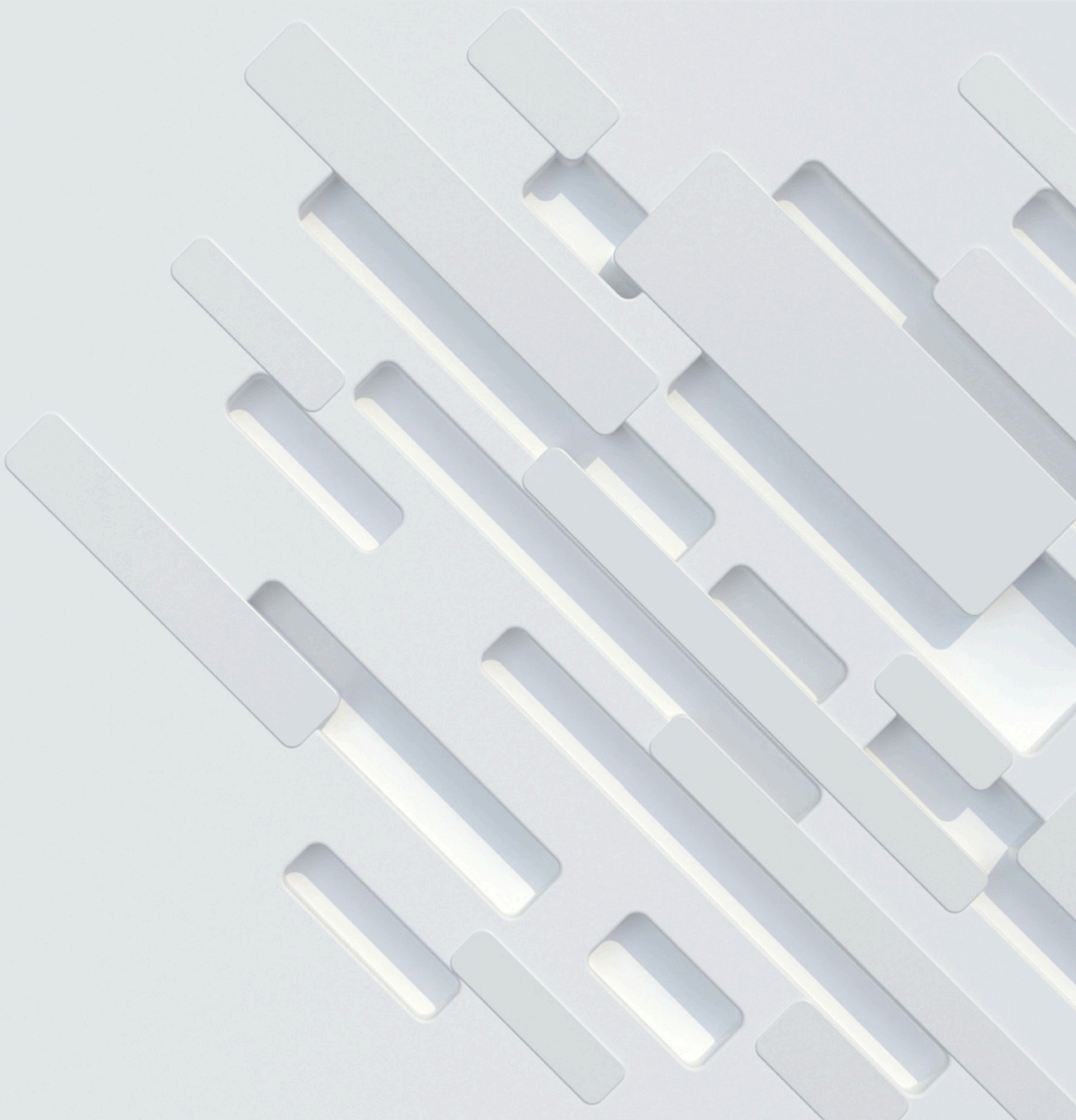
Hadoop and SPARK with WekaIO Matrix



SUMMARY

Hadoop and Spark are frameworks that are widely used for processing Big Data in a distributed environment. The Hadoop framework was built as a shared-nothing standalone architecture that relies heavily on data locality to improve performance. Data locality is the notion that data is processed on the local HDD/SSD inside the server to avoid sending it through the network. While this may work well for small environments, it is cumbersome to manage at scale because complete compute nodes have to be added in order to increase available capacity. In addition the data cannot be shared with other applications and analytics tools; so Hadoop becomes an island of storage that requires back and forth copying of data to and from other application data repositories.

WekaIO can remove this constraint by allowing all of the user applications to access a single namespace. This is done by using WekaIO POSIX client to access the data store instead of relying on dedicated Hadoop nodes utilizing HDFS. The result is a single data set that improves performance, eliminates copying of data ,and optimizes cost, as customers no longer have to manage multiple storage clusters.

WHAT IS HDFS?

HDFS (Hadoop Distributed File System) is a storage system used by several applications, including Hadoop, to implement a distributed file system (FS) for parallel data access across scalable clusters. HDFS was originally architected to leverage low cost hardware that frequently crashed, and to minimize network traffic by performing all tasks locally on the node that possesses the data. HDFS is not a Portable Operating System Interface (POSIX) compliant file system, which means that data stored in the HDFS format cannot be shared with other applications that expect POSIX file semantics. With HDFS, data cannot be accessed by simply mounting the file system as a POSIX file system; instead data is accessed via an API or via CLI tool.

HADOOP SUPPORT FOR POSIX FILE SYSTEMS

Hadoop has the ability to mount directly to a POSIX file system “out-of-the-box”. To access data in this manner, use Hadoop’s built-in connector to POSIX file systems called LocalFileSystem.

