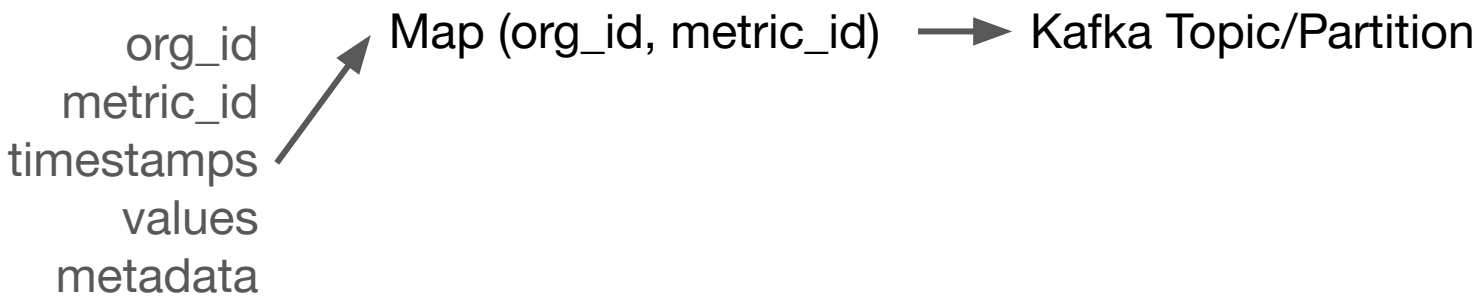
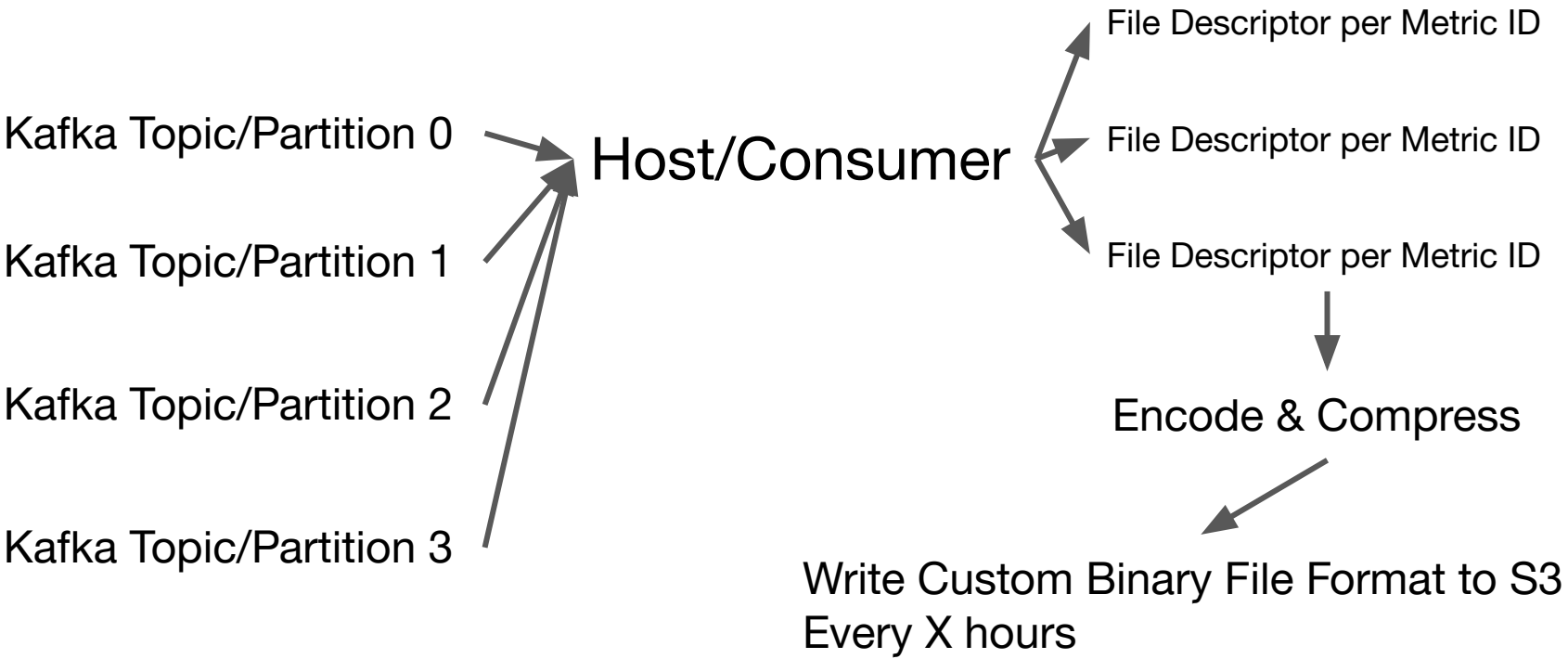
1. The original system

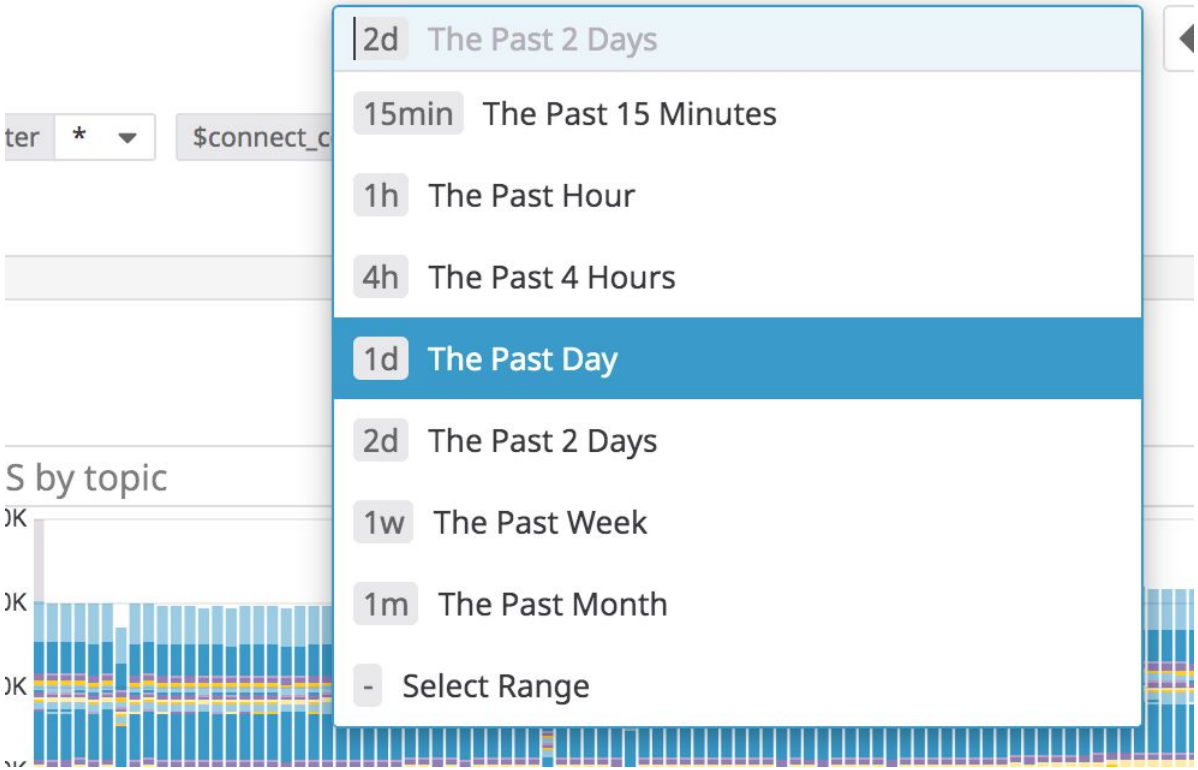
Payloads



1. The original system

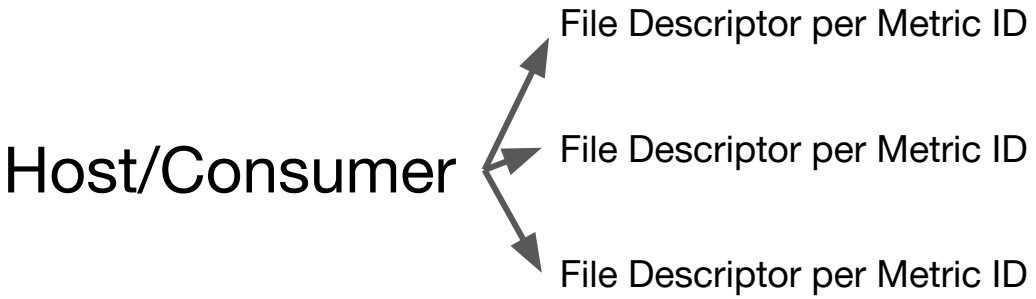


1. The original system

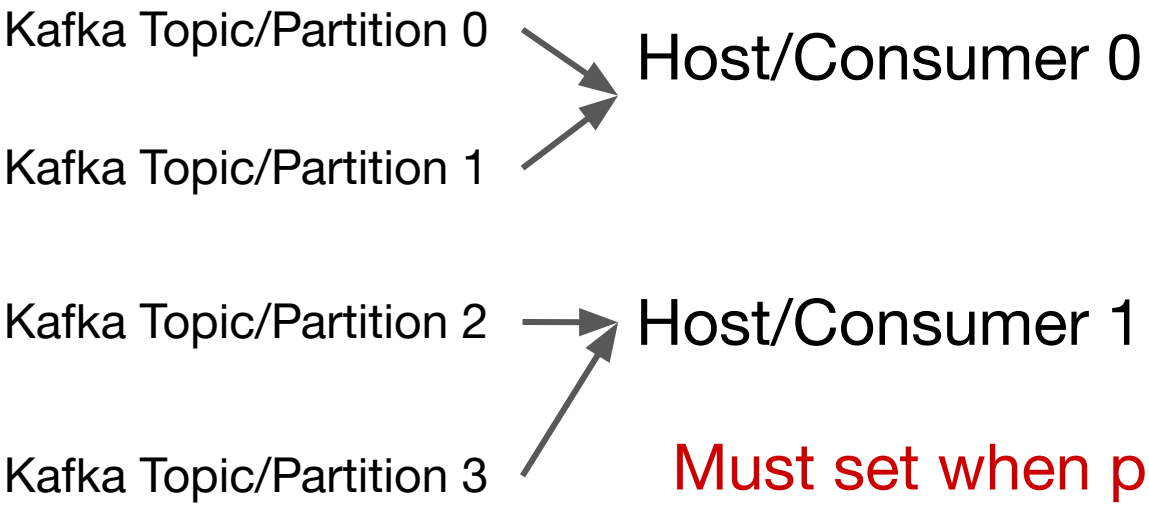


1. The original system

Max 1M file descriptors per host



1. The original system



Must set when previous consumer should stop and new start consuming, prone to mistakes

