# Exoshuffle-CloudSort

FRANK SIFEI LUAN*, UC Berkeley
STEPHANIE WANG, UC Berkeley and Anyscale
SAMYUKTA YAGATI, UC Berkeley
SEAN KIM, UC Berkeley
KENNETH LIEN, UC Berkeley
ISAAC ONG, UC Berkeley
TONY HONG, UC Berkeley
SANGBIN CHO, Anyscale
ERIC LIANG, Anyscale
ION STOICA, UC Berkeley and Anyscale

## 1 INTRODUCTION

We present Exoshuffle-CloudSort, a sorting application running on top of Ray using the Exoshuffle architecture [4]. Exoshuffle-CloudSort runs on Amazon EC2, with input and output data stored on Amazon S3. Using 40× i4i.4xlarge workers, Exoshuffle-CloudSort completes the 100 TB CloudSort Benchmark (Indy category [6]) in 5378 seconds, with an average total cost of $97.

## 2 IMPLEMENTATION

### 2.1 Overview

Exoshuffle-CloudSort is a distributed futures program running on top of Ray, a task-based distributed execution system. The program acts as the control plane to coordinate map and reduce tasks; the Ray system acts as the data plane, responsible for executing tasks, transferring blocks, and recovering from failures.

Exoshuffle-CloudSort implements a two-stage external sort algorithm. The first stage is map and shuffle. Each map task reads an input partition, sorts it, and partitions the result into $W$ output partitions, each sent to a merger on a worker node. A merger receives $W$ map output partitions, merges and sorts them, and further partition the result into $R/W$ output partitions, all of which are spilled to local disk.

The second stage is reduce. Once the map and shuffle stage finishes, each reduce task reads $W$ shuffled partitions, merges and sorts them, and writes the final output partition.

For the 100 TB CloudSort Benchmark, we set the following parameters:

- Total data size is 100 TB.
- Number of input partitions $M = 50\,000$. Each input partition is 2 GB.

*Author's address: lsf@berkeley.edu, 465 Soda Hall, Berkeley, CA, USA.

- Number of workers $W = 40$.
- Number of output partitions $R = 25\,000$.

### 2.2 Preparation

The first step in Exoshuffle-CloudSort is to compute the partition boundary values. For a sort record with 10-byte key, we view the first 8 bytes as a 64-bit unsigned integer partition key. We partition the key space $[0, 2^{64} - 1)$ into $R = 25\,000$ equal ranges, such that all the records within a key range should be sent to one reducer.

Every $R_1 = R/W = 625$ reducer ranges are combined into a worker range, and records in each worker range will be sent to one worker node. This yields $W = 40$ equally-partitioned worker ranges.

### 2.3 Map and Shuffle Stage

In the map and shuffle stage, Exoshuffle-CloudSort schedules the $M = 50\,000$ map tasks onto all worker nodes. In our experiments we set the map parallelism, i.e. the number of map tasks running on a single worker node, to be 3/4 of the total number of vCPU cores. Extra tasks are queued on the driver node. Whenever a worker node finishes a map task, the driver assigns a new task from the queue to this node.

In a map task, we first download the input partition from S3. We then sort the input data in memory, then partition it into $W = 40$ slices. Each slice is eagerly sent to a merge controller on each worker. The map task returns when all slices are sent.

On the receiving end, the merge controller accumulates the map blocks in memory until a threshold is reached. We set the threshold to 40 blocks, or about 2 GB of data. Once the threshold is reached, the controller launches a merge task to merge the already-sorted map blocks, and further partitions it into $R_1 = 625$ merged blocks, each corresponding to a reduce task on this node. These blocks are spilled to the local SSD for use by the reducers.

The merge parallelism is set to be the same as the map parallelism. When the number of merge tasks reaches the maximum parallelism, and the merge controller's in-memory buffer is filled up, it will hold off acknowledging the receipt of a map block until a merge task finishes and a new merge task can launch. This effectively creates back pressure to the map task scheduler to ensure the map, shuffle, and merge progresses are in sync.

In our experiments, the average map task duration is 24 seconds; 15 seconds are used for downloading input data. The average shuffle time (i.e. time to send and receive blocks) is 7 seconds. The merge task takes 17 seconds on average.

