 GB	384	57.675	3686.4 GB

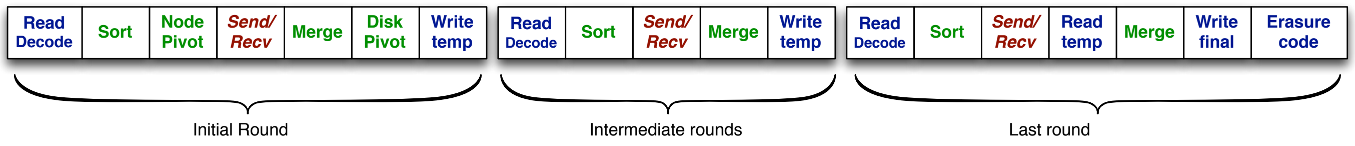

Fig. 1. Pipeline stages of the DeepSort design.

TABLE II
SUMMARY OF NOTATION

Notation	Typical Value[1]	Comments
N	400	Number of nodes in the cluster
R	10	Number of rounds
D	8	Number of disks per node
m	160	Merging threshold at destination nodes
B	10	Number of total batches per round
s	3	Oversample ratio in partition [2]
B_{node}	2	Batches per node needed for node splitter [2]
B_{disk}	4	Batches per node needed for disk splitter

[1] Illustrative number for 100 TB sort across 400 machines in Section III.
[2] For skewed data set, we increase S and B_{node} for load balancing.

TABLE III
NUMBER OF LIGHTWEIGHT THREADS PER NODE

Type	Number	Persistence	Typical Value
Read	D	Yes	8
Sort	$D \times B \times R$	No	800
Send	$N-1$	Yes	399
Receive	$N-1$	Yes	399
Pre-merge	$N \times D \times B \times R/m$	No	2000
Merge	$D \times R$	No	80
Write	D	Yes	8
Partition	1	No	1
Memory Manage	1	Yes	1
Total	$O(DBR + N)$		approx 3.6k

A record is first fetched from a source disk by a reader thread. One reader thread is assigned to each hard drive to guarantee the sequential access of disks. Data is fetched from disk consecutively and then divided evenly over the course of R rounds, where each round handles B batches of data. For Daytona sort, batch is also the erasure coding granularity, i.e. size of coding block. Once a batch of data is filled, a sorting thread is created. Once a round of data is fetched, reader threads are paused to wait for local write threads and synchronize with all other remote nodes. There are D reader threads per node and they are persistent through the whole program. If one node is missing or corrupted, other nodes will split its tasks and fetch data from peers for erasure decoding. Any missing data will be recovered.

Each sorting thread is started after a batch of data is fetched from one disk. The sorting thread leverages GNU C library for general sorting of any record and key types. After local sorting, the data are split based on their corresponding destination. At a master node, the partition is conducted in parallel with the first round of local sorting across all nodes. This partition will be explained in Section II-B. The sorting thread concludes after sender threads of all destinations are notified. There are $D \times B$ sorting threads each round per node. They start and conclude at different time based on the performance of disk and network access. They are also overlapped with disk and network threads of later and previous batches, respectively.

The communication threads are separated into senders and receivers. On each node, there are $N-1$ sender threads and $N-1$ receiver threads to build an all-to-all virtual crossbar for data transfer. These $2N-2$ communication threads are persistent through the whole program. Sorting threads notify sender threads once data is ready, and receivers notify mergers when data arrives. The details of the communication will be explained in Section II-D.

While short sorted lists are being collected by the receivers, the lists are merged as they arrive. For each node in each round, there are $N \times D \times B$ lists from all the peers to be merged before writing to disks. To maximize performance, there are two levels of merging threads. The first level merging threads, i.e. premergers, start after enough lists ($> m$) have been received. The second level threads merge the last arrived m lists with the resulting lists from the premerger threads. For the non-last rounds in multi-round sorting, e.g. GraySort, it is not necessary to wait until all lists arrive before the second level merging begins. To increase overlap between communication and disk access, the second level merging threads start shortly after disk read processes have concluded locally for this round, and the lists received after that will be postponed to merge in the next round. In the extreme skewed case, memory becomes full before all the $N \times D \times B$ lists are received. The memory management thread (Section II-E) also notifies the second level merging threads to start so that data can be spilled and memory space can be released. Like local sorting functions, merging functions are also based on open source libraries for general key and value sizes.

Similar to reader threads, there are D writer threads, each of which corresponds to one hard drive. Each of the writer thread is accompanied by one second level merge thread that corresponds to its key space. Therefore, the writing starts shortly after the data has been received. In case the data does not fit into the aggregated memory, multiple rounds are used. The write phase at the first $R-1$ rounds spill the results into sorted temporary files. Typically, writer threads generate one file per round per disk. In extremely skewed cases where the memory management thread notifies early second level merging, multiple files are generated per round per disk. The write phase at the last round merges temporary files generated at all rounds with the final data received to form the output sorted list. If replication is required as in Daytona sort, the write phase will be followed by erasure encoding detailed in Section II-F. Typically, one final output file with optional codec information is generated on each disk. The writing of output files is overlapped with erasure encoding and corresponding communication. For Indy Minute Sort, two output files with separate key partition spaces are generated per disk so that the writing of the first partition could be overlapped with the communication of the second partition.

