FuxiSort

Jiamang Wang, Yongjun Wu, Hua Cai, Zhipeng Tang, Zhiqiang Lv,
Bin Lu, Yangyu Tao, Chao Li, Jingren Zhou, Hong Tang

Alibaba Group Inc

{jiamang.wang, yongjun.wyj, hua.caihua, zhipeng.tzp, zhiqiang.lv,
bin.lub, yangyu.taoyy, li.chao, jingren.zhou, hongtang}@alibaba-inc.com

Overview

Apsara is a large-scale general-purpose distributed computing system developed at Alibaba Cloud Computing Inc (also called Aliyun). It is responsible for managing cluster resources within a datacenter, as well as scheduling parallel execution for a wide range of distributed online and offline applications. It is the common foundation of a majority of public cloud services offered by Aliyun, and supports all data processing workload within Alibaba as well. Some key components of the Apsara system include: (a) a distributed file system, called **Pangu**; (b) a distributed computation framework, called **Fuxi**, for cluster resource management and job scheduling; and (c) a parallel DAG-based data computation framework on top of Fuxi to process large datasets. Apsara is mostly written in C/C++ to achieve high performance and efficiency. More details about Apsara can be found in [1].

Apsara has been deployed on hundreds of thousands of physical servers across tens of data centers at Aliyun, with the largest clusters consisting of more than 5000 servers each. Any DAG-based distributed data processing can be implemented as a Fuxi job, including simple Map/Reduce jobs and more sophisticated machine learning jobs. Source inputs and outputs, along with intermediate data are placed in Pangu, with a variety of replication and locality configurations to achieve the optimal balance between performance and reliability in order to accommodate varying requirements from different applications.

In this paper, we introduce FuxiSort, a distributed sort implementation on top of Apsara, and present our Daytona GraySort and MinuteSort results with details in technical implementation, performance improvements, and efficiency analysis. FuxiSort is able to complete the 100TB Daytona GraySort benchmark in **377** seconds on random non-skewed dataset and **510** seconds on skewed dataset, and Indy GraySort benchmark in **329** seconds. For the Daytona MinuteSort benchmark, FuxiSort sorts **7.7** TB in **58** seconds on random non-skewed dataset and **79** seconds on skewed dataset. For the Indy MinuteSort benchmark, FuxiSort sorts **11** TB in **59** seconds.

System Configuration

Before describing the overall architecture and detailed implementation, we present our system configuration as follows.

1. **Hardware**

   We run FuxiSort in a relatively homogeneous cluster with a total of 3377 machines, classified into 2 groups.

- 3134 machines
  - 2 * Intel(R) Xeon(R) E5-2630@2.30GHz (6 cores per CPU)
  - 96GB memory
  - 12 * 2TB SATA hard drives
  - 10Gb/s Ethernet, 3:1 subscription ratio
- 243 machines
  - 2 * Intel(R) Xeon(R) E5-2650v2@2.60GHz (8 cores per CPU)
  - 128GB memory
  - 12 * 2TB SATA hard drives
  - 10Gb/s Ethernet, 3:1 subscription ratio

2. Software

- OS: Red Hat Enterprise Linux Server release 5.7
- Apsara distributed computing system
- FuxiSort implemented on top of the Apsara
- Occasional background workloads (the cluster is shared with multiple groups, and we choose relatively idle periods of the cluster to conduct our experiments).

FuxiSort Overview

In this section, we present an overview of the FuxiSort design on top of the Apsara system. The same implementation is used to perform both GraySort and MinuteSort, with various system optimizations outlined in the next section.

