SchnellSort 2016

Greg Hogan
greg@apache.org

August 2016

1 Introduction

Apache Flink [2] is an open source platform for distributed stream and batch data processing. Flink programs are assembled from a fluent selection of map, reduce, and join transformations and interoperate within the Apache big data software stack. Each program is compiled, optimized, and executed in the Flink distributed runtime with non-blocking transformations operating concurrently. Flink places a particular focus on streaming but handles batch processing as the special case in which the input source is consumed.

Sorting is a foundational but in isolation a painfully simple task for modern big data frameworks. This report presents an exploration of CloudSort [6] using Apache Flink with an emphasis on documenting the process for others and our future selves.

The current CloudSort champion [8] ran on Amazon Web Services [1] and a follow-up report [7] anaylzed GraySort on Google Cloud Platform [14]. These and other prior benchmarks used persistent block storage [8] or ephemeral instance storage [7] [15]. The following Indy CloudSort explores the use and current limitations of object storage for persistent input and output datasets.

2 CloudSort on Public Clouds

Public clouds sell three kinds of storage:

- ephemeral instance storage (legacy Amazon EC2, Google local SSD)
- persistent block storage (Amazon EBS, Google Persistent Disk)
- persistent object storage (Amazon S3, Google Cloud Storage)

The CloudSort sort benchmark requires that the input and output datasets be written to persistent storage. Comparative pricing is provided for persistent storage in table 1 and ephemeral storage in table 2. A significant distinction is that persistent storage can be allocated by the gibibyte whereas ephemeral

Storage Type	Amazon Web Services			Google Cloud Platform		
	S3 Standard	S3 Reduced	EBS gp2	GCS Standard	GCS Reduced	Persistent Disk
$/GiB-month	0.03	0.024	0.10	0.026	0.02	0.17
$/100 TB-hour	3.89	3.11	12.94	3.37	2.59	21.99

Table 1: Cost for cloud persistent storage

	Amazon Web Services						Google Cloud Platform			
Instance type	c4	c3	m4	r3	i2	x1	highcpu	standard	highmem	Local SSD
Price ($/hr)	1.675	1.68	2.394	2.66	6.82	13.338	1.20	1.60	2.00	0.113
Memory (GiB)	60	60	160	244	244	1952	28.8	120	208	-
Memory ($/100 TB-hr)	baseline	-	669.39	498.41	2603.37	571.49	baseline	408.48	423.20	-
SSD (GB)	-	640	-	640	6400	3840	-	-	-	375
SSD ($/100 TB-hr)	-	0.79	-	153.91	80.40	303.73	-	-	-	30.14

Table 2: Cost for cloud ephemeral storage

storage is a fixed instance allotment (excepting Google local SSD, which is allocated as a multiple of 375 GB disks). This means that clusters using persistent storage can be of nearly any size whereas using ephemeral storage may require a very large cluster and associated I/O. Not reflected in this table is SSD performance, which is very good for Google local SSD and very poor for Amazon c3 instances.

CloudSort proceeds in two phases. In the first phase data is read from persistent storage and shuffled across the network. In the second phase the output data is written to persistent storage. In the shuffle each byte transits two network interfaces. Records are sorted in the first phase before spilling to disk. Thus the cluster I/O and CPU requirements are much greater in the first phase compared with the second phase.

Phase 1:

- read input from persistent storage
- split into records and range partition
- shuffle records to remote worker
- spill records

Phase 2:

- read spilled records
- merge-sort and count duplicate keys
- write output to persistent storage

	Amazon Web Services		Google Cloud Platform
	c4 w/EBS	c3 w/Instance Storage	n1-standard-8 w/Local SSD
Instance count	98	164	280
Instance disk (GB)	1000	640	375
Instance I/O (Gbps)	10 + 4 EBS	10	16
Phase 1 maximum I/O (MB/s)	416	380	272
Phase 1 total I/O (GB/s)	40.8	62.3	76.1
Phase 1 minimum time (s)	2453	1605	1314
Phase 2 maximum I/O (MB/s)	800	480	390
Phase 2 total I/O (GB/s)	78.4	78.7	109.2
Phase 2 minimum time (s)	1276	1271	916
Overall minimum time (s)	3729	2876	2230
Instance cost ($/hr)	1.814	1.68	0.513
Minimum compute cost ($)	184.15	220.11	88.98

Table 3: Comparision of Performance and Compute Cost

Table 3 lists optimal cluster performance and cost of compute. The configurations include a 5% storage buffer to allow for filesystem metadata, unbalanced spill file output (even using a round-robin the first disk may receive an extra file), and slight partition skew.