B. Partition of the key space

In parallel with the data path described above, we perform partitioning of the key space and thereby compute the splitters, which are the keys that determine the boundaries among nodes and disks, so that the output could be almost uniformly distributed among all the disks. By evenly sampling data across all nodes as late as possible with small performance interference, we can cover a large portion of the data set. We use the following algorithms to create a set of splitter keys targeting even output partitions.

There are two levels of partitions: the first level node partitioning determines the key splitters among nodes in the cluster, i.e. node splitters, while the second level disk partitioning determines the key splitters among disks within a node, i.e. disk splitters. Both levels are variances of sample splitting [2]. Unlike in-memory sorting, where splitters can be gradually refined and data can be redistributed, rebalancing in external sorting generally comes with high IO overhead. Therefore, histogram splitting is not used.

The $N-1$ node splitters that partition key ranges among nodes are determined in a centralized master node. Each node is eligible to be a master node, but in case redundancy is neeeded, a master node could be picked by a distributed leader-election algorithm. From all the nodes, after B_{node} batches of data are sorted, $N-1$ node splitter proposals are picked to evenly partition the batch of data into N segments. These node splitter proposals are aggregated to the master node. A total of $N \times D \times B_{node} \times (N-1)$ proposals are then sorted in the master node, and the final node splitters are determined by equally dividing these proposals. Considering there might be data skew, we over-sample the data s times by locally picking $N \times s$ node splitter proposals that evenly partition the data

Algorithm 1 Partition among nodes: Proposals

```
Require: $N$, $s$, $D$, $B_{node}$
 1: procedure PROPOSAL($l$)                    ▷ A thread at each node
 2:     $count = 0$
 3:     while $list \leftarrow l$ do                    ▷ Pointer of sorted lists
 4:         $len = length(list)$
 5:         for $i = 0 ... N \times s - 1$ do
 6:             $pos = len \times i/(N \times s - 1)$
 7:             Send $list[pos].Key$
 8:         end for
 9:         $count = count + 1$
10:         if $count >= D \times B_{node}$ then
11:             break                    ▷ Sample threshold reached
12:         end if
13:     end while
14:     Receive $splitters$          ▷ Final splitters from the master
15: end procedure
```

Algorithm 2 Partition among nodes: Decision

```
Require: $N$, $s$, $D$, $B_{node}$
 1: procedure DECISION                    ▷ One thread in master
 2:     $list = []$
 3:     for $i = 1 ... N^2 \times D \times B_{node} \times s$ do
 4:         Receive key
 5:         $list = append(list, key)$
 6:     end for
 7:     Sort($list$)                    ▷ Sort all keys received
 8:     for $i = 1..N-1$ do
 9:         $pos = i * length(list)/(N-1)$
10:         Broadcast $list[pos]$
11:     end for
12: end procedure
```

into $N \times s$ segments. The final $N-1$ node splitters are thus picked from a sorted listed of $N \times D \times B_{node} \times N \times s$ proposals. This process is carried in parallel with data read and sort, but before the first round of sending data to its destinations to minimize network overhead. We describe the node splitter design in Algorithm 1 and Algorithm 2.

The $D-1$ disk splitters per node are determined distributively at each destination node. After receiving data that equals the size of B_{disk} input batches from all peers, the destination node starts a separate thread that calculate the disk splitters in parallel with receiver threads and merger threads. When the disk splitter threads start, there are about $B_{disk} \times N \times D$ sorted lists. For the i-th disk splitter, every sorted list proposes a candidate. This candidate will be used to divide all the other lists. The one that best approaches the expectation of i-th splitter is picked as the final disk splitters. Unlike the node splitter algorithm, the disk splitter algorithm is not on the critical path of the program and is thus executed in parallel with the first level merging.

Compared to existing external sort designs (Hadoop [3] and TritonSort [4]), we sample and partition the key space in parallel with reading and sorting. We also delay the disk partition as late as possible but before second level merging.

TABLE IV
EXTERNAL SORT SAMPLE COMPARISON

System	Time of Sample	Coverage rate	Location
Hadoop [3]	Before sorting	$1/10^6$	at most 10 nodes
TritonSort [4]	Before sorting	approx $1/1923$	All nodes
DeepSort	During sorting	$1/50$	All nodes

As shown in Table IV, DeepSort achieves higher sampling coverage across all computing nodes in participation. As will be shown, DeepSort handles skewed data much better than existing designs.