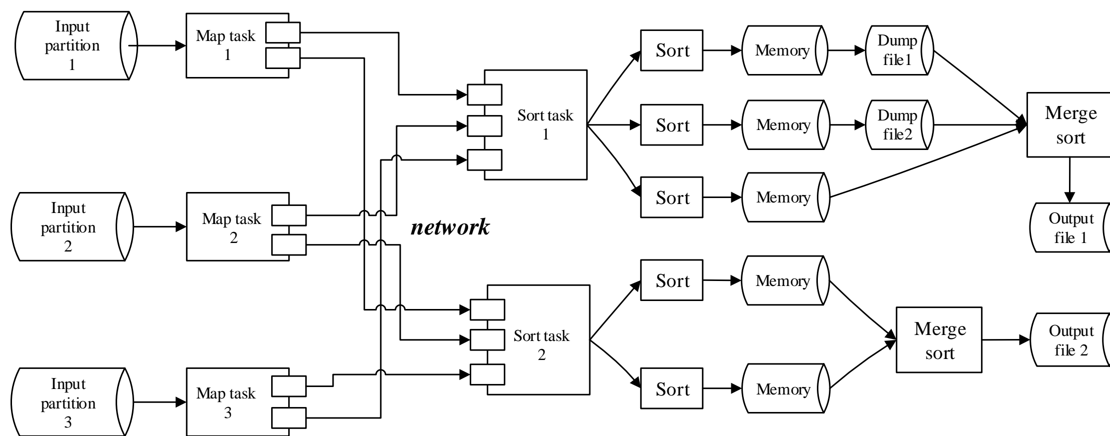

Figure 1. An Overview of FuxiSort

We first perform a sampling of the input to determine the range partition boundaries and divide the rest of the sorting process into two phases, *map* and *sort*, as shown in Figure 1. Each phase is carried out with multiple parallel tasks.

During the map phase, each input data segment is read from local disks (via Pangu Chunkserver daemon) and range partitioned by a map task. Each partitioned result of a map task is then transferred via network to the corresponding sort task.

During the sort phase, each sort task repeatedly collects records from the map tasks, and keeps

them in memory. It periodically performs an in-memory sort when its buffer becomes full, followed by writing the sorted data into temporary files (also stored on Pangu without replication). After all records are collected, it combines all the previously sorted data both in memory and temporary files using a merge sort and outputs the final result in Pangu. FuxiSort finishes with a number of range-partitioned and internally-sorted data files as the final result by the time all sort tasks finish.

Implementation and Optimizations

In this Section, we describe the implementation details of FuxiSort and explain several key performance optimizations, including data generation, sampling, I/O optimization, pipelined and overlapped execution, and network communications. Finally, we present additional improvements for the MinuteSort benchmark.

1. Input data generation

We prepare the input datasets in either random or skewed distribution using gensort, and copy them to Pangu. Each input dataset is split to multiple segments. The number of segments matches the number of map tasks so that each task processes one data segment. In addition, each input dataset is two-way replicated by Pangu. That is, every segment has two copies. Each copy of a segment consists of a number of Pangu chunks distributed evenly across all the disks on one machine. Each Pangu chunk is stored as a local file system file by Pangu plus associated metadata (Pangu manages individual disks directly to maximize parallel IO bandwidth instead of bundling them via RAID). All segments for a dataset are uniformly distributed among all machines in a balanced fashion. We use 20000 segments for the 100TB dataset in GraySort and 3400 segments for the datasets in MinuteSort. As described below, the output data are also two-way replicated in a similar way.

2. Input sampling

In order to discover data distribution and reduce data skew in the sort phase, we sample the input of Daytona Sort to generate the range partition boundaries so that each partition consists of a relatively equal amount of data to process.

Given that the input data is split into X segments, we randomly select Y locations on each segment, and read from each location Z continuous records. In this way, we collect $X * Y * Z$ sample records in total. We then sort these samples and calculate the range boundaries by dividing them into S range partitions evenly, in which S represents the number of sort tasks.

For the 100TB Daytona GraySort benchmark, with 20,000 data segments, we select 300 random locations on each segment, and read one record on each location. As a result, we collect 6,000,000 samples, sort, and divide them into 20000 ranges, same as the number of sort tasks. The calculated range boundaries are used by each map task to partition records in the map phase. The entire sampling phase takes approximately 35 seconds. For the Daytona MinuteSort benchmark, with 3350 data segments, we select 900 random locations on each segment, and read one record on each location. A total number of 3,015,000 samples are collected, sorted, and then divided to 10050 ranges. The entire sampling phase takes about 4 seconds. The sampling step is skipped for Indy sort benchmarks.