1. The original system

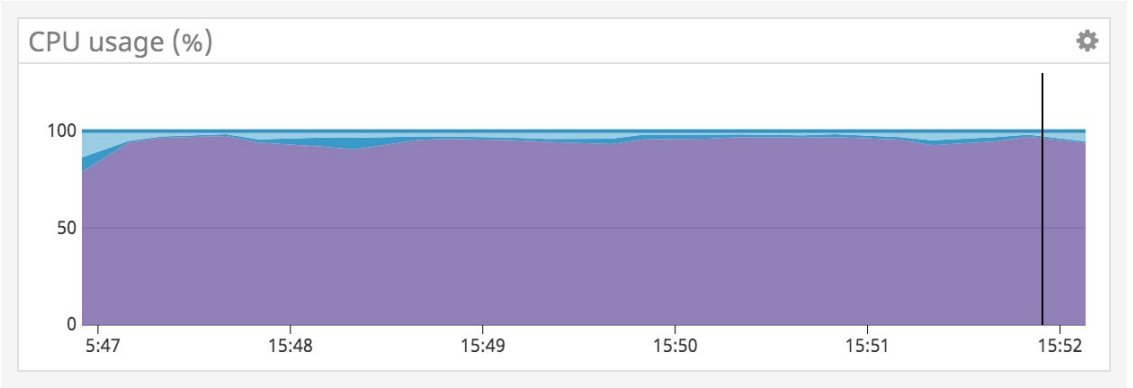
Kafka Topic/Partition 0

Kafka Topic/Partition 1

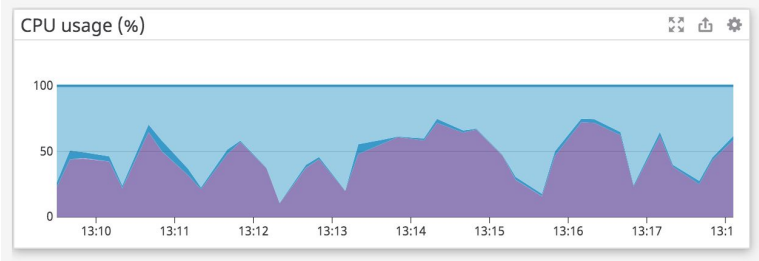
Kafka Topic/Partition 2

Kafka Topic/Partition 3

Host/Consumer



1. The original system



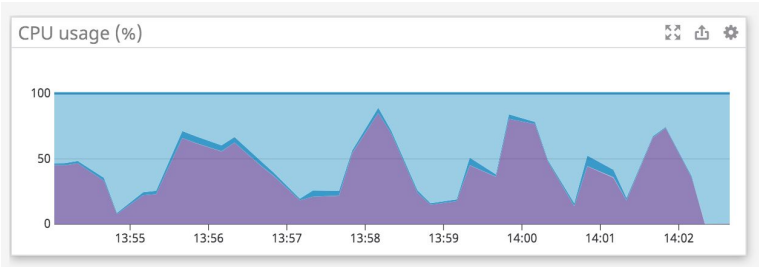
Kafka Topic/Partition 0 → Host/Consumer 0

Kafka Topic/Partition 1 → Host/Consumer 0

Underutilization

Kafka Topic/Partition 2 → Host/Consumer 1

Kafka Topic/Partition 3 → Host/Consumer 1

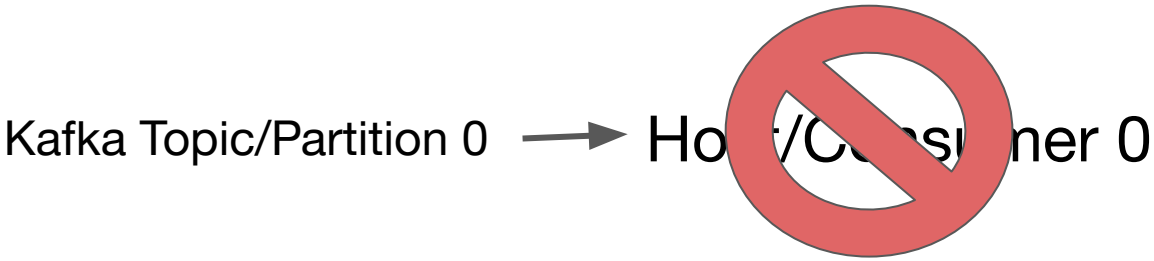


1. The original system

Kafka Topic/Partition 0 → Host/Consumer 0

Once you get to one partition per host and 1M of file descriptors, there's pretty much no room to upscale

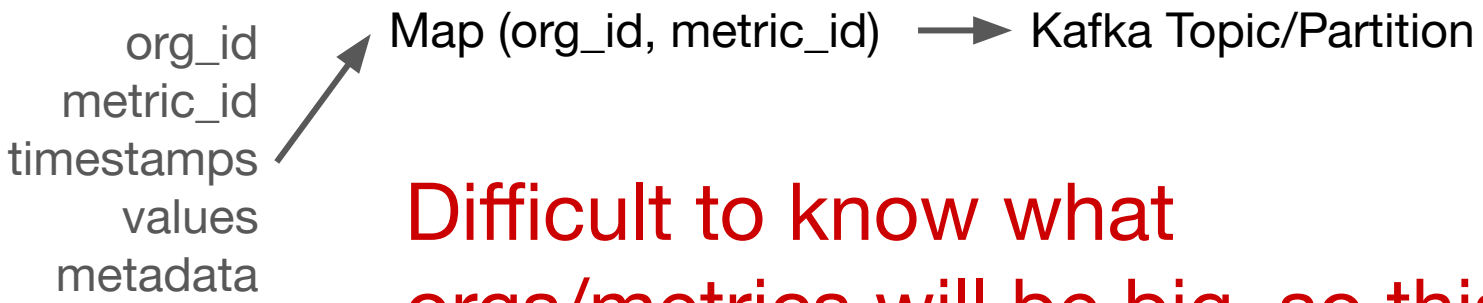
1. The original system



Have to start a new instance, reset offsets, replay data for the past X hours

1. The original system

Payloads



Difficult to know what orgs/metrics will be big, so this model is prone to create hot/big topics/partitions

1. The original system

Payloads

org_id
metric_id
timestamps
values
metadata

Automatically redirects payloads so each kafka topic/partition would be equally sized

Service (org_id, metric_id)

Kafka Topic/Partition 0

Kafka Topic/Partition 1

We have to consume all topics/partitions to get all data for a metric id

2. Requirements to the new system

Conceptual:

1. Must work with the new partitioning schema
2. Must be able to handle 10x growth (2x every year = 3 years)
3. Keep the cost at the same level as the existing system
4. Must be as fast as the existing system

2. Requirements to the new system

Operational:

1. Easily scalable without much manual intervention
2. Minimize impact on kafka (reduce data retention time)
3. Be able to replay data easily

2. Requirements to the new system (RFC)

[tailor-s] Service for producing historical S files #174

Merged **buryat** merged 8 commits into `master` from `historical-s-resolution-files` on Jul 13, 2018

Conversation 68 | Commits 8 | Checks 0 | Files changed 1

Changes from **all commits** | **File filter...** | **Jump to...** | 0 / 1 files

237 rfcs/historical-s-resolution-files/rfc.md

```
@@ -0,0 +1,237 @@
+ # Service for producing historical S files
+
+ - Authors: Vadim Semenov
+ - Date: 2018-06-27
+ - Status: draft
+ - [Discussion](https://github.com/DataDog/architecture/pull/0)
+
+ ## Overview
+
+ Next version of `rawls-extract-4h` that isn't tied to topics&partitions, can scale, and
```

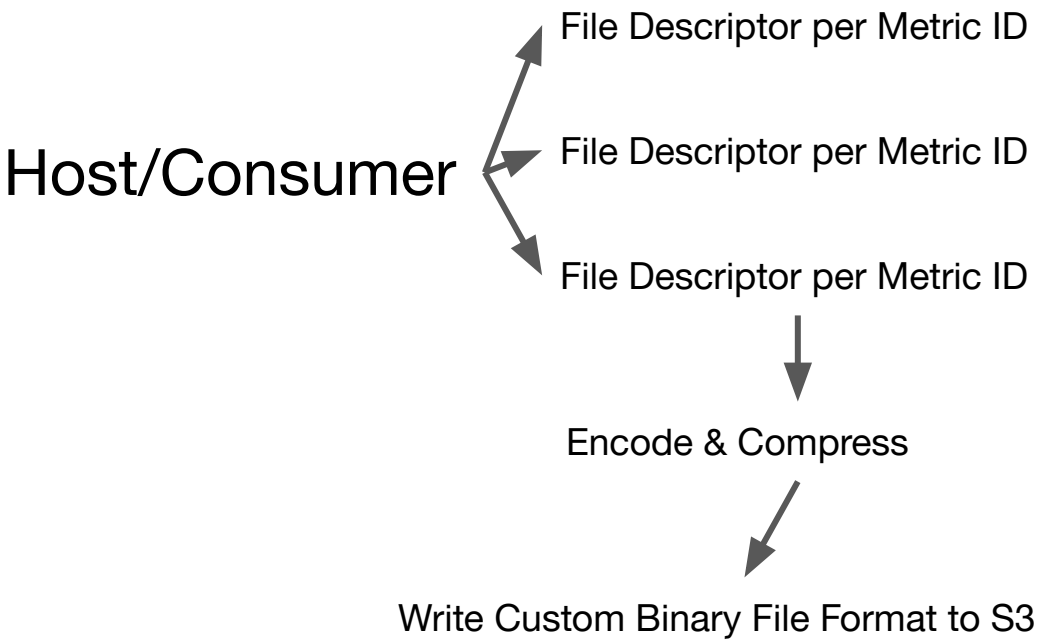
3. Decoupling state and compute

We need to load all topics/partitions to compose a single timeseries. Why not offload kafka to somewhere and then load the whole dataset with Spark?