The cost of high sample rates only reveals in Daytona Minute Sort. For Indy Sort, we set B_{node} to 1 and s to 1 to minimize the time impact of partition. The partition takes less than 2 seconds to complete at the master node. In order to account for skewed data set, we set B_{node} to 8 and s to 25 for Daytona Gray Sort. The node partition process takes approximately 30 seconds to complete at the master node. This delay has minimal impact of overall long sorting time but resulting in a reasonably balanced output. We set B_{node} to 5 and s to 20 for Daytona Minute Sort. The node partition takes approximately more than 10 seconds to complete. This delay has a major impact within a minute. The resulting Daytona Minute Sort data size is only about 70% of the Indy Minute Sort. Unlike node partitioning, disk partitioning is not in the execution critical paths and does not cause extra run time.

Algorithm 3 First-level premerger based on heap sort

```
Require: m, array, boundary, destination
 1: procedure MERGE
 2:     $heap \leftarrow []$
 3:     for i = 1..m do                              ▷ Initialize heap
 4:         heap[i].Key = array[boundary[i].Start]
 5:         heap[i].Position = boundary[i].Start
 6:         heap[i].Tail = boundary[i].End
 7:     end for
 8:     for i = m/2..1 do                            ▷ Heapify
 9:         CREATE(i)     ▷ BSD LibC macro to build heap
10:     end for
11:     while heap not empty do
12:         APPEND(destination, array[heap[1].Position])
13:         heap[1].Position ++
14:         if heap[1].Position==heap[1].Tail then
15:             k ← heap[m]
16:             $m \leftarrow m - 1$
17:             if m==0 then
18:                 break
19:             end if
20:         else
21:             heap[1].Key = array[heap[1].Position]
22:             k ← heap[1]
23:         end if
24:         SELECT(k)  ▷ BSD LibC macro to recover heap
25:     end while
26: end procedure
```

C. Sort and Merge

Unlike the rest of system, the individual sorting and merging functions are implemented based on open source C libraries, and we keep it general to any key and value sizes and types. Comparison sort has been a well-studied topic. Instead of examining the theoretical limits to pick the best algorithm, we experiment with different designs and pick the best one in practice.

For the individual local sort at the data source, we prefer in-place sorting to minimize memory footprint. The performance just needs to be higher than the IO speed so that computation latency could be hidden. Since the single disk hard drive read speed is 150MB/s, for 100-byte record, the cut-off performance is 1.5 million records per second. We picked QuickSort as the algorithm and use the implementation of GlibC *qsort*.

When hundreds or thousands of sorted list needs to be merged, we use the merge capability of heap-sort to form a multi-way merging to minimize memory foot print. Initially, a priority queue, i.e. heap, is formed using the head of all sorted list. The merged list is formed by extracting the head of the queue one by one. Every extracted element is replaced by the head of the corresponding input list, and thus the priority queue's ordering property is kept during the process. Heap sort implementation is adopted based on the *heapsort* in BSD LibC from Apple's open source site [5]. We keep its heapify macros for the implementation efficiency, but revise the sort function to facilitate the multi-list merge process. The algorithm is presented in Algorithm 3. The performance of merging determines the merging threshold m, i.e. the number of sorted lists required to trigger a merging thread. The last merging accompanying writing thread should be faster than the disk access speed to hide its latency. Our experiments show that for merging even over a thousand lists, the performance is still faster the disk access speed.

We also optimized the comparison operator to improve the performance for comparison sort. For string comparison where the order is defined in *memcmp*, we convert byte sequences into multiple 64-bit or 32-bit integers to minimize the comparison instruction counts. Such optimization has shown moderate performance improvement. The comparison operator could also be defined by users to accommodate different sorting types or orders.

D. Communication

Like many cloud applications, distributed sorting requires all-to-all reliable communication pattern. Therefore, each of the node instantiate $N-1$ sender and $N-1$ receiver threads for TCP/IP communication. In our work, we use the runtime system from the Go programming language to multiplexes these lightweight threads onto system threads for communication. Without such runtime management, setting up $2N-2$ system threads for communication scales poorly with N. Prior efforts like TritionSort [4] or CloudRAM sort [6] uses one dedicate communication thread to poll data from all computation threads and distribute them to all destination. On the other hand, by leveraging the Go runtime, we are able to

TABLE V
EXTERNAL SORT DATA BUFFERING COMPARISON

System	# of movements in HDD	# of movements in DRAM
Hadoop [3]	3 or 4	6
TritonSort[4]	2	5
DeepSort	< 2	4

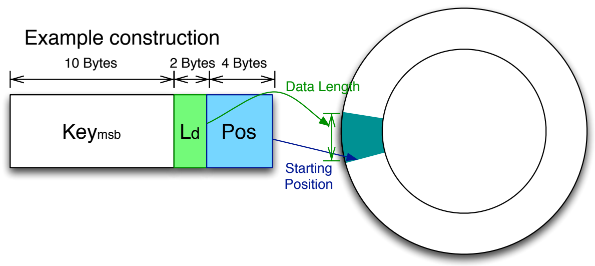

Fig. 2. The pointer structure to link the key array to the data array

implement highly scalable communication without additional code complexity.

To save the round-trip latency of data fetching, we adopt push-based communication in which the data source side initiate the transfer. The locally sorted lists are pushed to the destination. We also prioritize key transfer over value transfer so that the destination-side merging can start as early as possible.

