Analysis of Access and Transfer data for the dCache mass storage systems at DESY using Apache Spark

Gunay Abdullayeva, University of Tartu, Estonia

Supervisors: Thomas Hartmann and Christian Voss

September 5, 2018

Abstract

`dCache` is a system which is used to store and host large amounts of data. However, transfers between `dCache` and individual clients may fail due to various reasons, e.g. network disconnection. In this project, we have worked with `Apache Spark` to analyze all process in `dCache`. `Apache Spark` provides an interface to process data in clusters implicitly parallel. Our aim is to study all kinds of transfers on data in `dCache` at DESY. DESY is one of the world's leading accelerator centres where researchers use the large-scale facilities to explore the microcosm in all its variety - from the interactions of tiny elementary particles and the behaviour of new types of nanomaterials to biomolecular processes that are essential to life.

Contents

1 Introduction

dCache is a system which manages storage and access to scientific data, transparently distributed among a different number of disk storage nodes and tape backends. One of the key design features of the `dCache` is that although the location and multiplicity of the data are autonomously determined by the system, based on configuration, CPU load, and disk space, the namespace is represented within a single file system tree. The system has shown to be capable of improving the efficiency of connected tape storage systems, by caching, and scheduled staging techniques. Furthermore, it optimizes the throughput to and from data clients as well as smoothing the load of the connected disk storage nodes by dynamically replicating datasets on detection of high demand. The system is tolerant against failures of its data servers enabling administrators to go for commodity disk storage components. Supports a large set of standard access protocols to the data repository and its namespace are provided by `dCache`. DESY is one of the contributors. `dCache` stores *billing* files which contain information about transfers, accesses, and removals of files. As the total size of the files is large enough, which a local machine does not have the capacity to implement all operation, we use `Apache Spark` for all analyses.

Apache Spark is a fast and general-purpose cluster computing system and it has the architectural foundation which is called the resilient distributed dataset (`RDD`). Spark and its RDDs were developed in 2012 in response to limitations in the `MapReduce` cluster computing paradigm, which forces a particular linear dataflow structure on distributed programs. `MapReduce` algorithm reads input data from disk, maps a function across the data, reduces the results of the map, and stores reduction results on disk. Spark's RDDs function is a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory. `Apache Spark` allows the distribution of processes on data items over a cluster of machines. It supports high-level APIs in `Java`, `Scala`, `Python` and `R` and provides a rich set of higher-level tools including `Spark SQL` for `SQL` and structured data processing, `MLlib` for machine learning, `GraphX` for graph processing, and `Spark Kafka Streaming`.

Apache Kafka is a distributed streaming platform which has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant way
- Process streams of records as they occur.

Kafka is generally used for building real-time streaming applications that transform or react to the streams of data.

2 Motivation

This section discusses two factors that motivate this analysis on the DESY `dCache`. First, there is a need to use a fast and memory efficient tool to analyze billing files which cover the time span 2016-2018. In this case, `Apache Spark` provides distributed processes on these billing files. Second, as the billing file format has not a consistent structure, there is a need for a generalised structure. That is why JSON file format is used by Kafka. These are discussed in the following two subsections, where Subsection 3.1 and 3.2 discuss how `Apache Spark` and `Apache Kafka` are applied for this analysis.

3 Analysis Methods

3.1 Apache Spark

Apache Spark is a fast and general-purpose cluster computing system and it has the architectural foundation which is called the resilient distributed dataset (RDD). We created our parser method to get RDD structure and used Apache Spark to do analysis on it. The billing files contain information about time , operation type(transfer, request and, remove), file size, unique file id called pnfs id, pool to store files, pool group to store pools, appropriate protocol, the result transaction and some additional information. There is three type of operations: transfer, request and remove. It is given one example for each type as follows:

Transfer:

```
01.01 04:34:10 [pool:dcache-atlas89-02:transfer] [0000A7505A6E2B
394FCCB314439421952EFC,2973444] [/pnfs/desy.de/atlas/dq2/
atlasdatadisk/rucio/mc15_13TeV/5a/ef/EVNT.10366673._006883.pool.
root.1] atlas:atlasdatadisk@osm 2973444 375 false \{GFtp-2.0
130.246.176.69 51307} [door:GFTP-dcache-door-atlas13-AAVFABud_qg:
1483241649957000] {0:""}
```