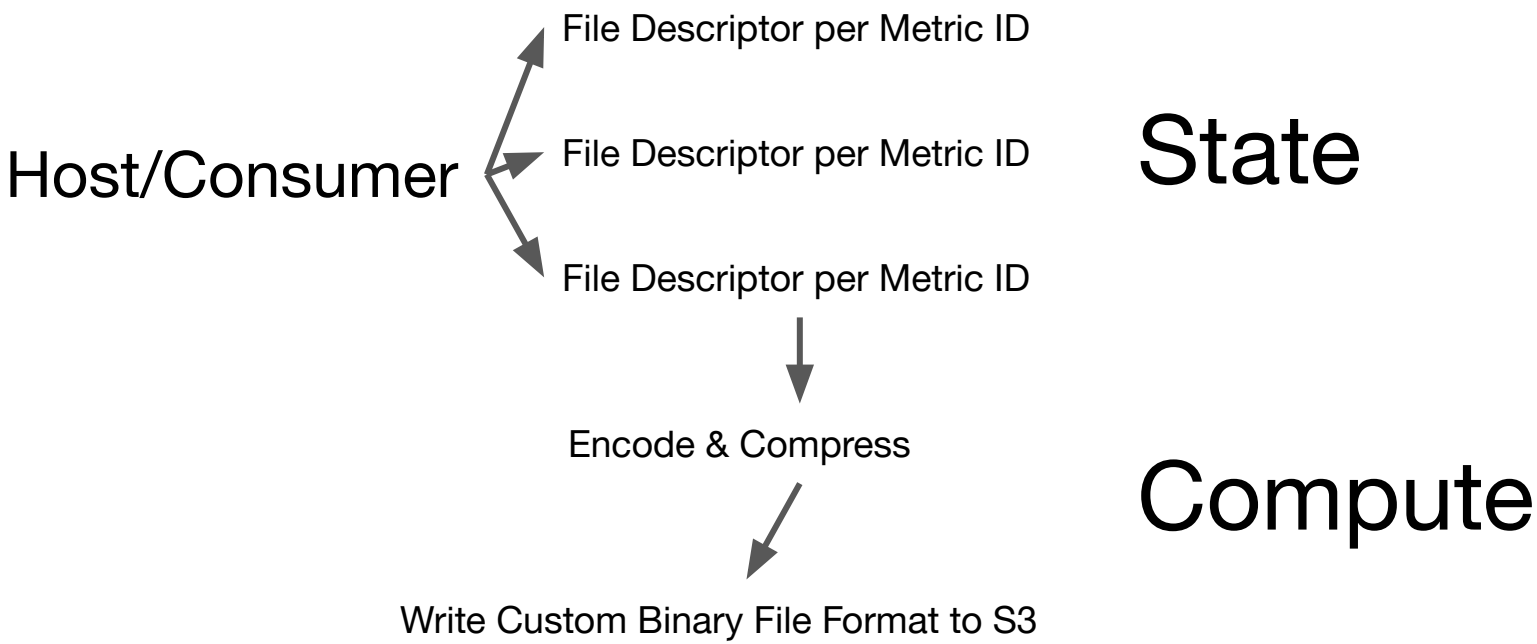
- Taylor Swift

photo by Jana Beamer
https://www.flickr.com/photos/94347223@N07/

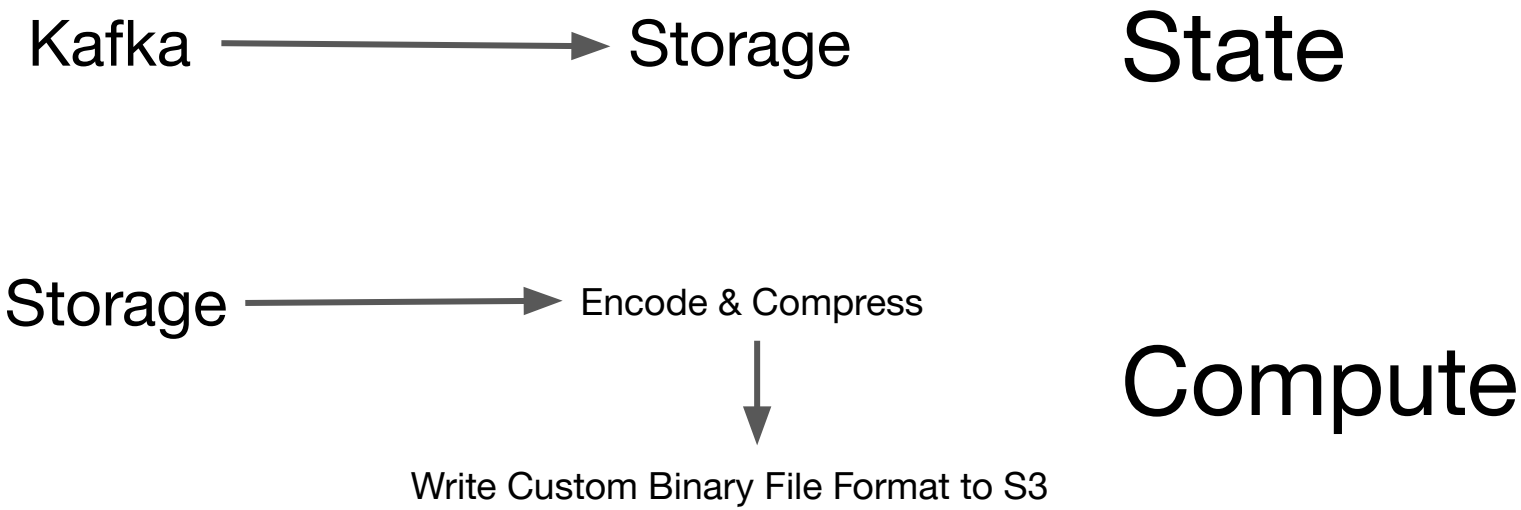
3. Decoupling state and compute



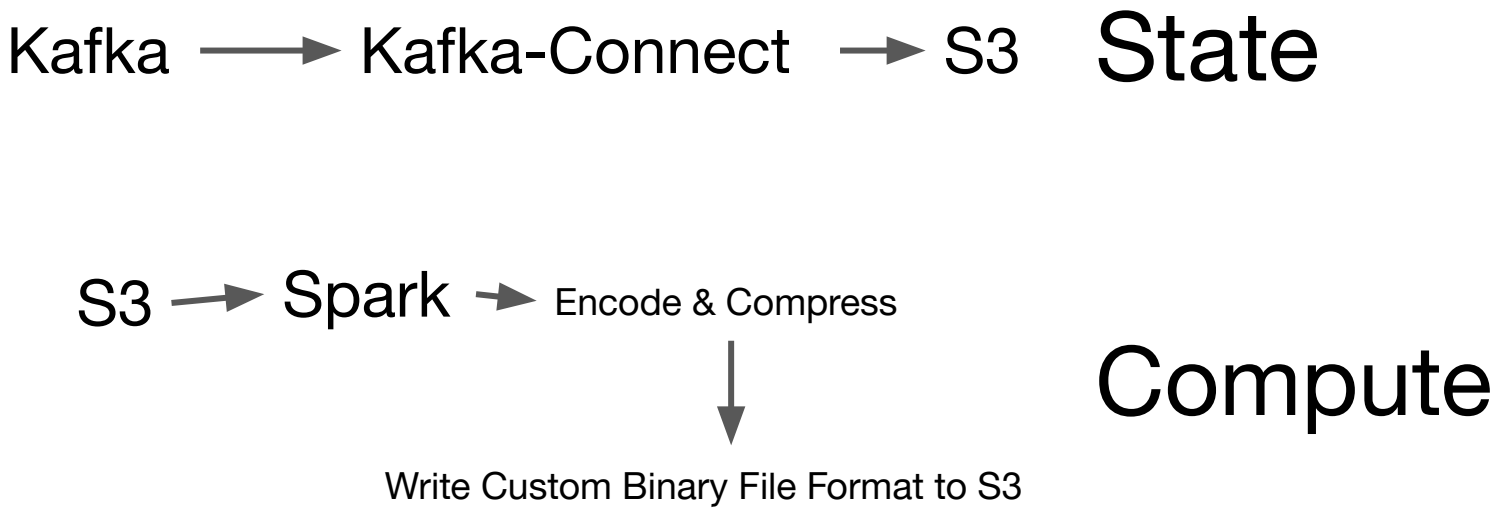
3. Decoupling state and compute



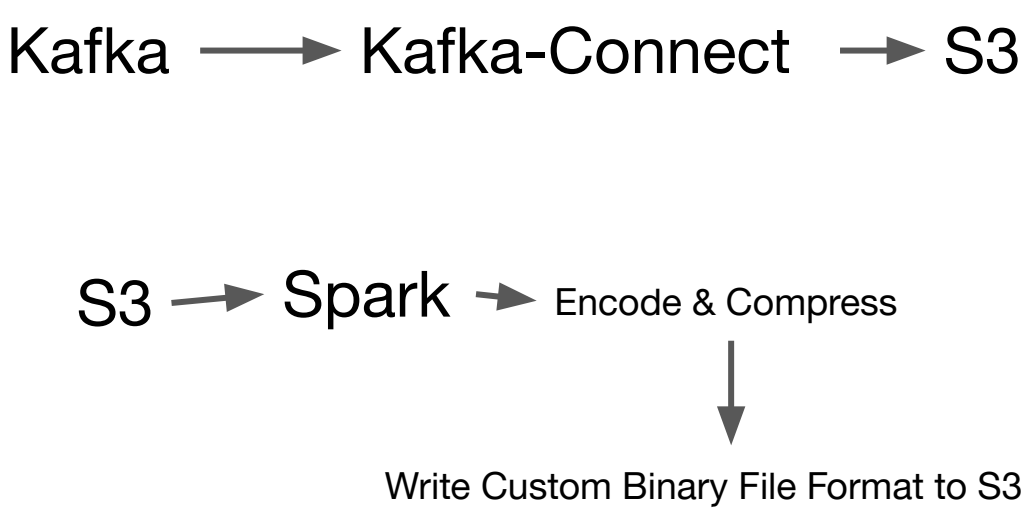
3. Decoupling state and compute



3. Decoupling state and compute



3. Decoupling state and compute

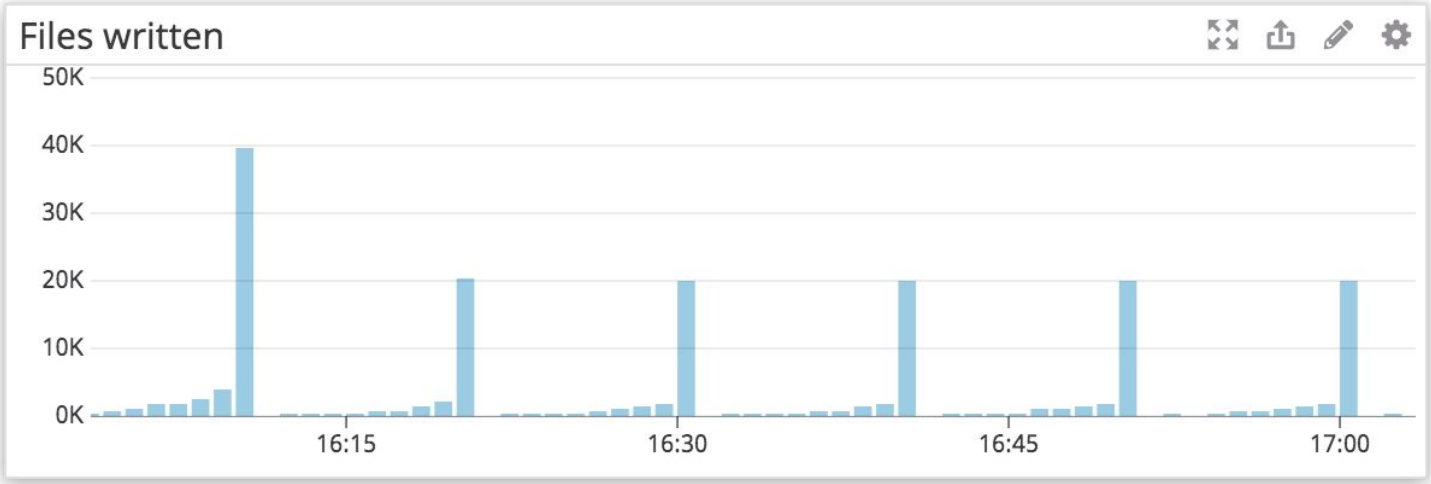


Tailors
Secondary
resolution
data

4. State: Kafka-Connect

https://docs.confluent.io/current/connect/index.html