To avoid destination overflow in push-based communication, we orchestrate the communication using a light-load synchronization mechanism between two rounds of data transfer. After a node finishes writing temporary data of a round, it broadcasts one byte synchronization message to all the peers notifying its progress. A node starts reading the input data of the next round after local disk writes concludes but hold the data transfer until it receives the synchronization messages from all the peers. This light all-to-all synchronization is performed during a relative idle period of network, and overlaps with disk access and sorting.

E. Memory Management

The memory management design goal is to minimize data movements for high performance and full utilization of resources. As shown in Table V, DeepSort minimizes data movements within hard drives and DRAM. Data records in the last sorting round read and write hard drives only once for input and output. The rest of records have one more trip to hard drives for intermediate buffering due to the fact that the aggregated memory size is smaller than overall data size.

DeepSort also minimizes the segmentation of memory and eliminates frequent movements of data within memory using the efficient memory management mechanism. Although Go language has its own mark-and-sweep garbage collection mechanism, in practice it could not recover released memory space immediately. Therefore, we manage the key and value spaces ourselves while leaving trivial memory usage to Go runtime.

DeepSort hosts two globally shared arrays that are persistent and takes the majority of available system physical memory. The first array stores the data values, which are large but infrequently access, and the second array stores the keys, which are frequently accessed and altered, with corresponding pointers to the data array. We refer to them as data array, and key array, respectively. In this memory structure, the only overhead per record is the pointers that link the key array to data array. We have plotted the pointer from the key array to the data array in Figure 2.

The pointer structure is parameterized to handle various key and value sizes. It has three fields. An illustrative construction is presented in Figure 2 and described as below.

1) Key_{msb} is the most important bytes of the key. Its size is a fixed configuration parameter. Users can write custom compare functions that further tap into the data array as the extension of the key. This is made possible as the whole structure is passed into sort and merge functions and the data array is globally visible in a process. In the example construction, we set the size to 10 bytes.
2) L_d represents the size of the payload data in terms of a predefined block size B_s. The actual payload data size is $B_s \times (L_d + 1)$ bytes. For instance, we configure the parameter B_s as 90 bytes and the width of L_d to two bytes so that the variable payload size for each record can be up to 5.9 MB. For the fixed size 90-byte payload, we set L_d to 0.
3) *Pos* indicates the starting location of the data block. Although the key array is frequently manipulated for sorting, merging, and splitting, the data array doesn't move until records have been assembled for network communication or disk accesses. At that time, the data is assembled the same as the input format but with a different order based on the key array.

For performance optimization, it is desirable to align the size of the pointer structure to cache-line and memory access. In the shown example, the structure size is 16 bytes.

The key array and data array are shared globally between source sides and destination sides. They are working as circular buffers. When sorted data from peers have arrived, they are appended to the end of the array. When locally sorted data have been released from the source, their space is reclaimed. There is a dedicate thread that manages global resources in parallel with other activities. To accommodate the buffering behavior and moderate skewed data set, the global arrays have auxiliary space that can be configured depends on application. Once the auxiliary space is about to be exhausted from receiving large amount of the extremely skewed data, the memory management thread will notify the second level merging threads to start spilling data to disks. Space will be freed after spilling, and such mechanism might iterate multiple times.

Figure 3 shows five snapshots of the key array memory layout. The process is explained as follows:

a) In the beginning of a round, the locally read data starts filling the memory. There might be some leftovers from previous round. All nodes synchronize with each other and

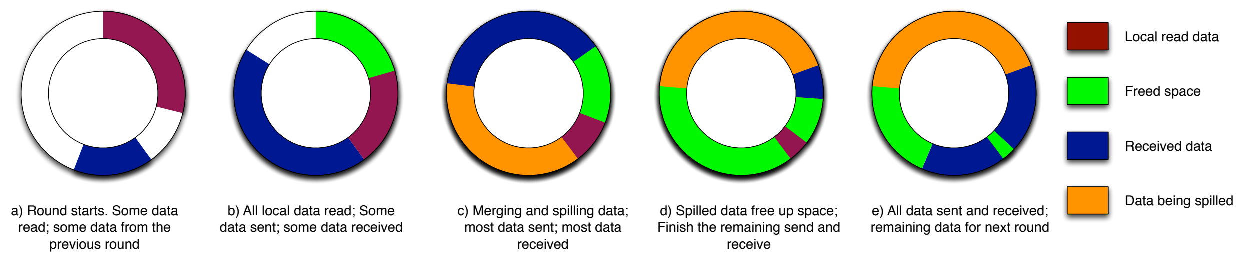
a) Round starts. Some data read; some data from the previous round
b) All local data read; Some data sent; some data received
c) Merging and spilling data; most data sent; most data received
d) Spilled data free up space; Finish the remaining send and receive
e) All data sent and received; remaining data for next round
Local read data
Freed space
Received data
Data being spilled

Fig. 3. Snapshots of memory layouts as a circular buffer.

get ready for all-to-all communication.

b) In the middle of a round, more data has been read. The initial batches of data has been sent out, and the space has been freed. Data has been received from peers of this round, and the memory gets filling up. The first-level mergers start processing received sorted lists.
c) Towards the end of a typical round, after all local data has been read, the disks are free to be written. The second-level mergers start to merge and writers start spilling sorted lists. If it is the last round, the spilling will wait until all data has been received.
d) Towards the end of a extremely skewed round, data is being received to fill the full memory space. Memory management thread notifies second-level mergers to spill data out to free space.
e) At the end of a round, all current round data have been received and sent. If the round is not the last, there might be leftovers for the next round.