Request:

```
01.01 01:50:07 [door:GFTP-dcache-door-atlas15-AAVE_c4oz4g@gridftp-
dcache-door-atlas15Domain:request] ["/DC=ch/DC=cern/OU=Organic Units/
OU=Users/CN=atlact1/CN=555105/CN=Robot: ATLAS aCT 1":40001:4000:
131.169.161.226] [00002C8EE4F7FE49468D96DC9AD0566BE0C9,4144006055]
[/pnfs/desy.de/atlas/dq2/atlasdatadisk/rucio/data16_13TeV/4f/2d/
data16_13TeV.00309759.physics\_Main.merge.AOD.f750_m1689._lb0582.
_0001.1] atlas:atlasdatadisk@osm 47206 0 {0:""}
```

Remove:

```
01.01 08:40:43 [pool:dcache-atlas129-05@dcache-atlas129-05Domain:remove]
[00002EDB507A343C40709A21C6628548E6D5,183923694] [Unknown] atlas:
atlasdatadisk@osm {0:""}
```

We created the database having taken important information from the billing file and considered all different cases depending on the file type. The database is described in Table 1 just for one instance.

Table 1: Data format with columns and values for one event.

Columns	**Values**
Time	11.01.2017 00:00:00
MessageType	transfer
Pool	dcache-atlas32-03
PoolGroup	atlasscratchdisk
PnfsId	0000B462D717D829475CB0E22DD9CE4BC136
FileSize	2071418723
Door	Xrootd-dcache-door-atlas17
Protocol	Xrootd
Domain	xrootd-dcache-door-atlas17
ClientIp	131.169.163.184
ReturnCode	451
ReturnMessage	Aborting transfer due to session termination

The analysis is split and distributed throughout the clusters. The following code snippet shows how to connect to the cluster, to work with the executor nodes and create a `Spark dataframe` object.

```
cluster = 'spark://os-spark-dev01.desy.de:9618'
spark_conf = SparkConf().setMaster(cluster)
.set('spark.executor.memory','14G')
.set('spark.driver.memory','5G')
sc = SparkContext(conf = spark_conf)
file = '/Billing-Data/ATLAS/2018/*/billing -2018.*.*'
billing_data = convert_data(file)
billing_data.createOrReplaceTempView("billing")
billing_data.cache()
```

First, we connect to the cluster via *spark://os-spark-dev01.desy.de:9618* using the `setMaster` function. The other parameters such as *spark.executor.memory* and *spark.driver.memory* define the amount of memory to use per executor memory and for the driver process respectively. Then, we initialize the `SparkContext` object to be used in the *convert_data()* function, where we used regular expression to parse the file to the new data format. Finally, the temporary view is formed and the dataframe is cached to accelerate subsequent queries.
Although we created the new data format, there were still some challenges in the conversion of *ClientIp*. Sometimes, *ClientIp* is given like ipv4 or ipv6 and ipv6 is also of different length. As ClientIp is formatted with different parameters and various length, we should consider all different cases in our parser. Since we did not want to lose much more time for parsing, we used *JSON* stream published by `Apache Kafka` producer.

3.2 Apache Kafka

`Apache Kafka` is a distributed streaming platform which provides an event stream between producer and consumer. In our case, the consumer takes the *JSON* stream provided by the producer. JSON stream is described for one instance as follows. **JSON stream**:

```
{'cellDomain': 'dcache-dot7-08Domain',
'msgType': 'transfer',
'fileSize': 942200,
'transferSize': 942200,
'status':
    {'msg': '',
    'code': 0},
'cellType': 'pool',
'transferPath': 'Unknown',
'session': 'pool:dcache-dot7-08@dcache-dot7-08Domain:15339-1390',
'readActive': 'PT0.001774434S',
'protocolInfo':
    {'host': '131.169.191.240',
    'protocol': 'Http',
    'versionMajor': 1,
    'port': 0,
    versionMinor': 1},
'isP2p': True,
'initiator': 'pool:dcache-dot8-05@dcache-dot8-05Domain',
'subject': ['UidPrincipal[0]', 'GidPrincipal[0,primary]'],
'readIdle': 'PT0.02863589S',
'meanReadBandwidth': 801808915.4950517,
'pnfsid': '0000DFE174F3A8594C5FB8429C7CC1D0AE67',
'storageInfo': 'dot:store@osm',
'date': 'Tue Aug 21 16:34:47 CEST 2018',
'version': '1.0',
'isWrite': 'read',
'queuingTime': 0,
'billingPath': 'Unknown',
'cellName': 'dcache-dot7-08',
'transferTime': 30}"
```