### 2.4 Reduce Stage

Once all map and merge tasks finish, Exoshuffle-CloudSort enters the reduce stage. Each reduce task loads $R_1 = 625$ from the local SSD, merges them, and uploads the sorted output partition to S3. The merging

of sorted records is implemented using a heap. In our experiments, each reduce task takes 22 seconds on average.

### 2.5 The Execution System

A highlight of the Exoshuffle architecture is that the application program only implements the control plane logic, and the distributed futures system, Ray, handles execution. This is reflected in Exoshuffle-CloudSort. Here is an incomplete list of features provided by Ray that we take "for free":

- Task scheduling: The program specifies when and where to schedule tasks; the system handles the RPC, serialization, and other bookkeeping.
- Network transfer: The program instructs data to be transferred by passing distributed futures as task arguments; the system implements high-performance network transfer.
- Memory management and disk spilling: The program manipulates data references in a virtual, infinite address space; the system uses reference counting to manage distributed memory, spills objects to local disks when memory is low, and restores objects from local disks when they are needed.
- Pipelining of network and disk I/O: The network transfer, spilling and recovery of objects are transparent to the application and are performed asynchronously. For example, the system shuffles map output blocks while other map and merge tasks are running; it spills merge task output to disk while other merge tasks are executing, and it restores merged blocks while reduce tasks are executing.
- Fault tolerance: this is transparent to the application: the system automatically retries the operation when it encounters network failures and worker process failures.

For more details, we refer the reader to the Ray Architecture Whitepaper [7], the ownership design for distributed futures systems [8], and the Exoshuffle paper [4].

### 2.6 Source Code

Exoshuffle-CloudSort is implemented in about 1000 lines of Python, and about 300 lines of C++. The C++ component implements two functionalities: sorting and partitioning records, and merging sorted record arrays. Exoshuffle-CloudSort runs on top of Ray, which is implemented in Python and C++. All of Exoshuffle-CloudSort's source code is available at https://github.com/exoshuffle/cloudsort.

## 3 EVALUATION

### 3.1 Environment Setup

We run Exoshuffle-CloudSort on AWS on a compute cluster configured as follows:

- 1× r6i.2xlarge master node. This node runs on 8 cores of an Intel Xeon 8375C CPU at 2.9 GHz, and 64 GiB memory.

- 40× i4i.4xlarge worker nodes. Each node runs on 16 cores of an Intel Xeon 8375C CPU at 2.9 GHz, and 128 GiB memory. Each node has a directly-attached 3.75 TB AWS Nitro NVMe SSD.
- Each node is attached with a 40 GiB Amazon EBS General Purpose SSD (gp3) volume.

The software stack is configured as follows:

- Ubuntu 22.04.1 LTS, Linux kernel version 5.15.0-1022-aws.
- XFS 5.13.0 filesystem.
- Intel oneAPI DPC++/C++ Compiler 2022.2.0.20220730.
- Python 3.9.13.
- Ray 2.1.0.

We measure the raw system I/O performance on the worker nodes using standard benchmarking tools:

- Network bandwidth: 25 Gbps between nodes, benchmarked with iperf.
- SSD: 2.9 GB/s read, 2.2 GB/s write, benchmarked with fio.

For storage, we use 40 buckets on Amazon S3 and randomly distribute the input and output partitions across the buckets.

### 3.2 Benchmark Setup

*Generating Input.* We use `gensort` version 1.5 as provided by the Sort Benchmark committee [5]. We run the command `gensort -c -b{offset} {size} {path}` to generate each partition. `{size}` is fixed at $P = 20\,000\,000$ such that each partition is exactly 2 GB. `{offset}` takes the values $\{i \cdot P : 0 \leq i < M\}$ where the number of input partitions $M = 50\,000$. `{path}` is a unique path in tmpfs. `-c` provides data checksum for validation. After generating an input file, we randomly choose a bucket and upload the partition to S3. We use Ray to schedule the 50 000 input generation tasks to all 40 worker nodes. The result is aggregated as an input manifest file, saved for use by Exoshuffle-CloudSort to locate the sort input.