3. I/O dual buffering

We employ the standard dual buffering technique for I/O operations on each Pangu file.

Specifically, FuxiSort processes data in one I/O buffer while instructing Pangu to carry out I/O operations with the other buffer. The roles of the two buffers switch periodically in order to guarantee both I/O and data processing to be done in parallel, thereby significantly reducing the task latency.

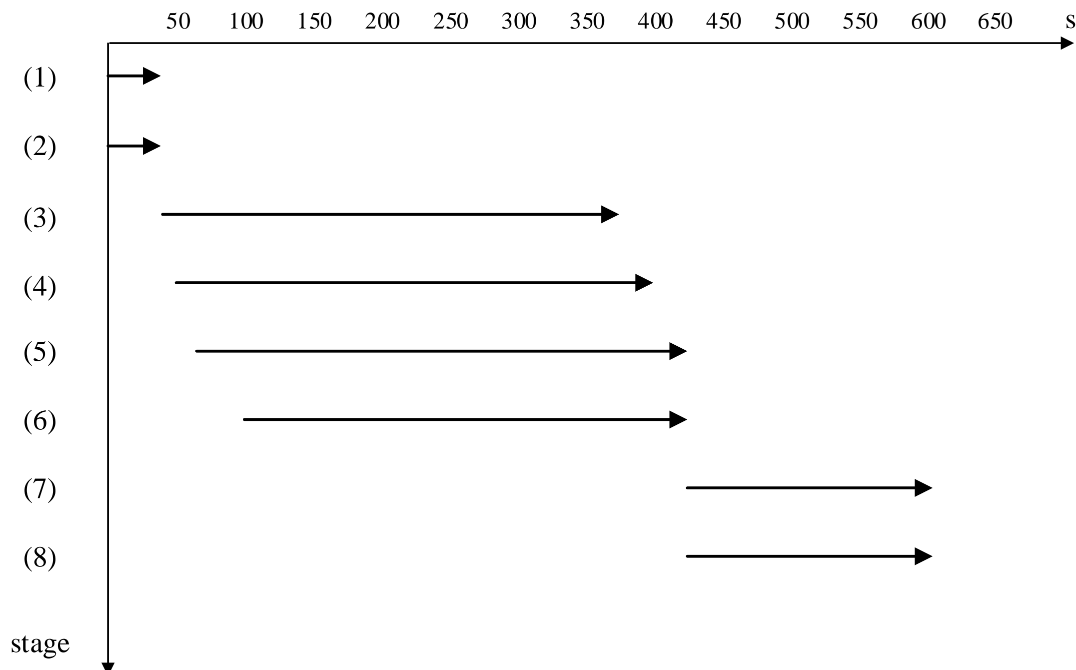

Figure 2. FuxiSort Latency Breakdown

4. Overlapped execution

As shown in Figure 2, to further reduce the overall latency, we break each phase into several small steps and overlap the execution of all the steps as much as possible during the process. These steps are listed below.

(1) Data sampling
(2) Job startup
(3) Map task reads input
(4) Map task sends data to sort task
(5) Sort task receives data
(6) Sort task sorts in-memory data and dumps to temporary files if necessary
(7) Sort task combines in-memory data and temporary files by merge sort
(8) Sort task writes final output

We parallelize the execution of data sampling and job startup, the latter of which dispatches all the task processes, and performs other bookkeeping chores such as collecting the network addresses of all the sort tasks, and informing the responding map tasks. When step (1) finishes, the sampling program stores the partition boundaries into Pangu and generates another flag file for notification. Once dispatched, all the map tasks wait for the notification by periodically checking the existence of the flag file. Once the partition boundaries become available, the map tasks immediately read them and perform partitioning accordingly.