As we see, there is a key and value structure and it makes comfortable to get important information from the stream. We created the new data format to analyze just transfers. The structure of the data is shown in Table 2.
The following code snippet shows how to connect to the producer, get the stream data and process it.

```
sc = SparkContext(appName="Streaming")
sc.setLogLevel("WARN")
ssc = StreamingContext(sc, 1)
topic = "billing"
broker = "dcache-billing-cloud:9092"
offset = "smallest"
kvs = KafkaUtils.createDirectStream(ssc, [topic],
{"metadata.broker.list": broker, 'auto.offset.reset': 'smallest'})
lines = kvs.map(lambda row: JSON.loads(row[1]))
.map(lambda row: billing(row, 'transfer.csv'))
ssc.start()
ssc.awaitTermination()
```

Since the Spark cluster works similar to a batch system, it is necessary to prepare the job before submission. This is the role of the `sparkContext` object. The job name is

Table 2: Data format with columns and values for one event.

Columns	**Values**
Time	1535632685
Pool	dcache-cloud07-08
PoolGroup	volatile
PnfsId	0000ED2BD4D300DC4E6C8386E7C8AD306EEC
MeanWriteBandwidth	1324418597.1540759
MeanReadBandwidth	2065596142.8823228
BillingPath	/pnfs/desy.de/dot/store/SET_testfile
FileSize	524288000
Door	DCap-gsi-dcache-dot3-AAVz37Hv6ug
Protocol	DCap
Domain	dcache-dot3_gsidcap
ClientIp	131.169.223.91
ReturnCode	0
ReturnMessage	

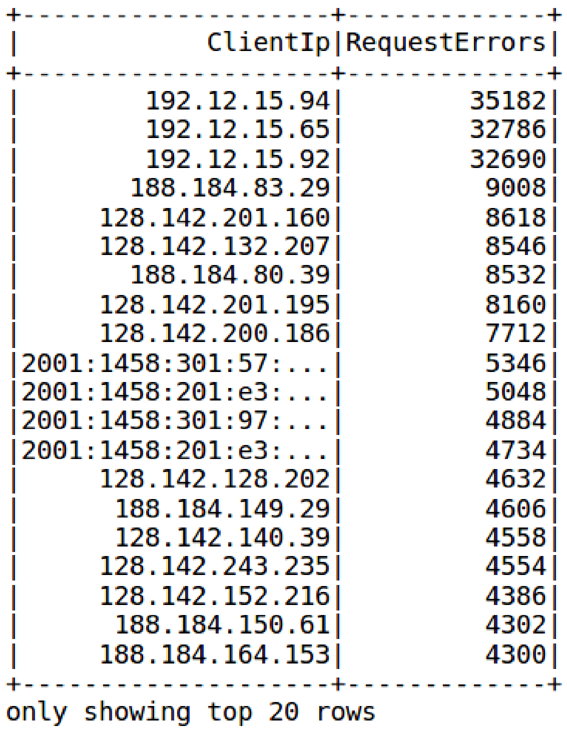
given by the *appName* parameter. The verbosity of the output can be configured with the `setLogLevel` function. We pass the Spark context object along with the batch duration, which here is set to 1 second to form a local `Streaming Context` object. Topic and broker should be specified to create a direct stream. The offset parameter is optional, but since we want to get all data in the stream, we should specify it. Using the native `Spark Streaming Kafka` capabilities, we use the streaming context, topic, broker and offset parameters to connect to the Kafka cluster. The inbound stream is a DStream object, which supports various built-in transformations such as `map` which is used here to parse the inbound messages from their native *JSON* format. In our case, we transform the message from *JSON* to the dataframe format. Having defined the streaming context, we can start the process and see the result of the transformation.

4 Analysis

In this section, we look at the result of all analysis. The sections 4.1 and 4.2 contain the analysis of the billing files.