*Validating Output.* Exoshuffle-CloudSort produces an output manifest file containing the bucket and keys of each output partition on S3. In each validation task, we first download the output partition to tmpfs, then run the command `valsort -o {sumpath} {path}` to validate the ordering of records in each partition. We use Ray to schedule the 25 000 output validation tasks to all 40 worker nodes. We concatenate the contents of the summary files from each validation task, then run `valsort -s` to validate the total ordering, and generate the total output checksum. Finally, we compare the output checksum with the input checksum to verify data integrity.

### 3.3 Experimental Results

*3.3.1 Job Completion Time.* On November 10, 2022, we ran the 100 TB CloudSort Benchmark in the AWS US West (Oregon, us-west-2) region with the setup described above. We first generated the input data on Amazon S3, then ran Exoshuffle-CloudSort 3 times, each followed by a validation step. All 3 runs succeeded

with the same output checksum as the input, indicating all bytes are preserved in the sort. Table 1 reports the job completion times of each run. The average job completion time is 5378 seconds, or 1.4939 hours.

| Run | Map & Shuffle Time | Reduce Time | Total Job Completion Time |
|---|---|---|---|
| #1 | 3509 s | 1852 s | 5361 s |
| #2 | 3496 s | 1852 s | 5348 s |
| #3 | 3520 s | 1906 s | 5426 s |
| Average | 3508 s | 1870 s | 5378 s |

Table 1. Job completion times of Exoshuffle-CloudSort on the 100 TB CloudSort Benchmark.

Figure 1 shows the system utilizations of all worker nodes in the compute cluster during run #1 of the 100 TB CloudSort Benchmark.


Fig. 1. Cluster utilization during run #1 of the 100 TB CloudSort Benchmark. Each thick line represents the median system utilization of all worker nodes; the highest and lowest lines represent the maximum and minimum utilization among all worker nodes, respectively.

*3.3.2 Total Cost of Ownership.* The total job cost comprises of two parts: compute cost (Amazon EC2), and the storage cost (Amazon S3). The storage cost is further divided into data storage cost and data access cost.

*Compute Cost.* The compute cost is calculated as the compute cluster's hourly cost times the job completion time. The total hourly cost is calculated as follows:

$$\begin{aligned}\text{Total Hourly Compute Cost} = {} & \text{Master Node Hourly Cost} \\ & + \text{Worker Node Hourly Cost} \times \text{Number of Workers} \\ & + \text{EBS Volume Hourly Cost} \times (\text{Number of Workers} + 1)\end{aligned} \tag{1}$$

We obtain the compute instance hourly costs from the Amazon EC2 on-demand pricing information [2]. For EBS, we use the Amazon EBS monthly price [1] divided by the average number of hours in a month ($\frac{365\times24}{12} = 730$) as the hourly price. The hourly cost of a 40 GiB gp3 volume is $\$0.08/730 \times 40 = \$0.0044$. Now we plug the cost variables into Equation (1):

- Master node (r6i.2xlarge) hourly cost is $0.504.
- Worker node (i4i.4xlarge) hourly cost is $1.373.
- Number of workers is 40.
- EBS volume hourly cost is $0.0044.

Hence, the total hourly compute cost is $55.6044. We multiply this hourly cost by the job completion time of 1.4939 hours to obtain the total compute cost of $83.0674.

*Data Storage Cost.* The storage cost comprises of data storage cost and data access cost. We first consider the data storage cost. Amazon S3 employs a pay-as-you-go pricing model, i.e. the user does not need to provision storage capacity ahead of time, and only pays for the storage cost of objects based on their sizes and storage duration. Amazon S3 charges $0.023 per GB-month for the first 50 TB, then $0.022 per GB-month for the next 450 TB [3]. Since the total data size is 100 TB, we take the average price between the first two tiers, i.e. $0.0225 per GB-month, or $3.0822 per hour per 100 TB.