These baseline costs do not include the cost of persistent storage for the input and output datasets, require consistent maximum network and disk I/O, and assume no delay when starting the cluster and transitioning between sort phases.

Increasing the number of nodes in a small cluster reduces the required per-instance storage. For AWS, the optimal cluster configurations in table 3 mark the minimum storage to maintain maximum I/O. Increasing cluster size may result in a faster sort but will not reduce the sort cost. For GCE the cost may be further reduced if an instance can drive full disk I/O with fewer CPUs or less memory.

3 FlinkSort

Code, artifacts, and results are available at flink-cloudsort [11]. All benchmarks were run with vanilla flink-1.1.1 [10] as the execution was not CPU bound. The code provides a custom IndyRecord implementation for 10-byte keys with 90-byte records. The partitioner and CRC32 for validation are adapted from Apache Hadoop [3].

There were three major challenges discovered during testing. First, profiling revealed that the Java implementation of SHA-1 used for SSL used 50% of the available CPU cycles. This is solved with intinsics provided by the upcoming release of Java 8 build 112 paired with a Skylake or newer genereration processor implementing the Intel SHA Extensions [16]. Since these processors may not

Benchmark	# Nodes	Average Time	Average Cost	Checksum	Duplicate Keys
Indy CloudSort	129	6799.57 s	$239.58	746a51007040ea07ed	0

Table 4: Benchmark Summary

be available from cloud providers for several users, the alternative was to pipe input and output from the AWS CLI [4].

The second hurdle was poor network performance starting with 16 x c4.8xlarge instances. This was alleviated by configuring Flink to configure Netty with larger network buffers, increasing the Linux default 4 MiB to 64 MiB, and increasing the number of Netty threads to equal the number of instance vcores (hyperthreads).

The third hurdle resulted from outlier performance when downloading from or uploading to Amazon S3. This was solved by killing and retrying transfers after a configurable timeout.

4 Benchmarks

The valsort checksum and number of duplicate keys for the 100 TB of non-skewed data are listed in table 4. As in [8], there were no duplicate keys found. Checksums for any number of gigabytes up to a petabyte can be processed from the provided CRC file [11] with the following python summation. This is useful for testing smaller quantities of data. gensort [12] runs fastest when writing to /dev/null.

The validation concatenation for the 400,384 output files is included in the flink-cloudsort repository [11].

```
head -n \${BLOCKS} /path/to/crc32 | python -c "import sys; \
  print hex(sum(int(l, 16) for l in sys.stdin))[2:].rstrip('L')"
```

The three runs in table 5 average $239.58. Each cluster used one master c4.4xlarge instance and 128 worker c4.4xlarge instances. Earlier tests were run with c4.8xlarge instances running two Flink TaskManagers each, one per NUMA domain. Cluster were launched in a placement group for maximum networking performance. When launched in a placement group each instance is throttled to 5 Gbps outside the cluster. Communication to Amazon S3 was consistently throttled to 4.50 Gbps. Early tests with large c4.8xlarge clusters did not perform as well as later tests run with c4.4xlarge instances, but as with all things ”cloud” it is nearly impossible to divine the reason.

For EBS storage, the master node was allocated a 50 GiB root partition. Worker nodes were allocated an 8 GiB root partition and three 255 GiB partitions for spilling intermediate data. The total allocation for spilled data was 105 TB.

The AWS S3 storage was computed by script (available as parse_bytes.py [11]) as storing the input dataset over the full runtime of the sort and the output

	Price	Run 1	Run 2	Run 3
Time		7133 s	6561 s	6706 s
AWS c4.4xlarge Instances		129	129	129
AWS c4.4xlarge Cost	$0.838/instance-hr	$214.20	$197.02	$201.38
AWS EBS gp2 GiB		98994	98994	98994
AWS EBS gp2 Cost	$0.10/GiB-mo	$27.25	$25.06	$25.62
AWS S3 Cost	$0.024/GiB-mo	$7.59	$7.09	$7.23
AWS S3 LIST, PUT		401,537	401,627	401,541
AWS S3 LIST, PUT Cost	$0.005 per 1,000	$2.01	$2.01	$2.01
AWS S3 GET		100,330	100,149	100,185
AWS S3 GET Cost	$0.004 per 10,000	$0.05	$0.05	$0.05
Total Cost		$251.10	$231.23	$236.39

Table 5: Benchmark Results

dataset from when the result output records were written to local memory. Flink JobManager and TaskManager statistics were collected (again, available in the repository) and each gauge value was processed from the previous timestamp.