4.1 Analysis of Errors

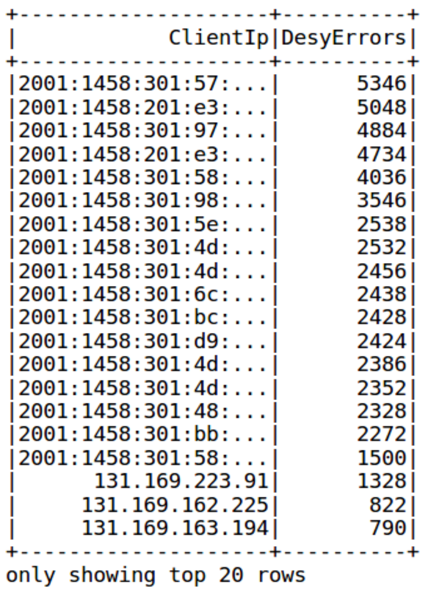
The following error analysis is performed on the billing data 2018. We counted errors in which the file transfer is successful, however, the file access fails with errors such as 451 which means the session is terminated. Totally, the number of the errors is 24384634 and the number of 451 error is 424082. The errors are ordered and associated by the number of occurrence per IP address in the Figure 1.



```
+--------------------+-------------+
|            ClientIp|RequestErrors|
+--------------------+-------------+
|        192.12.15.94|        35182|
|        192.12.15.65|        32786|
|        192.12.15.92|        32690|
|       188.184.83.29|         9008|
|     128.142.201.160|         8618|
|     128.142.132.207|         8546|
|       188.184.80.39|         8532|
|     128.142.201.195|         8160|
|     128.142.200.186|         7712|
|2001:1458:301:57:...|         5346|
|2001:1458:201:e3:...|         5048|
|2001:1458:301:97:...|         4884|
|2001:1458:201:e3:...|         4734|
|     128.142.128.202|         4632|
|      188.184.149.29|         4606|
|      128.142.140.39|         4558|
|     128.142.243.235|         4554|
|     128.142.152.216|         4386|
|      188.184.150.61|         4302|
|     188.184.164.153|         4300|
+--------------------+-------------+
only showing top 20 rows
```

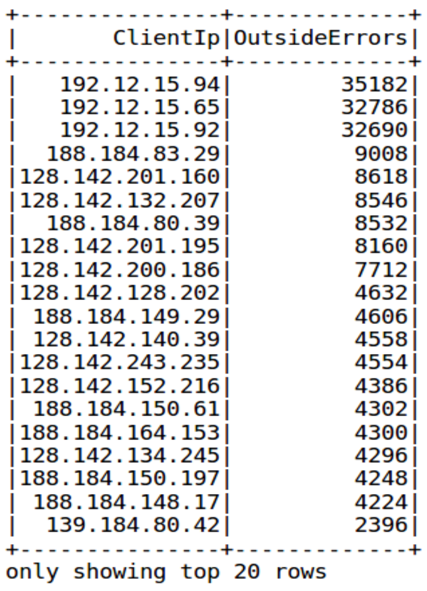
Figure 1: The number of the request errors by clients

We specified the errors which happen inside and outside DESY network. The Figure 2 and Figure 3 show the most 20 errors by clients inside and outside DESY, respectively.



```
+--------------------+----------+
|            ClientIp|DesyErrors|
+--------------------+----------+
|2001:1458:301:57:...|      5346|
|2001:1458:201:e3:...|      5048|
|2001:1458:301:97:...|      4884|
|2001:1458:201:e3:...|      4734|
|2001:1458:301:58:...|      4036|
|2001:1458:301:98:...|      3546|
|2001:1458:301:5e:...|      2538|
|2001:1458:301:4d:...|      2532|
|2001:1458:301:4d:...|      2456|
|2001:1458:301:6c:...|      2438|
|2001:1458:301:bc:...|      2428|
|2001:1458:301:d9:...|      2424|
|2001:1458:301:4d:...|      2386|
|2001:1458:301:4d:...|      2352|
|2001:1458:301:48:...|      2328|
|2001:1458:301:bb:...|      2272|
|2001:1458:301:58:...|      1500|
|      131.169.223.91|      1328|
|     131.169.162.225|       822|
|     131.169.163.194|       790|
+--------------------+----------+
only showing top 20 rows
```

Figure 2: The number of the request errors by clients inside Desy



```
+---------------+-------------+
|       ClientIp|OutsideErrors|
+---------------+-------------+
|   192.12.15.94|        35182|
|   192.12.15.65|        32786|
|   192.12.15.92|        32690|
|  188.184.83.29|         9008|
|128.142.201.160|         8618|
|128.142.132.207|         8546|
|  188.184.80.39|         8532|
|128.142.201.195|         8160|
|128.142.200.186|         7712|
|128.142.128.202|         4632|
| 188.184.149.29|         4606|
| 128.142.140.39|         4558|
|128.142.243.235|         4554|
|128.142.152.216|         4386|
| 188.184.150.61|         4302|
|188.184.164.153|         4300|
|128.142.134.245|         4296|
|188.184.150.197|         4248|
| 188.184.148.17|         4224|
|  139.184.80.42|         2396|
+---------------+-------------+
only showing top 20 rows
```

Figure 3: The number of the request errors by clients outside Desy

As we see from figures, the most of request errors come from outside of the DESY network.