Naturally, steps (3) (4) and (5) are pipelined during the map phase, and steps (7) and (8) pipelined

during the sort phase. In step (6), had the sorting and dumping been deferred until the entire memory assigned to a task is filled up, step (5) would have been blocked because the memory is still fully occupied during sorting and no memory is left for receiving new data. To mitigate this problem, we parallelize (5) and (6) by starting the sorting once a certain amount of memory is occupied, so that (6) is brought forward and (5) is not blocked. We start the merge and dump of sorted in-memory data to temporary files when memory is fully occupied. Obviously the lower the memory limit to start sorting, the earlier step (6) would start. In our experiments, as a tradeoff between concurrency and merge sort efficiency, we start (6) when approximately 1/10 of memory assigned to a task is occupied by the received data. As a result, we are able to parallelize I/O and computation without noticeable delay, at the expense of more temporary files to be merged later, without introducing significant overhead in our cases.

Figure 2 shows the elapsed time for each step and their overlapped execution.

5. Handling network communication

There is significant network communication traffic between map tasks and sort tasks. Each network packet arrival raises an interrupt to CPUs to process. If the interrupt handling is set to be affinitive to a specific CPU core, then the interrupt handling may be delayed when the same CPU is busy with sorting, resulting in request time-out and, even worse, packet loss. By balancing the load of network interrupts among all CPU cores via setting the "smp_affinity" system option, the rates of timeout events and packet loss are significantly decreased.

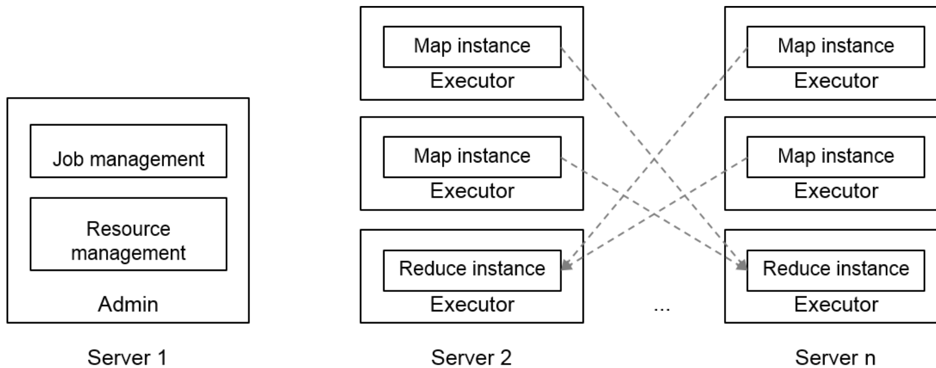

Figure 3. Real-time mode architecture

6. Additional improvements for MinuteSort

As the elapsed time of MinuteSort is required to be limited up to 60 seconds, the scheduling overhead of ordinary batch data processing becomes non-negligible. To mitigate the overhead, we run MinueSort in Fuxi's Real-time Job mode. The Real-time Job mode is developed to achieve high performance with low scheduling latency and in-memory computation. Figure 3 illustrates the architecture of Real-time Job mode. In a typical production environment, the system is setup as part of the cluster deployment process, and runs as a long-living service by creating a pool of persistent worker processes on each machine. Users can later submit a variety of jobs to the scheduler of the system and be informed the status of job execution. To comply with the requirements of sort benchmark competition, where all setup and shutdown activities directly related to sorting must be included in the reported time, we modified our client program to start the worker process pool before we submit the MinuteSort job, and stop the workers after job completion, and we include the time for starting/stopping the worker pool in the reported elapsed times.

The target scenario of Real-time Job mode is latency-sensitive data processing with medium-

sized dataset (no more than 10TB), since all records are likely to be cached in main memory, including both input and output data. In our experiments, we only use Real-time Job mode for MinuteSort benchmark.