A really simple consumer, writes payloads as-is to S3 every 10 minutes or once it hits 100k payloads. The goal is to deliver them to S3 as soon as possible with minimum overhead



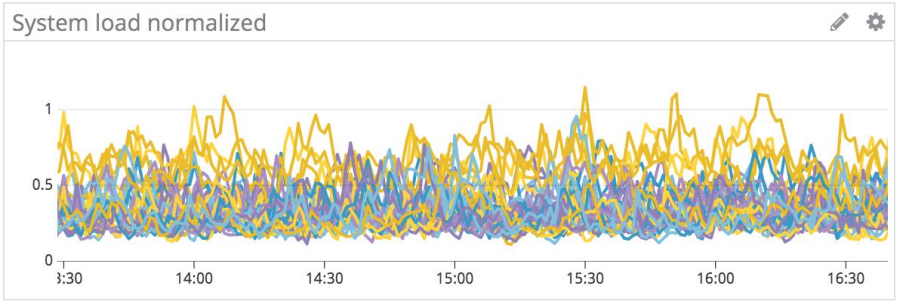
4. State: Kafka-Connect

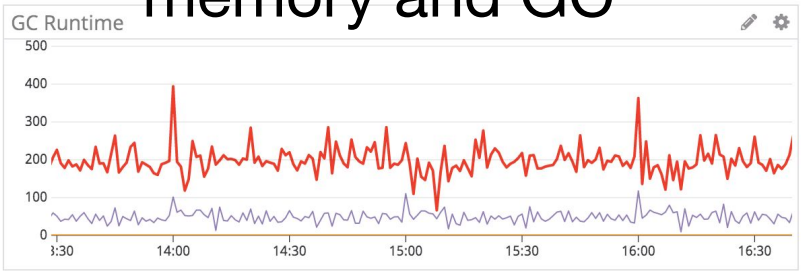
Easy to operate:

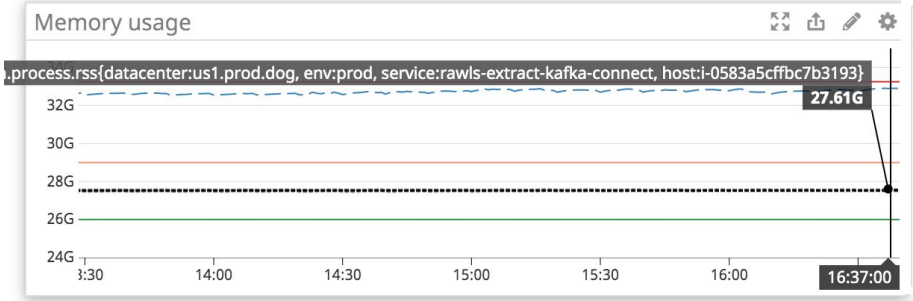
1. "topics": "points-topic-0,points-topic-1" — simply add/remove topics and kafka-connect will rebalance everything across workers automatically.
2. Add/remove workers and it rebalances itself
3. Stopping the system will push it back 10 minutes only — we can reduce kafka retention