4.2 Analysis of Transfers

All analyses are performed on the billing-2017.11.01 file for the first of November 2017. Figure 4 shows how many transfers happened on the first day of November. The x and y axes represent the number of unique pnfs ids and transfers, respectively. `Pnfs id` is unique for each file and we have 122844 unique `Pnfs ids` in this case. As we see, the number of transfers is high in some cases and the most transferred file is `00002D881EDA74CA46E9A9C36E690ED65342` which was transferred about 1600.

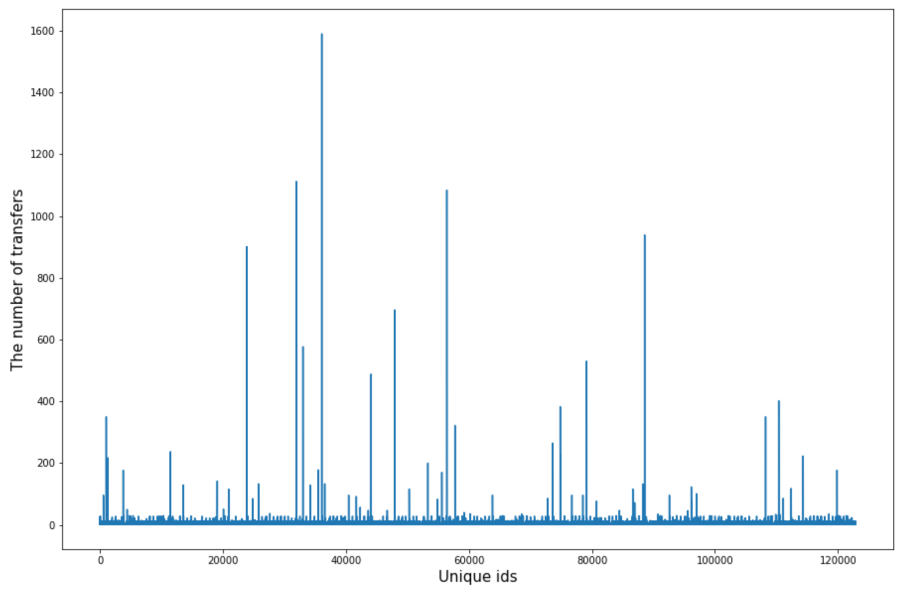


Figure 4: The number of transfers for each file in the first day of November.

Figure 5 is a different representation of the same data shown in Figure 4. This figure depicts how many files have been transferred a certain number of times. It is visible that the transfers are executed mostly once or twice time in more than 50000 files.

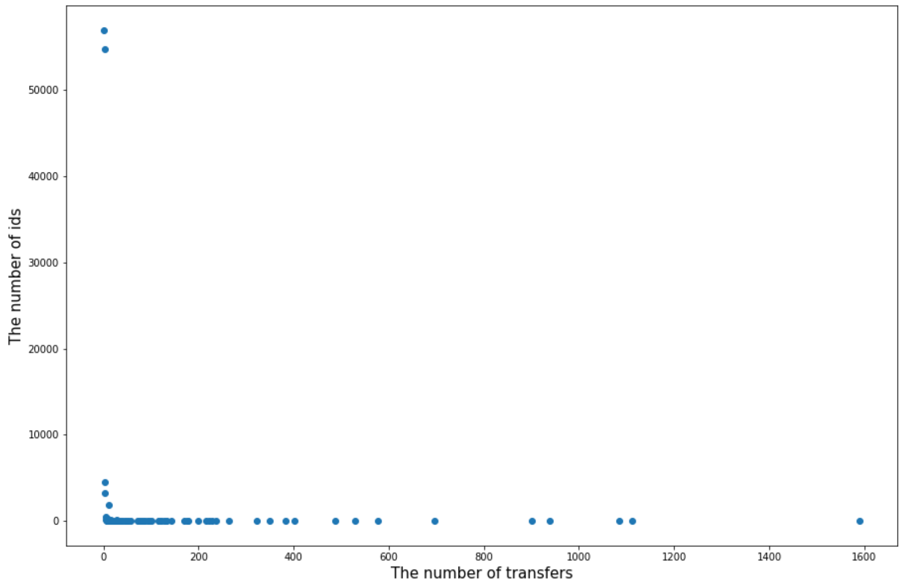


Figure 5: The same number of transfers for different files.

Figure 6 is the log scale description of the Figure 5. The number of the files for the same number of transfers is not readable in most cases. So this figure helps to see how the amount of files differs in the low rate of the same number of transfers.