These benchmarks were run using AWS S3 Reduced Redundancy Storage. This was not driven by cost or performance concerns but rather the consideration that creating and quickly deleting hundreds of terabytes should be performed with as light an impact as possible.

This Reduced Redundancy Storage satisfies the CloudSort requirements as described by the following description from Amazon. ”The RRS option stores objects on multiple devices across multiple facilities, providing 400 times the durability of a typical disk drive, but does not replicate objects as many times as standard Amazon S3 storage.” [5]

AWS S3 LIST requests return up to 1,000 results, requiring 100 requests for the 100,000 x 1 GB input files. The AWS S3 GET request count includes the extra requests for terminated downloads. The AWS S3 PUT requests are counted for the 400,384 x 256 MiB result files as well as the extra requests for terminated uploads.

It is critical to adjust the AWS CLI ”multipart_chunksize” to larger than 8 MiB. At the default size the cost of PUT requests for the 100 TB output is $62.50.

5 Running the sort

The following sections document the five phases to running flink-cloudsort on Amazon Web Services as benchmarked in this report.

5.1 Creating an Amazon Machine Instance

The custom AMI is created by launching the latest Amazon Linux AMI then applying the following commands. This installs required software, optimizes the system, and configures passwordless SSH.

```
sudo su
yum-config-manager --enable epel
yum update -y

# http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSPerformance.html
sed -i ’s/\(kernel .*\)/\1 xen_blkfront.max=256/’ /boot/grub/grub.conf

reboot
sudo su

yum install -y fio collectl ganglia-gmetad ganglia-gmond ganglia-web git htop iftop iotop pdsh \
  sysstat systemtap telnet xfsprogs
stap-prep

# optional: first download then install Oracle JDK
yum localinstall -y jdk-*.rpm && rm -f jdk-*.rpm

# optional: Amazon’s Linux AMI is not kept up-to-date
pip install --upgrade awscli

# install GNU Parallel
(wget -O - pi.dk/3 || curl pi.dk/3/ || fetch -o - pi.dk/3) | bash
rm -rf parallel*

# increase the number of allowed open files and the size of core dumps
cat <<EOF > /etc/security/limits.conf
* soft nofile 1048576
* hard nofile 1048576
* soft core unlimited
* hard core unlimited
EOF

cat <<EOF > /etc/pam.d/common-session
session required pam_limits.so
EOF

# mount and configure EBS volumes during each boot
cat <<EOF >> /etc/rc.local
mkdir -p /volumes
format_and_mount() {
    blockdev --setra 512 /dev/xvd\$1
    echo 1024 > /sys/block/xvd\$1/queue/nr_requests

    /sbin/mkfs.ext4 -m 0 /dev/xvd\$1
    mkdir /volumes/xvd\$1
    mount /dev/xvd\$1 /volumes/xvd\$1

    mkdir /volumes/xvd\$1/tmp
    chmod 777 /volumes/xvd\$1/tmp
}
for disk in b c d; do
    format_and_mount \${disk} &
done
EOF

sed -i ’s/^PermitRootLogin .*/PermitRootLogin without-password/’ /etc/ssh/sshd_config
service sshd restart

ssh-keygen -N "" -t rsa -f ~/.ssh/id_rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

cat <<EOF > ~/.ssh/config
Host *
    LogLevel ERROR
    StrictHostKeyChecking no
    UserKnownHostsFile /dev/null
EOF
chmod 600 ~/.ssh/config

rm -rf /tmp/*
> ~/.bash_history && history -c && exit

ssh-keygen -N "" -t rsa -f ~/.ssh/id_rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

cat <<EOF > ~/.ssh/config
Host *
    LogLevel ERROR
    StrictHostKeyChecking no
    UserKnownHostsFile /dev/null
EOF
chmod 600 ~/.ssh/config

> ~/.bash_history && history -c && exit
```

5.2 Starting an Amazon EC2 cluster using Spot Instances

The following configuration and command starts a cluster in a placement group for low latency and high throughput networking. For larger clusters it is recommended to start an additional, on-demand instance to operate as the master node and monitor the cluster. This node can be created without the block devices used for spilled data. The placement group, subnet, AMI, EFS, key, and security group must be created and configured before launching the cluster.