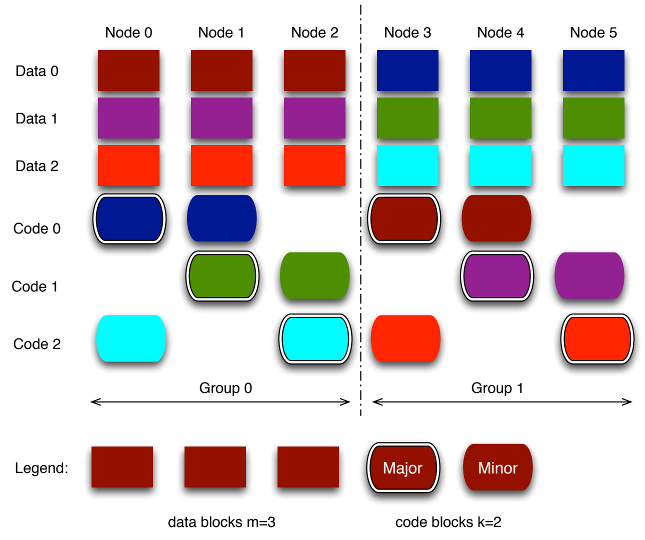

Fig. 4. Illustration of data layout for replication

F. Replication Strategy

We use erasure code to protect data failures across nodes. As an optional feature of DeepSort, replication can be turned on to protect data failures of up to two nodes in Daytona Sort. It can also be turned off for better performance in Indy Sort. Replication applies to input and output data but not intermediate data.

Specifically, we use Minimal Density RAID-6 Coding across nodes. This is a Maximum Distance Separable (MDS) codes. For every k nodes that hold input and output data, we use extra m nodes to hold codes that are calculated from the original k nodes. Since it is a MDS code, the system is able to tolerate any m nodes loss. We divided all nodes n into n/k k-node groups in a circle. The calculated m codes from each group are evenly distributed in the k nodes of the next group. In Figure 4, we show an illustration of data layout for a total of 6 nodes with k equaling 3 and m equaling 2. Using RAID-6 coding, we can trade-off between disk bandwidth and computation resources without sacrificing the data replication rate. In our Daytona sort experiments, we set k equals 8 and m equals 2. Therefore, it could tolerate at most 2-node failures and the extra disk bandwidth required for replication is only 25% of the data size.

For encoding and decoding, our implementation uses Jerasure erasure coding library from University of Tennesse [7]. We embedded this codec into our sorting pipeline. Decoding is part of data read. If any node loss or data corruption is detected, the data has to be pulled from multiple nodes to recover missing data. In case of no loss, the decoding is simplified as a direct read from data hard drives. Encoding is embedded in the final data write. We have completely overlap the computation, communication, and disk access for the write replication process. Figure 5 illustrates such pipeline.

On the destination side of DeepSort, the encoding process starts after initial sorted data has been merged and assembled in the pipeline, as shown in Figure 5. While the sorted data is being committed into hard drives, they are also being sent to major encoding nodes in the next k-node group. For each block of data, the location of the major encoding node is calculated based on a round-robin formula for workload balance across nodes and disks. In the major coding node, all k blocks of data are gathered and passed to the Jerasure library for coding. After encoding computation, m blocks of the data are generated. The first coding block is written to the local disk, and the rest are sent to minor coding nodes. For instance, in Figure 4, there are two 3-node group. The lowest order sorted data from all nodes of the first group are destined for the first node in the second group, i.e. “Node 3” is the major coding node. “Node 4” is the minor coding node for the lowest order sorted data from all nodes of the first group. After the data and codes have been written, a ledger file is recorded on the major and minor coding nodes for completeness check.