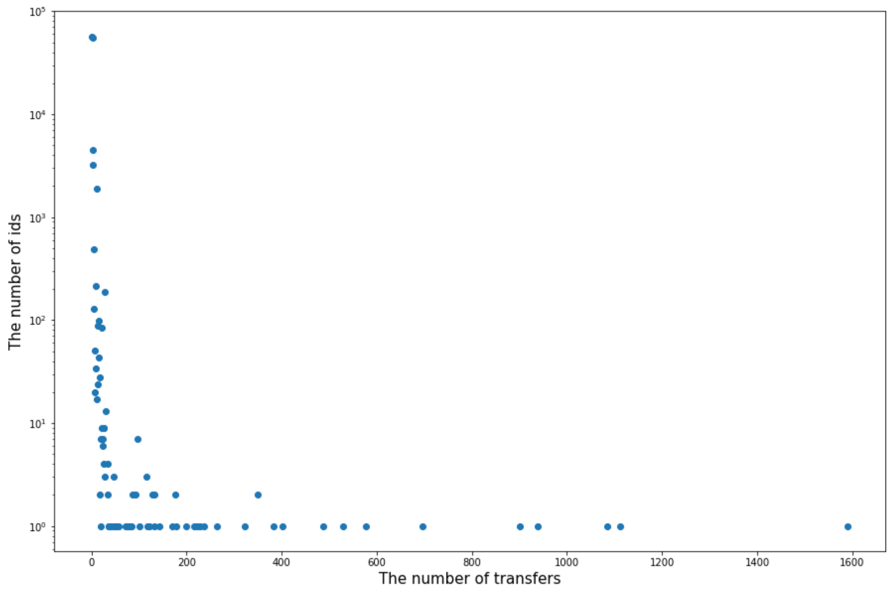


Figure 6: The log scale description of the Figure 5.

Figure 7 is analog to Figure 6, however, the data is split by pool groups. We have 3 types of pool group: scratch, data, and local disk. The scratch disk stores temporary files and remove arbitrary data depending on the free space. That is why, the transfers are implemented on the scratch disk mostly. The most transferred file is also in the scratch disk.

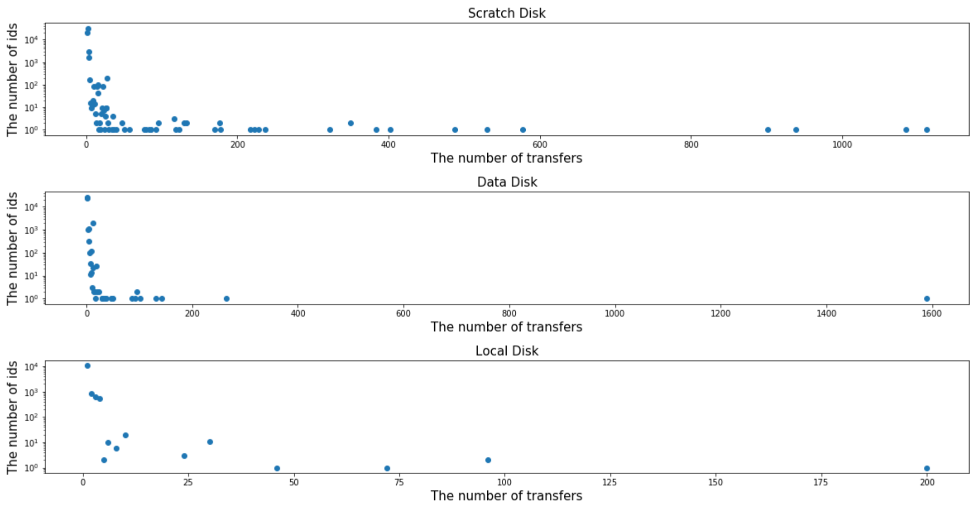


Figure 7: The log scale description of the Figure 5 for the different pool groups.

4.3 Analysis of Transfers using Apache Kafka

In contrast to the previous sections, we have performed the analysis on the `Kafka` stream data in this section. The stream data covers the information from 20.08.2018 to 28.08.2018.
Figure 8 shows the transfer rate over time. The x axis represents the unix time. The y axis shows the mean of instantaneous IO bandwidth while reading the file. The red and yellow lines indicates the mean and the standard deviation of the data points.