The user-data initialization of cluster instances mounts a common Amazon EFS network filesystem from which the Flink software is run.

```
INSTANCE_TYPE=c4.4xlarge

AVAILABILITY_ZONE=us-east-1a
PLACEMENT_GROUP=my-pg-a
SUBNET_ID=subnet-d67eb769

AMI=ami-815f3b96
EFS_ID_AND_REGION=fs-3f744dd8.efs.us-east-1
KEY_NAME=MyKey
SECURITY_GROUP_ID=sg-c25e687f

USER_DATA=$(base64 --wrap=0 <<EOF
#!/bin/bash
mkdir /efs && mount -t nfs4 -o nfsvers=4.1 ${AVAILABILITY_ZONE}.${EFS_ID_AND_REGION}.amazonaws.com:/ /efs
EOF
)

LAUNCH_SPECIFICATION=$(cat <<EOF
{
  "ImageId": "${AMI}",
  "KeyName": "${KEY_NAME}",
  "UserData": "${USER_DATA}",
  "InstanceType": "${INSTANCE_TYPE}",
  "Placement": {
    "AvailabilityZone": "${AVAILABILITY_ZONE}",
    "GroupName": "${PLACEMENT_GROUP}"
  },
  "BlockDeviceMappings": [
    { "DeviceName": "/dev/sdb",
      "Ebs": { "VolumeSize": 255, "DeleteOnTermination": true, "VolumeType": "gp2", "Encrypted": true } },
    { "DeviceName": "/dev/sdc",
      "Ebs": { "VolumeSize": 255, "DeleteOnTermination": true, "VolumeType": "gp2", "Encrypted": true } },
    { "DeviceName": "/dev/sdd",
      "Ebs": { "VolumeSize": 255, "DeleteOnTermination": true, "VolumeType": "gp2", "Encrypted": true } }
  ],
  "SubnetId": "${SUBNET_ID}",
  "EbsOptimized": true,
  "SecurityGroupIds": [ "${SECURITY_GROUP_ID}" ]
}
EOF
)

SPOT_PRICE="0.50"
INSTANCE_COUNT=128
aws ec2 request-spot-instances --spot-price $SPOT_PRICE --instance-count $INSTANCE_COUNT \
--type "one-time" --launch-specification "${LAUNCH_SPECIFICATION}"
```

5.3 Generating and validating input data

Cluster-specific configuration must first be updated.

```
# AWS CLI configuration and credentials
$ cat ~/.aws/config
[default]
output = json
region = us-east-1
s3 =
  multipart_threshold = 1073741824
  multipart_chunksize = 1073741824
$ cat ~/.aws/credentials
[default]
aws_access_key_id = ...
aws_secret_access_key = ...

# piping data through the AWS CLI looks to ignore the configuration,
# so the multipart configuration should also be changed in
# /usr/local/lib/python2.7/site-packages/awscli/customizations/s3/transferconfig.py

DEFAULTS = {
    'multipart_threshold': 1024 * (1024 ** 2),
    'multipart_chunksize': 1024 * (1024 ** 2),

# save the list of workers - could also filter on instance type;
# remove master node from list of IPs if also captured by the filter
aws ec2 describe-instances --filter Name=placement-group-name,Values=my-pg | \
  python -c $'import json, sys; print "\\n".join(i["PrivateIpAddress"] for r in \
  json.load(sys.stdin)["Reservations"] for i in r["Instances"] \
  if i["State"]["Name"] == "running")' > ~/workers && wc ~/workers

# copy AWS CLI configuration and credentials to each worker
pdsh -w ^/home/ec2-user/workers mkdir -p /home/ec2-user/.aws
pdcp -r -w ^/home/ec2-user/workers /home/ec2-user/.aws/ /home/ec2-user/.aws
```