- Input: The storage cost of the 100 TB input data is simply the cost to store 100 TB for the duration of the sort: $\$3.0822 \times 1.4939 = \$4.6045$.
- Output: The 100 TB output data is uploaded to and stored on Amazon S3 during the reduce stage of the sort. We use the duration of the reduce stage as the storage time of the 100 TB output data. This is an over-estimation because the output partitions are uploaded as the reduce stage progresses, and therefore most of the 100 TB is stored on S3 for less time than the entire reduce stage duration. Table 1 shows the average reduce stage time is 1870 seconds, or 0.5194 hours. Hence we get the output storage cost: $\$3.0822 \times 0.5194 = \$1.6009$.

Adding up the input and output data storage cost, we get the total data storage cost: $6.2054.

*Data Access Cost.* We consider GET and PUT requests to Amazon S3. Exoshuffle-CloudSort downloads the 100 TB input data in 50 000 map tasks. Each map task downloads a 2 GB input partition in 16 MiB chunks, resulting in 120 GET requests per task, or 6 000 000 GET requests in total. Amazon S3 charges $0.0004 per 1000 GET requests [3]. Hence the total GET cost is $2.4000.

Exoshuffle-CloudSort uploads the output data in 25 000 reduce tasks. Each reduce task uploads approximately 4 GB data in 100 MB chunks, resulting in 40 PUT requests, or 1 000 000 PUT requests in total. Amazon S3 charges $0.005 per 1000 PUT requests [3]. Hence the total PUT cost is $5.0000.

The actual number of requests could be marginally higher due to request failures and retries, but the amount should be negligible. Hence, the total data access cost is $7.4000.

*Total Cost of Ownership.* Adding up the compute cost and storage cost, we get the total cost of ownership for the 100 TB CloudSort Benchmark: $96.6728. Table 2 presents a summary of the cost analysis.

| Service | Unit Price | Amount | Total Price |
|---|---|---|---|
| Compute VM Cluster | $55.6044 / hr | 1.4939 hours | $83.0674 |
| Data Storage (Input) | $3.0822 / hr | 1.4939 hours | $4.6045 |
| Data Storage (Output) | $3.0822 / hr | 0.5194 hours | $1.6009 |
| Data Access (Input) | $0.0004 / 1000 requests | 6 000 000 requests | $2.4000 |
| Data Access (Output) | $0.005 / 1000 requests | 1 000 000 requests | $5.0000 |
| Total | – | – | $96.6728 |

Table 2. Cost breakdown of Exoshuffle-CloudSort on the 100 TB CloudSort Benchmark.

## ACKNOWLEDGMENTS

This work is done in the Sky Computing Lab at UC Berkeley, sponsored by Astronomer, Google, IBM, Intel, Lacework, Nexla, Samsung SDS, and VMware. This work is done in collaboration with Anyscale.

## REFERENCES

[1] Amazon. 2022. *Amazon EBS High-Performance Block Storage Pricing*. Amazon Web Services. https://aws.amazon.com/ebs/pricing/

[2] Amazon. 2022. *Amazon EC2 On-Demand Instance Pricing*. Amazon Web Services. https://aws.amazon.com/ec2/pricing/on-demand/

[3] Amazon. 2022. *Amazon S3 Simple Storage Service Pricing*. Amazon Web Services. https://aws.amazon.com/s3/pricing/

[4] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, SangBin Cho, Eric Liang, and Ion Stoica. 2022. Exoshuffle: Large-Scale Shuffle at the Application Level. https://doi.org/10.48550/ARXIV.2203.05072

[5] Chris Nyberg. 2022. *Sort Benchmark Data Generator and Output Validator*. Ordinal Technology Corp. http://www.ordinal.com/gensort.html

[6] Mehul A. Shah, Amiato, and Chris Nyberg. 2014. CloudSort: A TCO Sort Benchmark. http://sortbenchmark.org/2014_06_CloudSort_v_0_4.pdf. (Accessed on 11/10/2022).

[7] Ray Team. 2022. *Ray v2 Architecture*. Anyscale. https://docs.google.com/document/d/1tBw9A4j62ruI5omIJbMxly-la5w4q_TjyJgJL_jN2fI/preview

[8] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. 2021. Ownership: A Distributed Futures System for Fine-Grained Tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Virtual, 671–686. https://www.usenix.org/conference/nsdi21/presentation/cheng