The erasure decoding process is similar to the encoding

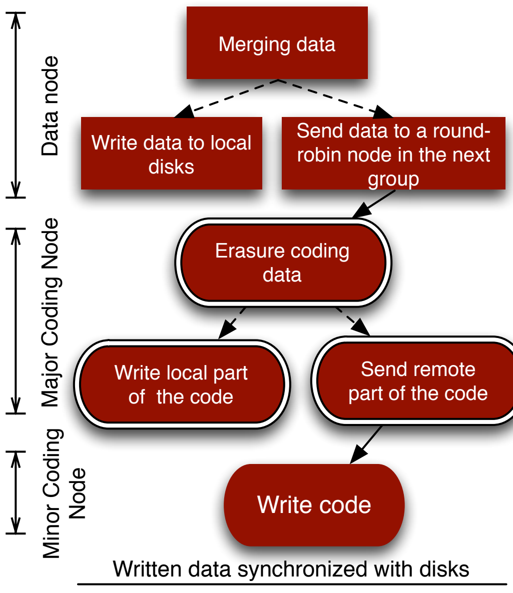
Data node
Merging data
Write data to local disks
Send data to a round-robin node in the next group
Major Coding Node
Erasure coding data
Write local part of the code
Send remote part of the code
Minor Coding Node
Write code
Written data synchronized with disks

Fig. 5. Data pipeline of erasure coding

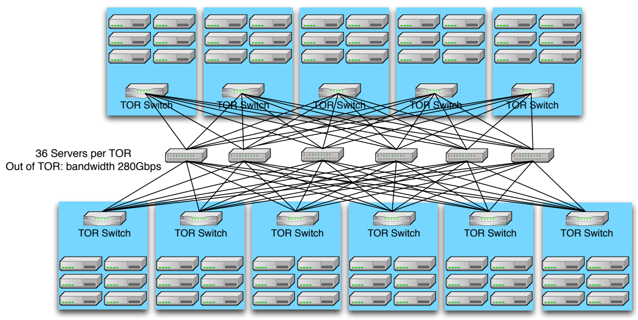

Fig. 6. Network topology of the cluster

process. Coding nodes collect short acknowledgment messages from data nodes. Major coding nodes compare them with ledgers for completeness and send short acknowledgment messages to minor coding nodes. In case of data node failures or timeout, major coding nodes actively collect data from data nodes and minor coding nodes to reconstruct missing data using erasure decoding function in Jerasure library. The master node will also notice the failure and avoid generating splitters for the failed nodes. The reconstructed data behaves like normal input data from the major coding nodes but with a reconstruction label. In case of failures or time out at major coding nodes, the minor coding nodes act as major coding nodes for completeness check or reconstruction. The timeout value is configurable and defaults to three times of the approximate duration of read and communication at each round. Specifically, for Daytona Minute Sort, the timeout is 45 seconds and for Daytona Gray Sort, the timeout is 1.5 minutes.

III. Hardware and System Setup

A. Server Configuration

In our experiments, we used our production compute cluster comprising nearly 400 Dell PowerRidge R720 commodity servers, each with two 22nm Intel Xeon E5-2620V2 processors per node running at 2.10GHz with a 7.2 GT/s QPI connection. Each processor has six cores with HyperThreading enabled and 15MB L3 cache. There are four 16GB DDR3-1600 RDIMMs and eight 3TB 7.2K RPM SAS hard drives in each server. There is no RAID configuration across disks within a node. With *ext4* file system equipped, each hard drive is measured to have approximately 150MB/s read speed and 120MB/s write speed. However, since we do not control the placement of data, the disk speed varies substantially. There is no striping within a node. The actual number of active nodes in each experiment varies because this is a production cluster, and we can only isolate a majority of relatively idle machines for some short time slots.

We are running Cent OS 6.4 with Linux Kernel version 2.6.32. Source code are compiled using Go version 1.3.1 and GCC version 4.4.7.

Our network infrastructure is shown in Figure 6. Each of the server has a 10Gbps full-duplex Ethernet port. These servers form a two tier fat-tree network topology. Approximately 36 servers share a Quanta top-of-rack (TOR) switch. Eleven TOR switches connect to six Cisco core switches with 280Gbps rack-to-spine connection. The effective TOR:Spine subscription ratio is 1.28:1. The TOR switches are the network bottleneck and the effective out-of-TOR raw bandwidth is approximately 7.7 Gbps per node.

B. Performance Expectation

The design of DeepSort tries to hide computation, communication, memory management all behind disk IO access. Therefore, the performance expectation is straightforward to calculate.

For multi-round sort, the data has to pass disks twice. Assuming 400 nodes with 3200 disks, 150 MB/s read speed, and 120 MB/s write speed, the aggregated throughput is approximately 6.4 TB/min. Our Indy Gray Sort is about 62% of this idealized value.

For single-round sort, the data has to pass disks once. Assuming 400 nodes with 3200 disks, 150 MB/s read speed, and 120 MB/s write speed, the aggregated throughput is approximately 12.8 TB/min. Our Indy Minute Sort is about 41% of this idealized value.

IV. Experiment Results

A. Data Preparation and Validation

The data is prepared and validated based on SortBenchmark's *gensort* and *valsort* programs, respectively. The input data is distributively generated in binary records. Each record has a 10-byte key and 90-byte value. Each hard drive in the cluster contains an input data file with codec information. For Daytona sort, we modify the *gensort* program so that the erasuring codes are generated with the input data. For Indy sort, we use *gensort* directly across all nodes. For Indy sort, only uniform random data are generated. For Data sort, we also generate skewed data set. After the DeepSort is finished, the output is distributed across all the hard drives in a global order. The concatenation of these files forms a sorted list of

TABLE VI
INDY GRAY SORT

Amount of Data	109.728 TB
Duplicate Keys	0
Input Checksum	*7fbd7a61ae81a41542*
Trial 1	26 min 19.283 sec
Trial 2	27 min 28.803 sec
Trial 3	27 min 29.883 sec
Trial 4	27 min 42.999 sec
Trial 5	27 min 36.875 sec
Output checksum	*7fbd7a61ae81a41542*
Median time	27 min 29.883 sec
Sort rate	3.99 TB/min

records. We validate the order and data checksum using the *valsort* program.