Ganglia is useful for monitoring aggregate network I/O. It could be useful for monitoring Flink, but because the Flink reporter broadcasts a unique ID for each TaskManager the Ganglia daemon is overwhelmed and crashes.

```
# edit /etc/ganglia/gmond.conf with master IP

cat <<EOF > /etc/httpd/conf.d/ganglia.conf
#
# Ganglia monitoring system php web frontend
#

Alias /ganglia /usr/share/ganglia

<Location /ganglia>
  Require all granted
</Location>
EOF

sudo pdcp -r -w ^/home/ec2-user/workers /etc/ganglia/gmond.conf /etc/ganglia/gmond.conf
sudo pdsh -w ^/home/ec2-user/workers service gmond start
sudo $GMOND_CONF /etc/ganglia/gmond.conf
sudo service gmond start
sudo service gmetad start
sudo service httpd start
```

The 100 TB of input data is constructed in blocks using gensort [12]. The following command uses GNU Parallel [13] to distribute work among nodes. It is preferable to write to shared memory both for performance and so that instances may be configured without storage.

When uploading 1 GB blocks, the md5 checksums can be validated against the list provided in the flink-cloudsort repository [11]. Use ”head -n” if validating less than a petabyte.

```
# create and upload files with proper MD5 etags
BLOCKS=100000
BUCKET=cloudsort
GENSORT=/efs/bin/gensort

GENDIR=/dev/shm
PARALLELISM=8
PREFIX=input
RECORDS=$((10*1000*1000)) # 1 GB
STORAGE_CLASS=REDUCED_REDUNDANCY
WORKER_IPS=~/workers

# first remove any files
pdsh -w ^/home/ec2-user/workers rm -f /dev/shm/block\*
```

```
# generate and upload input data
seq -f "%06g" 0 $(($BLOCKS - 1)) | \
  parallel -j ${PARALLELISM} --slf ${WORKER_IPS} --bar --timeout 120 --retries 490 "${GENSORT} \
  -b{}${RECORDS:1} ${RECORDS} ${GENDIR}/block{},buf && aws s3api put-object --storage-class ${STORAGE_CLASS} \
  --bucket ${BUCKET} --key ${PREFIX}/block{} --body ${GENDIR}/block{} > /dev/null && rm /dev/shm/block{}"

# fetch MD5 checksums for validation of uploaded files
aws s3api list-objects-v2 --bucket cloudsort --prefix input | python -c $'import json, sys; print \
  "\\n".join(file["ETag"].strip(\'"\') for file in json.load(sys.stdin)["Contents"])' > md5
```

5.4 Executing FlinkSort

The following script should be run with nohup in case the SSH session is terminated. It should be run from the flink-1.1.1 directory. Usage would be ”nohup ./run.sh 1 >run1.log &”. Multiple runs can be initiated by creating and running with nohup an outer script which calls run.sh more than once.

```
#!/usr/bin/env bash

# flush Linux memory caches
sudo sh -c "sync ; echo 3 > /proc/sys/vm/drop_caches"
sudo pdsh -w ^/home/ec2-user/workers sh -c "sync ; echo 3 > /proc/sys/vm/drop_caches"

CLOUDSORT_DIR=/efs/cloudsort

if [ "$#" -ne 1 ]; then
    echo "Usage: $0 <run id>"
fi

RUN_ID=$1
RUN_DIR=${CLOUDSORT_DIR}/run/${RUN_ID}

if [ -d "$RUN_DIR" ]; then
  echo "Run directory $RUN_DIR already exists!"
  exit -1
fi

# '-u' prevents python from buffering stdin and stderr
python -u ${CLOUDSORT_DIR}/statsd_server.py 9020 > statsd_server.log &
statsd_server_pid=$!

# record time before starting cluster
date +%s.%N
./bin/start-cluster.sh

# read JobManager configuration
CONF=conf/flink-conf.yaml
```

```
read HOST PORT SLOTS <<<$(python -c ’import yaml; conf=yaml.load(open("’${CONF}’")); \
  print conf["jobmanager.rpc.address"], conf["jobmanager.web.port"], conf["parallelism.default"]’)

# wait for all TaskManagers to start
while [ $SLOTS -ne ‘curl -s http://${HOST}:${PORT}/overview | python -c $’import sys, json; \
  data=sys.stdin.read(); print(json.loads(data)["slots-total"] if data else 0)’‘ ] ; do sleep 1 ; done

# execute FlinkSort
./bin/flink run -q -class org.apache.flink.cloudsort.indy.IndySort \
  ${CLOUDSORT_DIR}/flink-cloudsort-0.1-dev_shm_timeout.jar \
  --input awscli --input_bucket cloudsort --input_prefix input/ \
  --output awscli --output_bucket cloudsort --output_prefix output${RUN_ID}/ \
  --buffer_size 67108864 --chunk_size 250000000 --concurrent_files 16 --storage_class REDUCED_REDUNDANCY \
  --download_timeout 120 --upload_timeout 60

date +%s.%N
./bin/stop-cluster.sh

if kill -0 $statsd_server_pid 2>&1; then
  kill $statsd_server_pid
else
  echo "No statsd_server found with PID $statsd_server_pid"
fi

mkdir -p $RUN_DIR

mv statsd_server.log $RUN_DIR

mv log $RUN_DIR
mkdir log
```