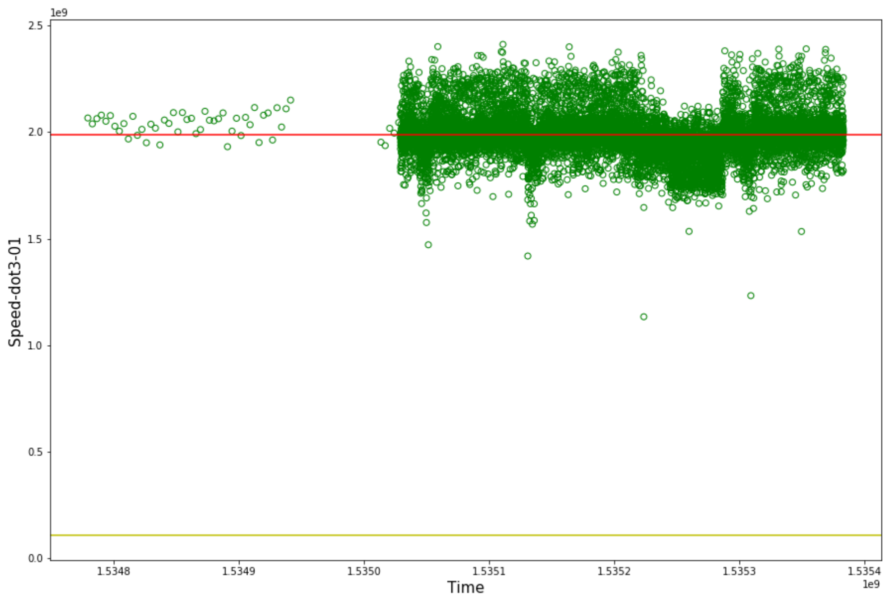


Figure 8: The speed distribution of transferred files in different time.

The streaming analysis is run agaist a different `dCache` instance. This contains the poolgroup *store* among others. The pools *dcache-dot7-01*, *dcache-dot7-05*, *dcache-dot7-08*,*dache-dot8-04*, *dcache-dot7-04* are located in the store pool group. We selected the transfers in *dcache-dot7* and *dcache-dot8* in the Figure 9 separately. The total number of transfers is about identical for both hosts during the stream period of eight days.

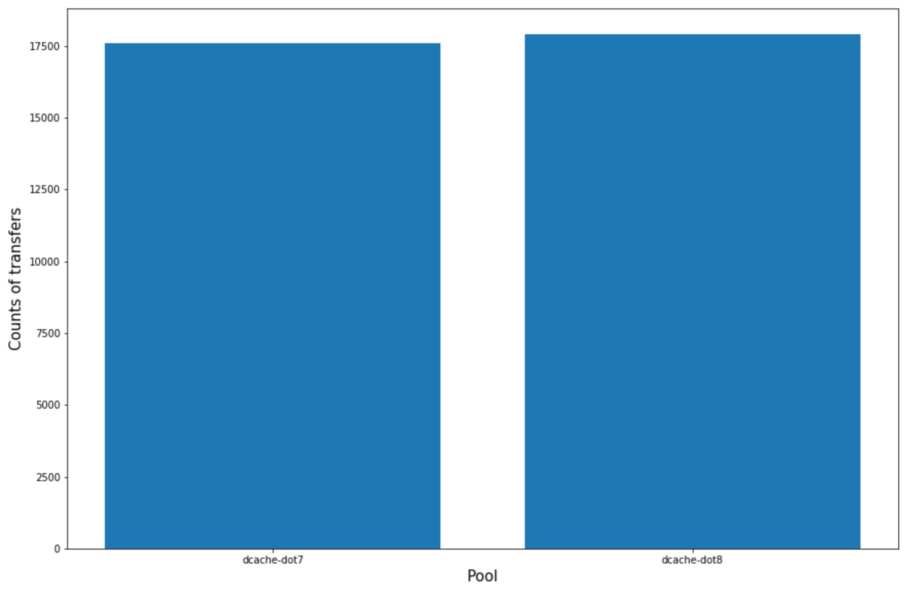


Figure 9: The number of transfers for *dcache-dot7* and *dcache-dot8* in the store pool group.

Figure 10 shows the dependence of the transfer rate on the file size. The x axis represent the file size with a unit byte and the y shows the transfer rate of the file with a unit bit per second. The interesting thing is that, although the files size is small, the transfer rate of the same sized files can be slow or fast.