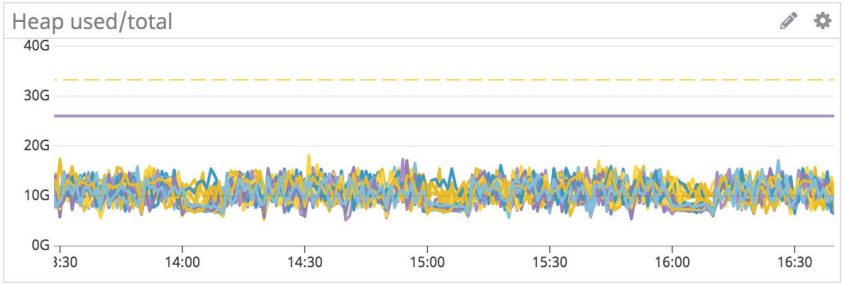
4. State: Kafka-Connect

Keeping an eye on memory and GC

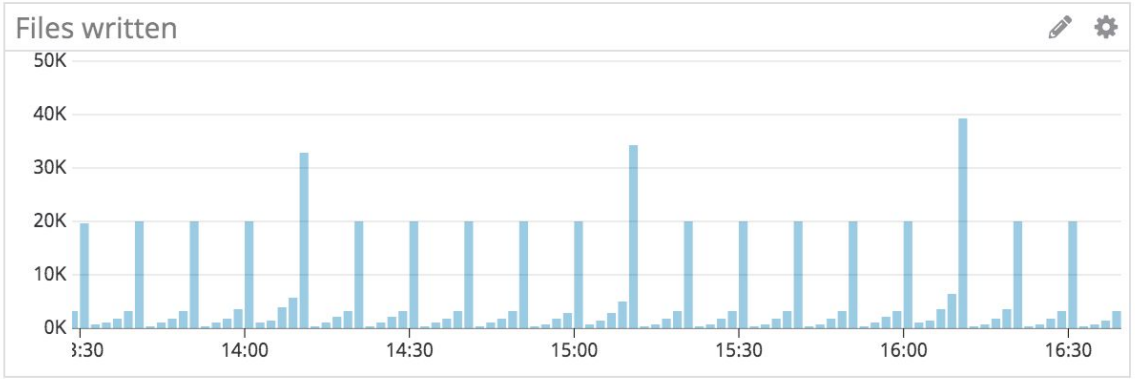


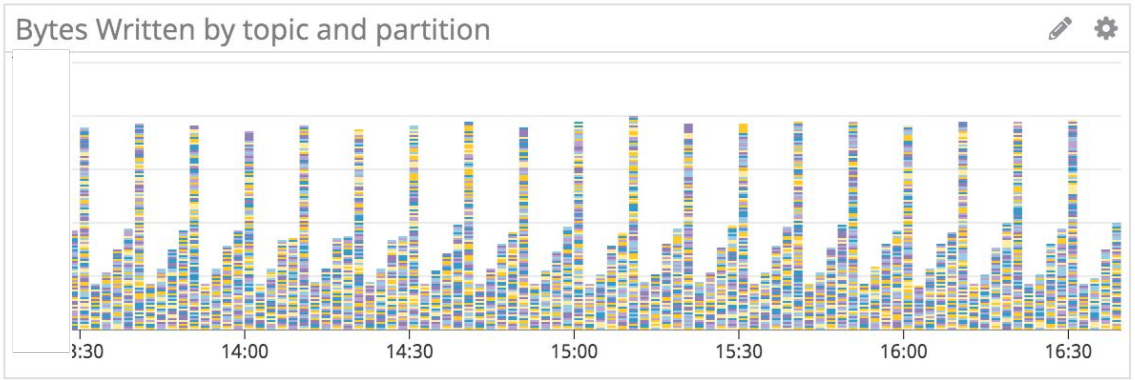






4. State: Kafka-Connect



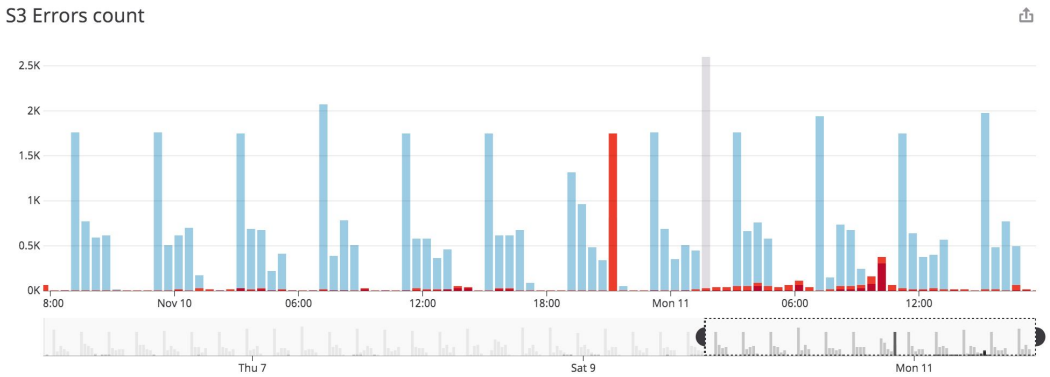


Every 10 minutes we write a lot of data

4. State: Kafka-Connect

Had to optimize writes:

1. Randomized key prefixes, to avoid having hot underlying S3 partitions
2. Parallelize multipart uploads (https://github.com/confluentinc/kafka-connect-storage-cloud/pull/231)
3. Figure out optimal size of buffers to avoid OOMs (we run with s3.part.size=5MiB)
4. Still have lots of 503 Slow Down from S3, so we have exponential backoff for that and monitor retries



5. Compute: Spark

Lost of unknowns: reading 10T points is very difficult:

1. Lots of objects, so we need to minimize GC
2. Figure out how to utilize internal APIs of Spark
3. Is it even possible with Spark??
4. Make it cost-efficient

5. Compute: Spark (Minimizing GC)

Reusing objects:

1. Allocate a 1MiB ByteBuffer once we open a file

5. Compute: Spark (Minimizing GC)

Reusing objects:

1. Allocate a 1MiB ByteBuffer once we open a file
2. Keep decoding payloads (ZSTD) into the allocated memory

5. Compute: Spark (Minimizing GC)

Reusing objects:

1. Allocate a 1MiB ByteBuffer once we open a file
2. Keep decoding payloads (ZSTD) into the allocated memory
3. Get data from the same byte buffer

5. Compute: Spark (FileFormat)

org.apache.spark.sql.execution.datasources.FileFormat

provide a reader of
org.apache.spark.sql.catalyst.InternalRow

then point InternalRow directly to regions of memory in the allocated buffer

5. Compute: Spark (FileFormat)

```
/* 027 */   protected void processNext() throws java.io.IOException {
/* 028 */     while (scan_mutableStateArray_0[0].hasNext()) {
/* 029 */       InternalRow scan_row_0 = (InternalRow) scan_mutableStateArray_0[0].next();
/* 030 */       ((org.apache.spark.sql.execution.metric.SQLMetric) references[0] /* numOutputRows */).add(1);
/* 031 */       do {
/* 032 */         boolean scan_isNull_0 = scan_row_0.isNullAt(0);
/* 033 */         int scan_value_0 = scan_isNull_0 ? -1 : (scan_row_0.getInt(0));
/* 034 */         boolean scan_isNull_3 = scan_row_0.isNullAt(3);
/* 035 */         int scan_value_3 = scan_isNull_3 ? -1 : (scan_row_0.getInt(3));
```

```
550
551     // We keep reusing the same row which helps avoid GC
552     private val singletonRow = new InternalRow {
553       override def numFields: Int = 10
554
555       override def setNullAt(i: Int): Unit = {
```

```
594   override def getInt(ordinal: Int): Int = ordinal match {
595     case ORG_ID            ⇒ orgId
596     case TIMESTAMP         ⇒ timestamp
```

```
482     def orgId: OrgId = _orgId
483     def metricId: MetricId = byteBuffer.getLong(bodyOffset + currentPointOffset)
```

5. Compute: Spark (FileFormat)

```
/* 027 */   protected void processNext() throws java.io.IOException {
/* 028 */     while (scan_mutableStateArray_0[0].hasNext()) {
/* 029 */       InternalRow scan_row_0 = (InternalRow) scan_mutableStateArray_0[0].next();
/* 030 */       ((org.apache.spark.sql.execution.metric.SQLMetric) references[0] /* numOutputRows */).add(1);
/* 031 */       do {
/* 032 */         boolean scan_isNull_0 = scan_row_0.isNullAt(0);
/* 033 */         int scan_value_0 = scan_isNull_0 ? -1 : (scan_row_0.getInt(0));
/* 034 */         boolean scan_isNull_3 = scan_row_0.isNullAt(3);
/* 035 */         int scan_value_3 = scan_isNull_3 ? -1 : (scan_row_0.getInt(3));
```

```
  // We keep reusing the same row which helps avoid GC
  private val singletonRow = new InternalRow {
    override def numFields: Int = 10

    override def getInt(ordinal: Int): Int = ordinal match {
      case ORG_ID           ⇒ orgId
      case TIMESTAMP        ⇒ timestamp
```

```
    def orgId: OrgId = _orgId
    def metricId: MetricId = byteBuffer.getLong(bodyOffset + currentPointOffset)
```

Directly delivers primitives to Spark's memory bypassing creating objects completely

5. Compute: Spark (FileFormat)

Class	Objects	Shallow Size	Retained Size
java.lang.Long	13,443,478 44 %	322,643,472 33 %	≈ 322,643,472 33 %
java.lang.Object[]	3,374,244 11 %	179,464,224 19 %	≈ 181,541,544 19 %
byte[]	1,416,878 5 %	171,718,384 18 %	≈ 171,718,384 18 %
java.lang.Double	3,361,823 11 %	80,683,752 8 %	≈ 80,683,752 8 %
org.apache.spark.sql.catalyst.expressions.UnsafeRow sun.misc.Launcher$AppC	1,387,309 5 %	55,492,360 6 %	≈ 55,492,360 6 %
java.lang.Integer	3,362,624 11 %	53,801,984 6 %	≈ 53,801,984 6 %
org.apache.spark.sql.catalyst.expressions.GenericInternalRow sun.misc.Launch	3,361,751 11 %	53,788,016 6 %	≈ 53,788,016 6 %
char[]	103,367 0 %	14,259,320 1 %	≈ 14,259,320 1 %

Class	Objects	Shallow Size	Retained Size
com.datadog.spark.data.PointRow sun.misc.Launcher$AppClassLoader	3,361,771 36 %	215,153,344 36 %	≈ 215,153,344 36 %
byte[]	1,784,715 19 %	201,485,672 33 %	≈ 201,485,672 33 %
java.util.LinkedList$Node	1,758,087 19 %	42,194,088 7 %	≈ 42,226,968 7 %
org.apache.spark.sql.catalyst.expressions.UnsafeRow sun.misc.Launcher$Ap	1,757,691 19 %	70,307,640 12 %	≈ 70,307,640 12 %
char[]	100,325 1 %	13,672,720 2 %	≈ 13,672,720 2 %
java.lang.String	82,374 1 %	1,976,976 0 %	≈ 8,751,624 1 %

5. Compute: Spark (FileFormat)

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(50)	0	0.0 B / 6.6 TB	0.0 B	1372	28	1	79665	79694	534.7 h (157.4 h)	0.0 B	0.0 B	1.8 TB	0
Dead(14)	0	0.0 B / 1.9 TB	0.0 B	392	0	357	20307	20664	208.4 h (29.8 h)	0.0 B	0.0 B	364.6 GB	0
Total(64)	0	0.0 B / 8.5 TB	0.0 B	1764	28	358	99972	100358	743.0 h (187.2 h)	0.0 B	0.0 B	2.2 TB	0

Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(50)	50	2 MB / 6.6 TB	0.0 B	1372	66	0	99934	100000	388.3 h (64.7 h)	0.0 B	0.0 B	2.1 TB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(50)	50	2 MB / 6.6 TB	0.0 B	1372	66	0	99934	100000	388.3 h (64.7 h)	0.0 B	0.0 B	2.1 TB	0