Benchmark Results

We present our FuxiSort results for both 100TB GraySort and MinuteSort benchmarks below. Apsara has been deployed in our production environment for over six years, with non-interruptive progressive upgrade to avoid impacting jobs or online services during the process. FuxiSort is in fact a lightweight specialization of our general-purpose DAG-based data computation framework, and most of the aforementioned optimizations are inside Apsara and the DAG-based data computation. Many of our production jobs using the same framework process much larger datasets than 100TB, and it is fairly common for a job to run multiple hours (Fuxi and Pangu has built-in mechanisms to recover from a variety of hardware/software failures).

For all benchmark runs described below, input/output/temporary data are uncompressed. We ensure data are not buffered in OS page cache by running a data purge job that randomly reads from local file system before each benchmark run. Pangu ensures data durability by flushing data to disk when an output file stream is closed.

1. **GraySort**

Table 1 shows the performance of FuxiSort in different GraySort benchmarks. Daytona sort includes sampling on start while Indy sort does not. For Daytona, the final output data are stored with a replication factor of two, same as the input data. Pangu ensures that the two copies are replicated to different machines (and racks). For Indy, the final output data are stored with only one copy. Overall, FuxiSort is able to perform Indy GraySort against the randomly distributed input dataset in 329 seconds, Daytona GraySort against the randomly distributed input dataset in 377 seconds, and Daytona GraySort against the skewed input dataset in 510 seconds. They are equivalent to a sorting throughput of 18.2 TB, 15.9 TB and 11.7 TB per minute, respectively.

Table 1. GraySort benchmark results

	Indy GraySort Random	**Daytona GraySort Random**	**Daytona GraySort Skewed**
Input Size (bytes)	100,000,000,000,000	100,000,000,000,000	100,000,000,000,000
Input File Copy	2	2	2
Output File Copy	1	2	2
Time (sec)	329	377	510
Effective Throughput (TB/min)	18.2	15.9	11.7
Number of Nodes	3377	3377	3377
Checksum	746a51007040ea07ed	746a51007040ea07ed	746a50ec9293190d87
Duplicate Keys	0	0	30285373571

2. **MinuteSort**

We also run FuxiSort using Real-time Job mode for the MinuteSort benchmark. With 15 trials, the median time of sorting 7.7 TB is 58 seconds for uniformly-distributed data, and 79 seconds for

the Daytona category with skewed data distribution. For the Indy category, FuxiSort is able to sort 11 TB randomly distributed input dataset in 59 seconds. The experiment details are shown in table 2. Similar to GraySort, all input and output datasets are written in 2 copies, stored in two different machines.

Table 2. Daytona MinuteSort benchmark results

	Indy MinuteSort Random	**Daytona MinuteSort Random**	**Daytona MinuteSort Skewed**
Input Size	11 TB	7.7 TB	7.7 TB
Records	109951164050	76965814500	76965814500
Trial 1	58	59	85
Trial 2	59	61	81
Trial 3	59	61	78
Trial 4	61	60	80
Trial 5	60	61	80
Trial 6	60	62	81
Trial 7	60	57	79
Trial 8	61	57	77
Trial 9	59	58	78
Trial 10	59	57	77
Trial 11	61	60	79
Trial 12	58	58	79
Trial 13	59	58	80
Trial 14	59	57	79
Trial 15	58	58	80
Median Time(sec)	**59**	**58**	**79**
Nodes	3377	3377	3377
Checksum	ccccb43e2a7b4183c	8f5c15cd7fc7a47b2	8f5c2f6f4068d6876
Duplicate Keys	0	0	265727609

Acknowledgements

We would like to thank the members of the Fuxi team for their contributions to the Fuxi distributed computation framework, on top of which FuxiSort is developed and implemented. We would also like to thank the entire Apsara team at Aliyun for their support and collaboration.

Reference

[1] Zhang, Z., Li, C., Tao, Y., Yang, R., Tang, H., & Xu, J. (2014). Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In Proceedings of the VLDB Endowment, 2014, 7(13), 1393-1404.