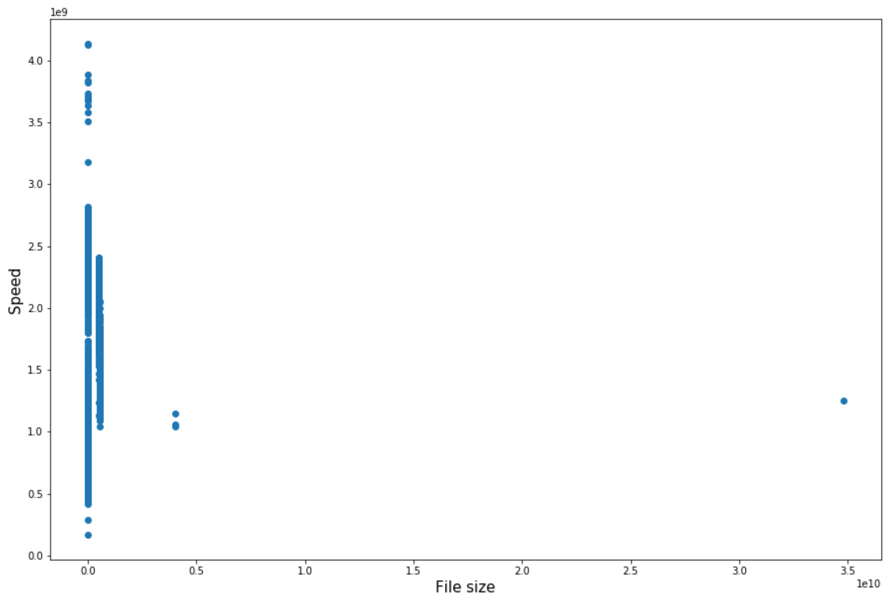


Figure 10: The transfer rate of files depending on the files size.

Figure 11 describes the different transfer rates of the files which are transferred at the same time. The color represents the various transfer rate in this *heatmap*. The dark and light colors show the high and low speed respectively. The purpose of this analysis is to see whether the transfer rate of the files differ for the same time.

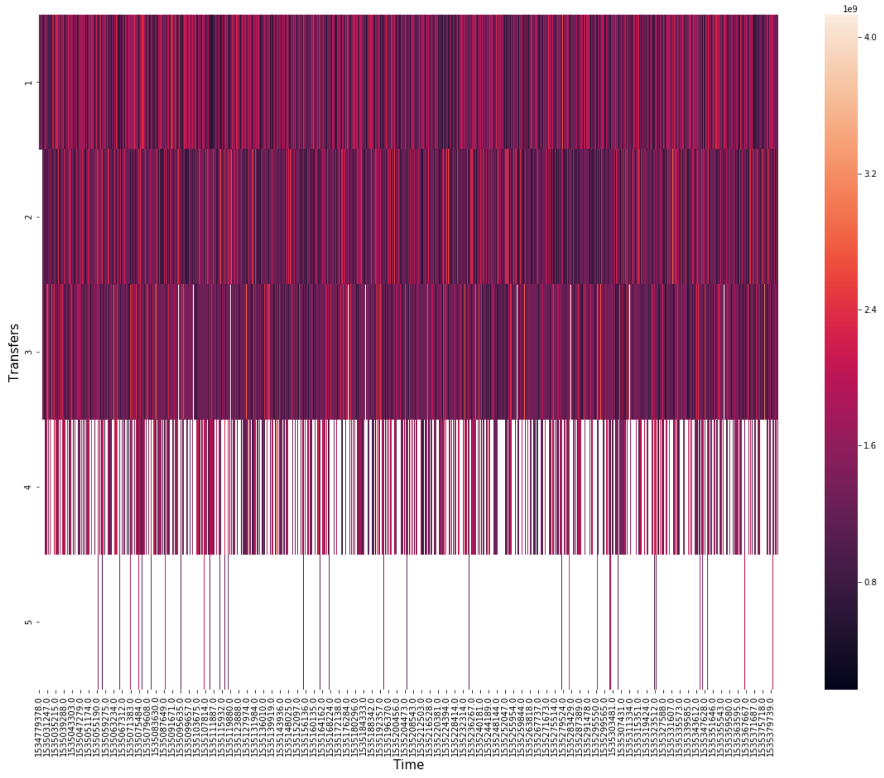


Figure 11: The transfer rates of the files which are transferred at the same time.

5 Conclusion

We used Apache Spark to analyze on the files in clusters. As we see from the results, `Apache Spark` is able to parallelize all processes on the executor nodes equally. As the billing files contain different variations, parsing of the billing files took several weeks to consider all various cases. That is why we moved to `Apache Kafka` to work with *JSON* formatted stream data and did all analysis later on it.

References

[1] https://www.dcache.org/. *dCache*

[2] https://spark.apache.org/. *Apache Spark*

[3] https://en.wikipedia.org/wiki/Apache_Spark. *Wikipedia*

[4] https://kafka.apache.org/. *Apache Kafka*

[5] https://spark.apache.org/docs/2.3.0/streaming-kafka-0-8-integration.html. *Apache Spark*

[6] hhttps://medium.com/@kass09/spark-streaming-kafka-in-python-a-test-on-local-machine-edd47814746. *Medium*