Executors

5. Compute: Spark (Files > 2GiB)

Can't read files bigger than 2GiB into memory because arrays in java can't have more than 2^31 - 8 elements. And sometimes kafka-connect produces very big files

5. Compute: Spark (Files > 2GiB)

1. Copy a file locally
2. MMap it using com.indeed.util.mmap.MMapBuffer
3. Allocate an empty ByteBuffer using java reflections
4. Point ByteBuffer to a region of memory inside the MMapBuffer
5. Give ByteBuffer to ZSTD decompress
6. Everything thinks that it's a regular ByteBuffer but it's actually a MMap'ed file

5. Compute: Spark (Files > 2GiB)

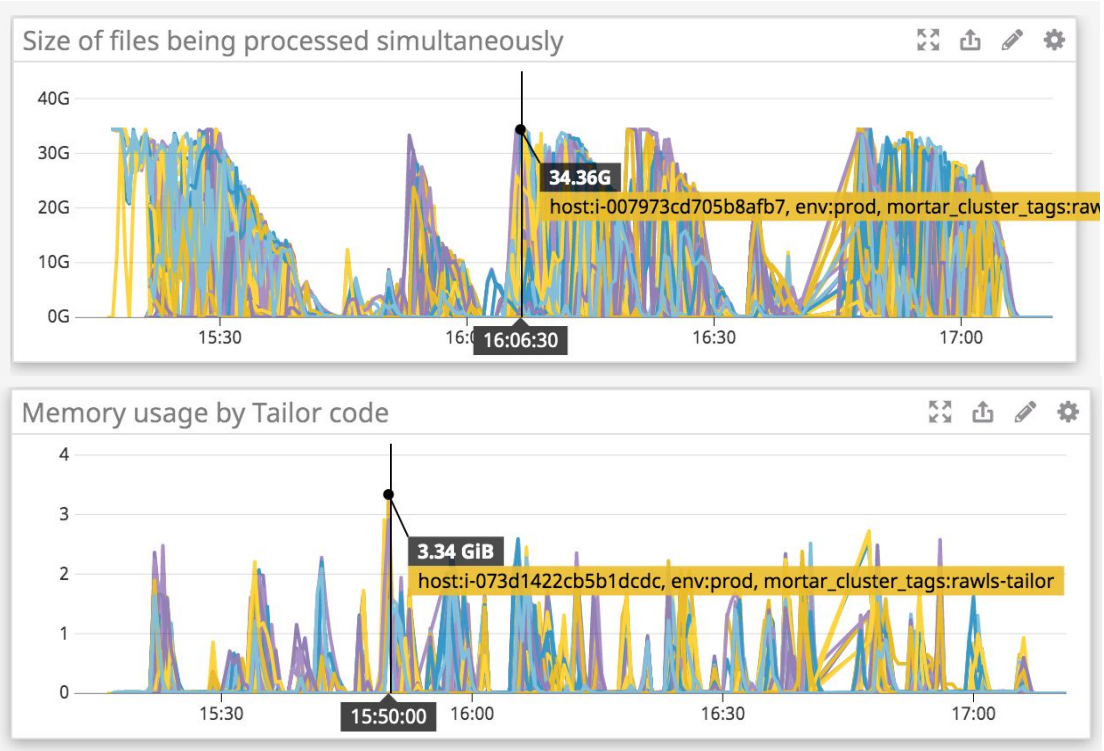
```
42    /**
43     * We create a "fake ByteBuffer" object and point it to the address that directMemory has
44     * so then ZstdUtil could work with a "fake ByteBuffer" so we avoid memory copying.
45     * Zstd uses a C library that reads data from a region of memory and outputs uncompressed
46     * into another region, so we can just point a ByteBuffer to a region of memory where we loaded
47     * compressed data
48     **/
49    def createByteBufferLinkedToDirectMemory(
50      directMemory: DirectMemory,
51      offset: Long,
52      length: Int
53    ): ByteBuffer = {
54      val address = classOf[Buffer].getDeclaredField( name = "address")
55      address.setAccessible(true)
56      val capacity = classOf[Buffer].getDeclaredField( name = "capacity")
57      capacity.setAccessible(true)
58
59      val bb = ByteBuffer.allocateDirect( capacity = 0).order(ByteOrder.nativeOrder)
60      // Let's point the ByteBuffer to the region that bigger DirectMemory has
61      address.setLong(bb, directMemory.getAddress + offset)
62      capacity.setInt(bb, length)
63      // Need to reset the byte buffer position, so zstd uncompress would work correctly
64      bb.limit(length)
65      bb.position( newPosition = 0)
66      bb
67    }
```

5. Compute: Spark (Files > 2GiB)

Some files a very big, so we need to read them in parallel.

1. Set spark.sql.files.maxPartitionBytes=1GB
2. Write length,payload,length,payload,length,payload
3. Each reader will have startByte/endByte
4. Keep skipping payloads until >= startByte

5. Compute: Spark (Files > 2GiB)



Because of lots of tricks we have to track allocation/deallocation of memory in our custom reader. It's very memory efficient, doesn't use more than 4GiB per executor

5. Compute: Spark (Internal APIs)

DataSet.map(obj => …)

1. must create objects
2. copies primitives from Spark Memory (internal spark representation)
3. has schema
4. type-safe

5. Compute: Spark (Internal APIs)

DataSet.queryExecution.toRdd(InternalRow =>)

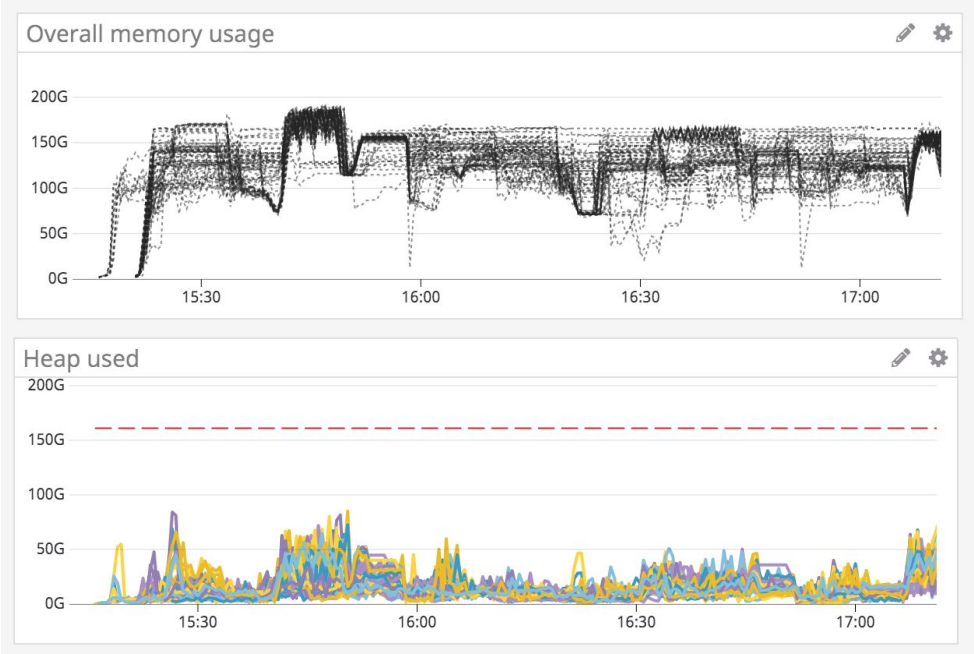
1. doesn't create objects
2. doesn't copy primitives
3. has no schema
4. not type-safe, you need to know position of all fields
5. InternalRow has direct access to Spark memory

5. Compute: Spark (Internal APIs)

```
val df = records
  .groupBy( col1 = "org_id",  cols = "metric_id", "context_k
  // sort_array works for `struct`s, it sorts all tuples
  .agg(
    sort_array(collect_list(struct( colName = "timestamp",
  )
DataFrameUtil.explainPlanWithCodeGenAndCost(df)
// Get the internal version of the RDD. Avoids copies and
// Allows to bypass creating scala objects like tuples ar
// And instead we have to get field values using field po
val rdd = df.queryExecution.toRdd
  .mapPartitions(_.flatMap { row ⇒
    val startTime = System.nanoTime()
    val orgId = row.getInt( ordinal = 0)
    val metricId = row.getLong( ordinal = 1)
```

5. Compute: Spark (Memory)

spark.executor.memory = 150g
spark.yarn.executor.memoryOverhead = 70g
spark.memory.offHeap.enabled = true,
spark.memory.offHeap.size = 100g



5. Compute: Spark (GC)

Here we only compare ratio of GC to task time, screenshots were taken not at the same point within the job

Executors

offheap=false (default setting), almost 50% is spent in GC

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Inp
Active(99)	0	0.0 B / 19.8 TB	0.0 B	2744	2842	0	12848	15690	57.1 h (31.1 h)	0.0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0
Total(99)	0	0.0 B / 19.8 TB	0.0 B	2744	2842	0	12848	15690	57.1 h (31.1 h)	0.0

Executors

offheap=true, GC time drops down to 20%

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Inp
Active(99)	0	0.0 B / 14.7 TB	0.0 B	2744	3	1000	99667	100670	471.8 h (90.4 h)	0.0
Dead(2)	0	0.0 B / 298 GB	0.0 B	56	0	57	329	386	4.0 h (9.4 min)	0.0
Total(101)	0	0.0 B / 15 TB	0.0 B	2800	3	1057	99996	101056	475.8 h (90.6 h)	0.0