We measure the overall execution time using the Linux *time* utility. All experiments have been running longer than an hour. To make sure that the sorting is performed strictly from disk to disk, three things have been taken care of: 1) We clear the operating system in-memory file caches before each run using Linux drop caches; 2) Output data is synchronized with hard drives after each run; 3) After being validated, the output files from the previous run are removed before the current run starts.

No compression is used at any stage of the program.

We measure the data size in a consistent manner of Sort Benchmark rules such that one terabyte (TB) equals 10^{12} bytes, and one gigabyte (GB) equals 10^9 bytes.

B. Gray Sort Indy Results

For the Indy variant of Gray Sort, we sorted 109.728 TB data in about 1650 seconds, yielding 3.99 TB per minute sorting rate.

Specifically, we use 381 nodes of the cluster. Each node hosts about 288GB data. The input checksum is *7fbd7a61ae81a41542* without duplication. We've sorted this data over five consecutive trials, and the running times reported by *time* utility are listed in Table VI. The output checksums are all verified to be *7fbd7a61ae81a41542*.

C. Gray Sort Daytona Results

For the Daytona variant of Gray Sort, we sorted 101.376TB uniform random data in 2050 seconds, yielding 2.97 TB per minute sorting rate. The sorting time for skewed data is less than twice of the random data.

Specifically, we use 384 nodes of the cluster. Each node hosts 264GB data. The input checksum is *76046478aca758c4d9* without duplications. We've sorted this data over five consecutive trials, and the running times reported by *time* utility are listed in Table VII. The output checksums are all verified to be *76046478aca758c4d9*.

We also prepared skewed dataset and run DeepSort using 384 nodes of the cluster, totaling 101.376TB of skewed data with a checksum of *760464fe84f3bb8567*. There are 31152217581 duplicate keys in the input. We've also sorted the skewed data over five consecutive trials, and the running times reported by *time* utility are listed in Table VIII. The checksums of output files are *760464fe84f3bb8567*. The median sorting time is about 2598 seconds which is less than twice of the random data sorting time.

TABLE VII
DAYTONA GRAY SORT RANDOM

Amount of Data	101.376 TB
Duplicate Keys	0
Input Checksum	*76046478aca758c4d9*
Trial 1	33 min 59.685 sec
Trial 2	33 min 45.209 sec
Trial 3	35 min 1.511 sec
Trial 4	34 min 36.252 sec
Trial 5	34 min 10.044 sec
Output checksum	*76046478aca758c4d9*
Median time	34 min 10.044 sec
Sort rate	2.97 TB/min

TABLE VIII
DAYTONA GRAY SORT SKEWED

Amount of Data	101.376 TB
Duplicate Keys	31152217581
Input Checksum	*760464fe84f3bb8567*
Trial 1	43 min 17.953 sec
Trial 2	43 min 54.304 sec
Trial 3	40 min 48.561 sec
Trial 4	39 min 43.702 sec
Trial 5	44 min 1.277 sec
Output checksum	*760464fe84f3bb8567*
Median time	43 min 17.953 sec
Sort rate	2.34 TB/min

D. Minute Sort Indy Results

For the Indy variant of Minute Sort, we sorted 5249.6 GB data in 59.859 seconds, which is the median time of 17 consecutive runs.

Specifically, we use 386 nodes of the cluster. Each node hosts about 13.6GB data. The input checksum is *61c8003b9d8c8b40d* without duplication. We've sorted this data in seventeen consecutive trials, and the running times reported by *time* utility are listed in Table IX. The output checksums are also *61c8003b9d8c8b40d*. There is no intermediate data for Minute Sort.

E. Minute Sort Daytona Results

For the Daytona variant of Minute Sort, we sorted 3686.4GB random data in 57.675 seconds which is the median time of 17 consecutive runs. The sorting time for skewed data is less than twice of the random data.

Specifically, we use 384 nodes of the cluster. Each node hosts 9.6GB input data. The input checksum is *44aa164cc14340759* without duplications. We've sorted this data in seventeen consecutive trials, and the running times reported by *time* utility are listed in Table X. The outputcheck-sum is also *44aa164cc14340759*. There is no intermediate data for Minute Sort.