Any Hadoop application can use LocalFileSystem to access WekaIO’s global namespace, removing the need for a dedicated Hadoop/HDFS cluster. LocalFileSystem can be mounted as a POSIX file system (FS) that will be recognized by the applications.

Enabling LocalFileSystem support requires a code change to use a local path:
Replace hdfs path (starting with hdfs://) with local path (starting with file://)

BENEFITS OF WekaIO POSIX FS

NO TRIPLE REPLICATION REQUIRED

HDFS utilizes triple replication for data protection, which becomes prohibitively expensive when high performance storage media such as NVME or Flash is required to drive performance and reduce latency. WekaIO eliminates the need for three data copies as it employs a form of erasure coding that delivers significantly better capacity utilization. The WekaIO mount not only replaces HDFS as the shared FS, but also provides efficient distributed data coding. Capacity utilization jumps from 33.3% to over 70%.

DATA SHARING ACROSS APPLICATIONS

Hadoop clusters typically operate as stand-alone application clusters that perform business critical tasks required for further workflow needs. Other applications may need this data, requiring a full data copy and doubling storage costs. WekaIO POSIX FS allows multiple applications to access a single copy of data reducing storage resource needs. This reduces cost and saves time as data does not need to be replicated across islands of storage.

WekaIO SPARK OPERATIONS (OPS) TESTING

To validate that this solution works, WekaIO conducted a series of benchmarks utilizing Spark; the following section outlines tests that can easily be replicated in a test cluster.

THE ENVIRONMENT

WekaIO Cluster

For this demo we created an AWS WekaIO cluster as well as an AWS EMR (Elastic MapReduce) cluster that was connected to the WekaIO clustered filesystem.

The Data

Following the cluster creation, we generated data by running the following Spark code in a spark-shell on the EMR master:

```
import org.apache.spark.sql.{functions => F}
val baseDs = spark.createDataset(0 until 100000)
val numDuplications = 9
val duplicatedDs = (0 until numDuplications).foldLeft(baseDs)((x, _) => x unionAll x)
duplicatedDs.select(F.rand() as "v").as[Double].write.parquet("hdfs:///user/zeppelin/data")
```

This code generates data of roughly 6 GB with ~150 parquet files of even size. The value of numDuplications can be changed to create a bigger or smaller dataset. The amount in the dataset will be created is:

```
10^5 * 2^numDuplications
```

To copy the data to the WekaIO filesystem we ran the following command:

```
hadoop fs -copyToLocal /user/zeppelin/data /mnt/weka/
```

THE BENCHMARK

To run the benchmark using the WekaIO POSIX file mount, we used Spark as it utilizes Hadoop's FileSystem class under the hood.

The two main benchmarks were as follows: Read-Sort-Count and and Read-Sort-Write.

```
1. spark.read.parquet("/mnt/weka/data").orderBy($"v").count
2. spark.read.parquet("/mnt/weka/data").orderBy($"v").write.parquet("/mnt/weka/dataSorted")
```

The tests were repeated for HDFS by simply replacing the paths as described in the previous section.

Results

WekaIO completed the Read-Sort-Count faster than HDFS, recording a wall clock time that was almost 50% faster.
WekaIO completed the Read-Sort-Write faster than HDFS, recording a wall clock time that was 10% faster.
Both tests demonstrate that the WekaIO POSIX mount is as fast or faster than HDFS on local disk.

WekaIO SPARK BANDWIDTH TESTING

The Data

To perform this test, data was generated by running the following Bash code on the WekaIO client:

```
for i in {1..16}; do for j in {1..13}; do dd if=/dev/zero of=file$i-$j.bin bs=16777216 count=100 & done; wait;
done
```

This code generates approximately 325GB of data in 200 binary-zeroed files.

THE BENCHMARK

To run the benchmark utilizing the WekaIO POSIX file mount, cd into Spark's directory, and start the shell using:

```
./bin/spark-shell --conf spark.task.cpus=5
```

We then ran the following in the shell:

```
import java.time.Instant
val rdd = spark.sparkContext.binaryFiles("/wekafs/driver/default/data")
val rdd2 = rdd.map {
 case (_, s) =>
  val arr = Array.fill[Byte](1024*1024*10)(0)
  val stream = s.open()
  Stream.continually(()).takeWhile(_ => stream.available() > 0).map(_ => {
   stream.readFully(arr)
   0
  }).reduce((_: Int, _: Int) => 0)
}
def a(): Unit = {
 println(Instant.now)
 println(rdd2.reduce(_ + _))
 println(Instant.now)
}
a()
```

Results

When utilizing **fio**,Spark got the same performance on a local file system and WekaIO POSIX file system, reaching 2.8GB/s.

Note: The performance was limited to the network bandwidth provided on the AWS instances (performance was limited to 2.8GB/s by the AWS 25Gbs network bandwidth limit). If the benchmark is run on a 100GbE network the performance will increase to approximately 11GB/s, saturating the network link.

910 E Hamilton Avenue, Suite 430, Campbell, CA 95008 T: 408.335.0085 E: info@weka.io www.weka.io

©2019-2020 All rights reserved. WekaIO, WekaFS, WIN, Weka Innovation Network, the Weka brand mark, the Weka logo, and Radically Simple Storage are trademarks of WekaIO, Inc. and its affiliates in the United States and/or other countries. Other trademarks are the property of their respective companies. References in this publication to WekaIO's products, programs, or services do not imply that WekaIO intends to make these available in all countries in which it operates. Product specifications provided are sample specifications and do not constitute a warranty. Information is true as of the date of publication and is subject to change. Actual specifications for unique part numbers may vary.