5. Compute: Spark (GC)

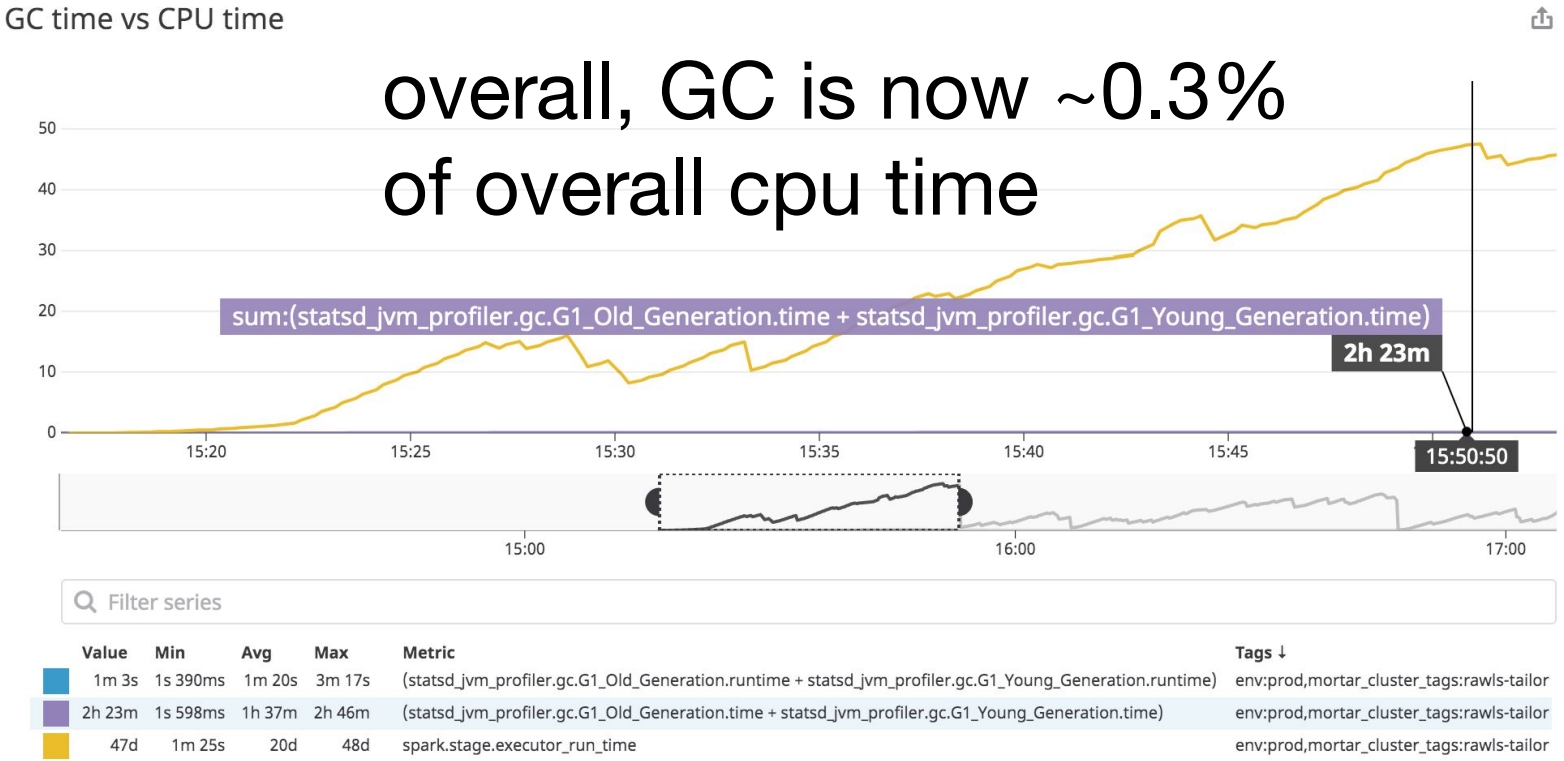
time spent in GC = 63.8/1016.3 = 6.2%

Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(99)	0	0.0 B / 22.5 TB	0.0 B	3136	0	0	66391	66391	963.4 h (60.4 h)	20 TB	10.3 TB	9.7 TB	0
Dead(7)	0	0.0 B / 1.6 TB	0.0 B	224	0	0	2739	2739	52.8 h (3.5 h)	984 GB	520.2 GB	430.9 GB	0
Total(106)	0	0.0 B / 24.1 TB	0.0 B	3360	0	0	69130	69130	1016.3 h (63.8 h)	21 TB	10.8 TB	10.1 TB	0

5. Compute: Spark (GC)



6. Testing

1. Unit tests
2. Integration tests
3. Staging environment
4. Load-testing
5. Slowest parts
6. Checking data correctness
7. Game days

6. Testing (Load testing)

Once we had a working prototype, we started doing load testing to make sure that the new system is going to work for the next 3 years.

1. Throw 10x data
2. See what is slow/what breaks, write it down
3. Estimate cost

6. Testing (Slowest parts)

Have good understanding of the slowest/most skewed parts of the job, put timers around them and have historical data to compare.

And we know limits of those parts and when to start optimizing them.

6. Testing (Slowest parts)

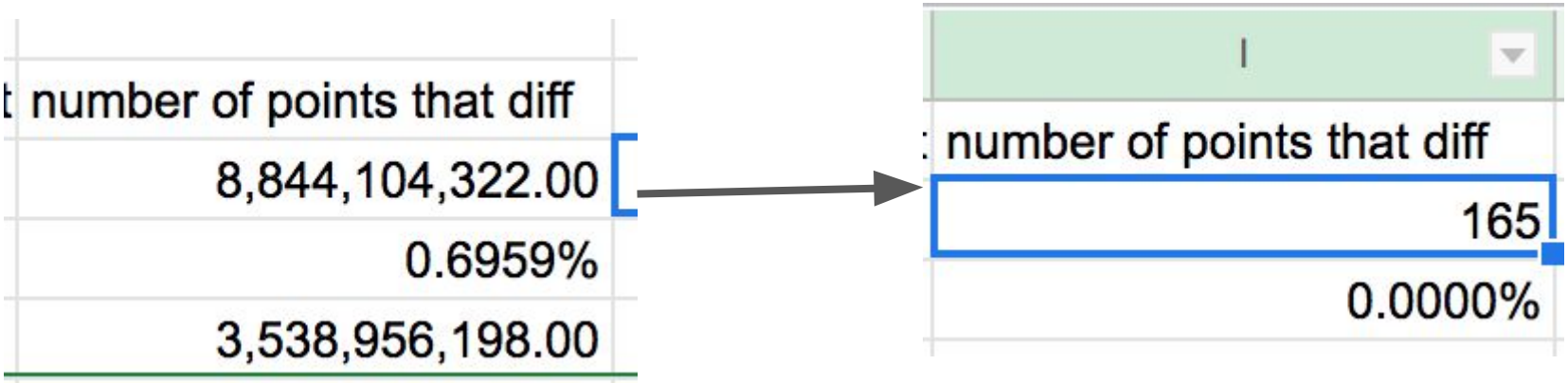
```
18    private val task = new java.util.TimerTask {
19      def run(): Unit = {
20        client.gauge( aspect = "memory", RawlsKafkaConnectMemory.usage)
21        client.gauge( aspect = "size_of_files", RawlsKafkaConnectFileFormat.fileSize)
22        client.count( aspect = "corrupt_readings", RawlsKafkaConnectFileFormat.corruptErrors)
23
24        client.count( aspect = "readingFileMs", RawlsKafkaConnectTimers.getReadingFileMs)
25        client.count( aspect = "readingFileCounts", RawlsKafkaConnectTimers.getReadingFileCounts)
26
27        client.count( aspect = "readingFileInMemoryMs", RawlsKafkaConnectTimers.getReadingFileInMemoryMs)
28        client.count( aspect = "readingFileInMemoryCounts", RawlsKafkaConnectTimers.getReadingFileInMemoryCounts)
29
30        client.count( aspect = "readingFileMmapMs", RawlsKafkaConnectTimers.getReadingFileMmapMs)
31        client.count( aspect = "readingFileMmapCounts", RawlsKafkaConnectTimers.getReadingFileMmapCounts)
32
33        client.count( aspect = "totalDecodingMs", RawlsKafkaConnectTimers.getTotalDecodingMs)
34        client.count( aspect = "totalDecodingCounts", RawlsKafkaConnectTimers.getTotalDecodingCounts)
35
36        client.count( aspect = "zstdDecompressMs", RawlsKafkaConnectTimers.getZstdDecompressMs)
37        client.count( aspect = "zstdDecompressCounts", RawlsKafkaConnectTimers.getZstdDecompressCounts)
38
39        client.count( aspect = "zstdNativeMs", RawlsKafkaConnectTimers.getZstdNativeMs)
40        client.count( aspect = "zstdNativeCounts", RawlsKafkaConnectTimers.getZstdNativeCounts)
41
42        client.count( aspect = "totalDataFrameTimeMs", RawlsKafkaConnectTimers.getTotalDataFrameTimeMs)
43        client.count( aspect = "totalDataFrameTimeCounts", RawlsKafkaConnectTimers.getTotalDataFrameTimeCounts)
44      }
45    }
```