TABLE IX
INDY MINUTE SORT

Amount of Data	5249.6 GB
Duplicate Keys	0
Input Checksum	*61c8003b9d8c8b40d*
Trial 1	0 min 58.910 sec
Trial 2	0 min 57.136 sec
Trial 3	0 min 59.537 sec
Trial 4	0 min 57.855 sec
Trial 5	0 min 58.142 sec
Trial 6	1 min 1.843 sec
Trial 7	0 min 59.859 sec
Trial 8	1 min 4.309 sec
Trial 9	1 min 16.645 sec
Trial 10	0 min 58.581 sec
Trial 11	1 min 2.380 sec
Trial 12	0 min 59.456 sec
Trial 13	1 min 1.352 sec
Trial 14	1 min 0.455 sec
Trial 15	1 min 4.131 sec
Trial 16	1 min 4.793 sec
Trial 17	0 min 57.127 sec
Output checksum	*61c8003b9d8c8b40d*
Median time	0 min 59.859 sec

TABLE X
DAYTONA MINUTE SORT RANDOM

Amount of Data	3686.4 GB
Duplicate Keys	0
Input Checksum	*44aa164cc14340759*
Trial 1	0 min 57.299 sec
Trial 2	0 min 55.581 sec
Trial 3	0 min 59.408 sec
Trial 4	0 min 57.675 sec
Trial 5	0 min 59.146 sec
Trial 6	0 min 56.590 sec
Trial 7	0 min 55.557 sec
Trial 8	0 min 57.937 sec
Trial 9	0 min 59.300 sec
Trial 10	0 min 57.017 sec
Trial 11	0 min 59.858 sec
Trial 12	0 min 51.672 sec
Trial 13	0 min 59.145 sec
Trial 14	1 min 1.159 sec
Trial 15	0 min 59.053 sec
Trial 16	0 min 56.647 sec
Trail 17	0 min 54.610 sec
Output checksum	*44aa164cc14340759*
Median time	0 min 57.675 sec

We also prepared skewed dataset and run DeepSort using 384 nodes of the cluster, totaling 3686.4 GB of skewed data with a checksum of *44aa2f5711f86b65f*. There are 66369138 duplicate keys in the input. We've also sorted the skewed data in seventeen consecutive trials, and the running times reported by *time* utility are listed in Table XI. The output checksums are also *44aa2f5711f86b65f*, and the median sorting time is 91.157 seconds, which is less than twice of the random input sorting time.

TABLE XI
DAYTONA MINUTE SORT SKEW

Amount of Data	3686.4 GB
Duplicate Keys	66369138
Input Checksum	*44aa2f5711f86b65f*
Trial 1	1 min 33.686 sec
Trial 2	1 min 23.333 sec
Trial 3	1 min 33.077 sec
Trial 4	1 min 51.229 sec
Trial 5	1 min 25.131 sec
Trial 6	1 min 30.805 sec
Trial 7	1 min 27.761 sec
Trial 8	1 min 33.522 sec
Trial 9	1 min 33.351 sec
Trial 10	1 min 24.555 sec
Trial 11	1 min 31.157 sec
Trial 12	1 min 28.696 sec
Trial 13	1 min 27.028 sec
Trial 14	1 min 26.570 sec
Trial 15	1 min 31.841 sec
Trial 16	1 min 33.769 sec
Trial 17	1 min 39.421 sec
Output checksum	*44aa2f5711f86b65f*
Median time	1 min 31.157 sec

V. ACKNOWLEDGMENT

DeepSort results are made possible with the strong support of all members in the Cloud Research Lab of Samsung Research America, Silicon Valley. Specifically, Lohit Giri, Venko Gospodinov, Hars Vardhan, and Navin Kumar have helped with the infrastructure provision. Minsuk Song, Jeehoon Park, Jeongsik In, and Zhan Zhang have helped with the networking. Hadar Isaac and Guangdeng Liao have advised on the sorting design. Thomas Phan have helped with the writing. We would like to appreciate Sort Benchmark committee members (Chris Nyberg, Mehul Shah, and Naga Govindaraju) for valuable comments.

REFERENCES

[1] C. Nyberg, M. Shah, and N. Govindaraju, "Sort FAQ (14 March 2014)," http://sortbenchmark.org/FAQ-2014.html, 2014, [Online].

[2] J. Huang and Y. Chow, "Parallel sorting and data partitioning by sampling," in *IEEE Computer Society's Seventh International Computer Software and Applications Conference (COMPSAC'83)*, 1983, pp. 627–631.

[3] T. Graves, "GraySort and MinuteSort at Yahoo on Hadoop 0.23," http://sortbenchmark.org/Yahoo2013Sort.pdf, 2013.

[4] A. Rasmussen, G. Porter, M. Conley *et al.*, "Tritonsort: A balanced large-scale sorting system," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011, p. 3.

[5] "Heap sort," http://www.opensource.apple.com/source/Libc/Libc-167/stdlib.subproj/heapsort.c, 1999, [Online].

[6] C. Kim, J. Park, N. Satish *et al.*, "CloudRAMSort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 841–850.

[7] J. S. Plank and K. M. Greenan, "Jerasure: A library in C facilitating erasure coding for storage applications – version 2.0," University of Tennessee, Tech. Rep. UT-EECS-14-721, January 2014.