5.5 Validating the output data

Generating input and validating output can be performed on instances without additional storage. Since spot pricing only applies to instances this can result in substantial savings.

```
# validate using valsort
PARALLELISM=8
WORKER_IPS=~/workers
VALSORT=/efs/bin/valsort
DATDIR=/efs/validate
BUCKET=cloudsort
PREFIX=output

# run valsort on each output file and save validation files
aws s3 ls s3://${BUCKET}/${PREFIX}/ --recursive | awk ’{print $4}’ | \
  parallel -j ${PARALLELISM} --slf ${WORKER_IPS} --bar --retries 490 "mkfifo /tmp/fifo{#} ; \
  mkdir -p ${DATDIR}/{//} ; aws s3 cp s3://${BUCKET}/{} - > /tmp/fifo{#} & ${VALSORT} -o ${DATDIR}/{}.dat \
  /tmp/fifo{#},buf ; rm /tmp/fifo{#}"

# concatenate validation files and run valsort on the full set
find ${DATDIR} -type f -name '*.dat' -printf '%P\n' | sort -V | xargs -i{} cat ${DATDIR}/{} > \
  ${DATDIR}/checksums
${VALSORT} -s ${DATDIR}/checksums

# delete the cloudsort output
aws s3 rm --recursive s3://cloudsort/${PREFIX}
```

6 Conclusion

Given a time machine I would not attempt this sort benchmark; however, I look forward to further optimizing Flink and FlinkSort for next year. Had this been a sponsored attempt it would have been much less stressful. This report will be included at a high-level in my presentation this month at Flink Forward 2016 [9].

The cloud is beautiful, and powerful resources can be obtained very cheaply (particularly on nights and weekends). It is also a black box and frustratingly difficult to apprehend.

```
sudo pdsh -w ^/home/ec2-user/workers sudo shutdown -h now
```

References

[1] *Amazon Web Services*. URL: https://aws.amazon.com.

[2] *Apache Flink*. URL: http://flink.apache.org.

[3] *Apache Hadoop*. URL: http://hadoop.apache.org/.

[4] *AWS Command Line Interface*. URL: https://aws.amazon.com/cli/.

[5] *AWS Reduced Redundancy Storage*. URL: https://aws.amazon.com/s3/reduced-redundancy/.

[6] *CloudSort*. URL: http://sortbenchmark.org/2014_06_CloudSort_v_0_4.pdf.

[7] Michael Conley, Amin Vahdat, and George Porter. *Sorting 100 TB on Google Compute Engine*. 2015. URL: http://cseweb.ucsd.edu/~gmporter/papers/themis-gce15-tr.pdf.

[8] Michael Conley, Amin Vahdat, and George Porter. *TritonSort 2014*. 2014. URL: http://sortbenchmark.org/TritonSort2014.pdf.

[9] *Flink Forward*. URL: http://flink-forward.org/.

[10] *flink-1.1.1*. URL: http://www.apache.org/dyn/closer.lua/flink/flink-1.1.1/flink-1.1.1-bin-hadoop1-scala_2.10.tgz.

[11] *flink-cloudsort*. URL: https://github.org/greghogan/flink-cloudsort.

[12] *gensort*. URL: http://www.ordinal.com/gensort.html.

[13] *GNU Parallel*. URL: https://www.gnu.org/software/parallel.

[14] *Google Cloud Platform*. URL: https://cloud.google.com.

[15] *GraySort on Apache Spark by Databricks*. URL: http://sortbenchmark.org/ApacheSpark2014.pdf.

[16] *Intel SHA Extensions*. URL: https://software.intel.com/en-us/articles/intel-sha-extensions.