6. Testing (Easter egg)

```
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:     Are you ready for it?
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:         Let the games begin
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                                  .:+oys++++/:-.
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                               ./sddhddssooooooo//---
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                            ./osooshNmdyssyydhyy+oysso+-
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                          -:+hddhssyhho+ooooossssyydhysyho-
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                        -/+smNdysoooo+:-----...--/+ssyhdNds-
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                      .sydmmdhso+:-..          ...-/+oohmdd/
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                      /+hNMNdo/-.              ....:/++ohNMN:
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                      -ohMNh+-.                 ...-:+++ymMMm
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                      +ydNh/-.                   ...--:/+ssdMMo
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                     +hmNy/-..                   ....--:/+shhNMN:
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                    -dNNds:.....---:::--.. ...-/oosssoosydmMMMh
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                    yNMNds-..-----:/+oo+:. .-/ydNNmhyyhdmNNMMMM/
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                   .dNMMms-..--////+osoo+-  .sNNmhsyyhmMMMMMMMMy
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                   :NMMMNs...:+shhdmhss:..  .ohsyssdmNNMNmNNMMMN
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                   +NMMMNo....-:/::::-.      :s+---:/+ossoohNMMd
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                   oNMMMm/..              .. ./sy/.....--:/oyNMMy.
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                   +Ndohh/-...          .+o+smmds-...--:/oymNMMNh-
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                  .yNoyos/:--....      .+yoohNMNm/---:/osdNMMMMMMo
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                  +hm+--o+//::--...     ...--/syy+::/+oymMMMMMMMM:
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                 .+yh+.::-///::--...    -+oosyys////+oyhmNMMMMMMs
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                 -sys+/-. -//::--.....-smNNNMMMMdsossyhdNMMMMMm
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                 +yyhyo/...::--...../hhhysyyyhmNNdoyyyhdNMMMMN:
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                /shhdyshdmmy:--......../syhhdmNNMmysyyhdNMMNm/
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                               :syhhysyydmMN+:--.......-:/+ossssssssydmNMMNm:
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                               .osssdshhhydNh+//:----....   ...-/+sydNMMNdmo
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                                +o+/Nyhddshmy:oso+//::-----://+oymNNNNmddNd
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                              ..+yyomhhdddody/:shhyyyssssssyhhhhdmmmmddmNMd.
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                           .://oohhsmhNNmmhhyo-:oyddddddddddddmmmmmmNNNd.
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                       -:---//+o+syohmNMNmdyhmy-.:/syddmmmmmmmmmmmmNNNNNd.
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                    .+/::::::/+oosydNMMMNsyhmd:.--:+oyhdmmNNNNNNNNNNNNy.
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                   /o/++++ooosyhhmNMMNNyosyhms/--::/+oshdmNNNNNMNNNNo
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:                ..-:-:::://+sysoosydNdyyys:ooosy+:///+++osyyhdmNNNNNMNy
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:              .-:::::::://+/:/++osyhso+++//::/+oo.--/+ossyyhhdmNNNMMMMy-
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:           .-:/++//+/////:::--::://:///+++--/oo--://+osyyhdmmNNNMNMMMNh+:.
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:           ..://:------.....-.--:::-:/+oos//+oo::://+oossyhhdmNNMMMMMMMMMNdy+:.
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:         ..----..-:-::::/-------::::/ooosyho/oo//++ossyhhdmmNMMMMMMMMMMMMNmho:.
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:       .---..---:::::::--------/sssyyoosdoos+++oosyyhdddmmNNMMMMMMMMMMMMMMMMMMNms-
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:      .:-..-------::-:-..-----:+oyhhs/+hyoysooooosyhmmmmmmNNMMMMMMMMMMMMMMMMMMMMMMNs.
19/11/11 23:30:17 INFO RawlsKafkaConnectReporter:      ....--:---::--....-----::/sssyssyds+oo+oooosyhmmmmmmmMMMMMMMMMMMMMMMMMMMMMMMd-
```

6. Testing (Data correctness)

We ran the new system using all the data that we have and then did one-to-one join to see what points are missing/different. This allowed us find some edge cases that we were able to eliminate



6. Testing (Game Days)

"Game days" are when we test that our systems are resilient to errors in the ways we expect, and that we have proper monitoring of these situations. If you're not familiar with this idea, https://stripe.com/blog/game-day-exercises-at-stripe is a good intro.

1. Come up with scenarios (a node is down, the whole service is down, etc.)
2. Expected behavior?
3. Run scenarios
4. Write down what happened
5. Summarize key lessons

6. Testing (Game Days)

Test 1: All Rawls-Extract-Kafka-Connect nodes are down

Results

Expected results	Actual results	Comments
`rawls-extract-kafka-connect stopped consuming data` should fire https://app.datadoghq.com/monitors/8719086	FAIL	We don't have monitors on staging, so nothing fired but we observed that consumer lag increased
ASG should bring nodes up	PASS	10:39 - we terminated all nodes 10:47 - first node started consuming
New nodes should start consuming from previous point and the lag should start dropping	PASS	
Files should appear on S3 after restart	PASS	10:49 - first file appeared on S3

6. Testing (Game Days)

Test 3: Slow Rawls-Extract-Kafka-Connect node

Increased CPU load

Prerequisites

- Rawls-Extract-Kafka-Connect working normally
- Pick a rawls-extract-kafka-connect node
- Install stress command `sudo apt-get install stress`

Action

- `date; sudo stress --cpu 8 --timeout 60`

7. Sharding

Once we confirmed that our prototype works using the whole volume of data, we decided to split the job into shards:

1. We use spot instances, so losing a single job for a shard will not result in losing the whole progress.
2. If for some reason there's an edge case, it'll only affect a single shard.
3. Ability to process shards on completely separate clusters.

7. Sharding

We need to identify independent blocks of data, and in our case it's orgs level since one org's data doesn't depend on other org's data.

Kafka-Connect using config file decides in which shard an org would go:

1. org-mod-X (we have 64 shared shards)
2. org-X (org's own shard)

7. Sharding

We know that a single job can process all the data we have.

And now we have 64x shards which means that a single shard can grow up to 64x times until we reach the same volume.

If our volume of data continues doubling every year, that would be enough for next 6 years after which we can increase number of shards.

8. Migrations

In order to replace existing system we need to do lots of things:

1. Run both systems alongside

8. Migrations

In order to replace existing system we need to do lots of things:

1. Run both systems alongside
2. Figure out a release plan and a rollback plan
3. Make sure that systems that depend on our data work fine with both
4. Do partial migrations of customers
5. Check everything
6. Do final migration

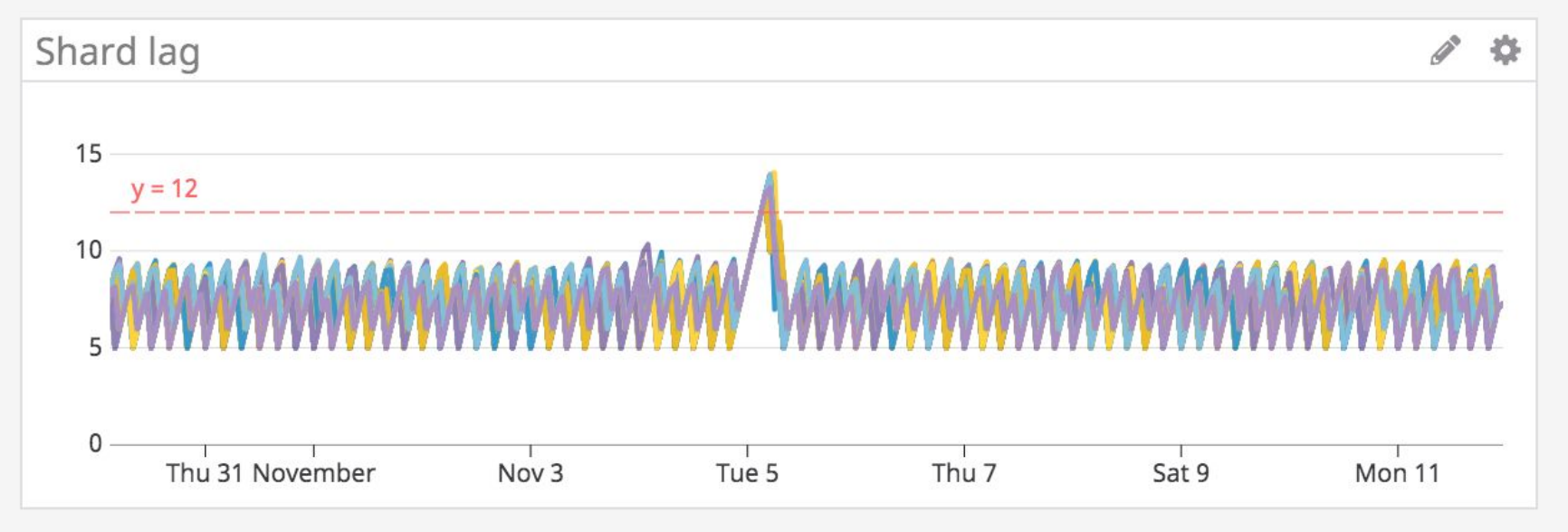
8. Migrations (Final migration)

1. Picked a date, added additional integration tests
2. Tested on staging
3. Rolled in production
4. Let the old system run for a week
5. Kill the old system
6. Cleanup

9. Results (Cost)

Old system	100%
New system	
Kafka Connect compute costs	13%
Kafka Connect storage costs	39%
Spark compute costs	77%
Kafka retention savings	-163%
Total without Kafka savings	129%
Total	-34%
Savings	134%

9. Results (Speed)



9. Results (high-level)

1. ✔️ Must work with new partitioning schema
2. ✔️ Must be able to handle 10x growth (2x every year = 3 years)
3. ✔️ Keep the cost at the same level as the existing system
4. ✔️ Must be as fast as the existing system

9. Results (Operational)

1. ✔️ Easily scalable without much manual intervention
   a. Both storage and compute can scale independently
2. ✔️ Minimize impact on kafka
   a. We reduced data retention in kafka
   b. We actually store kafka data in S3 2x longer, so we actually increased retention
3. ✔️ Be able to replay data easily
   a. We had to replay kafka-connect and spark jobs many times and it was easy