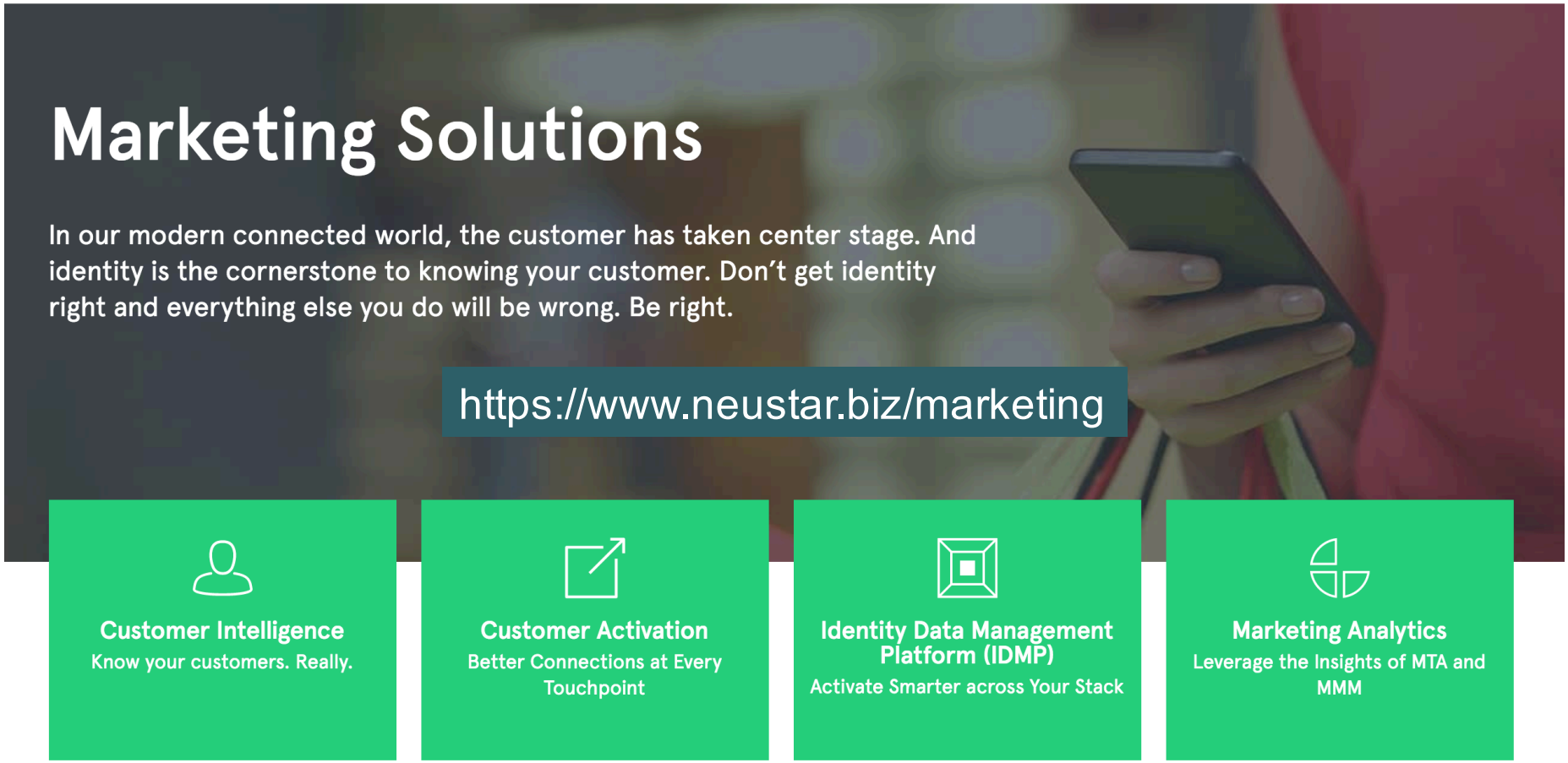
Optimal Strategies for Large-Scale Batch ETL Jobs

Emma Tang, Neustar

#EUDev3

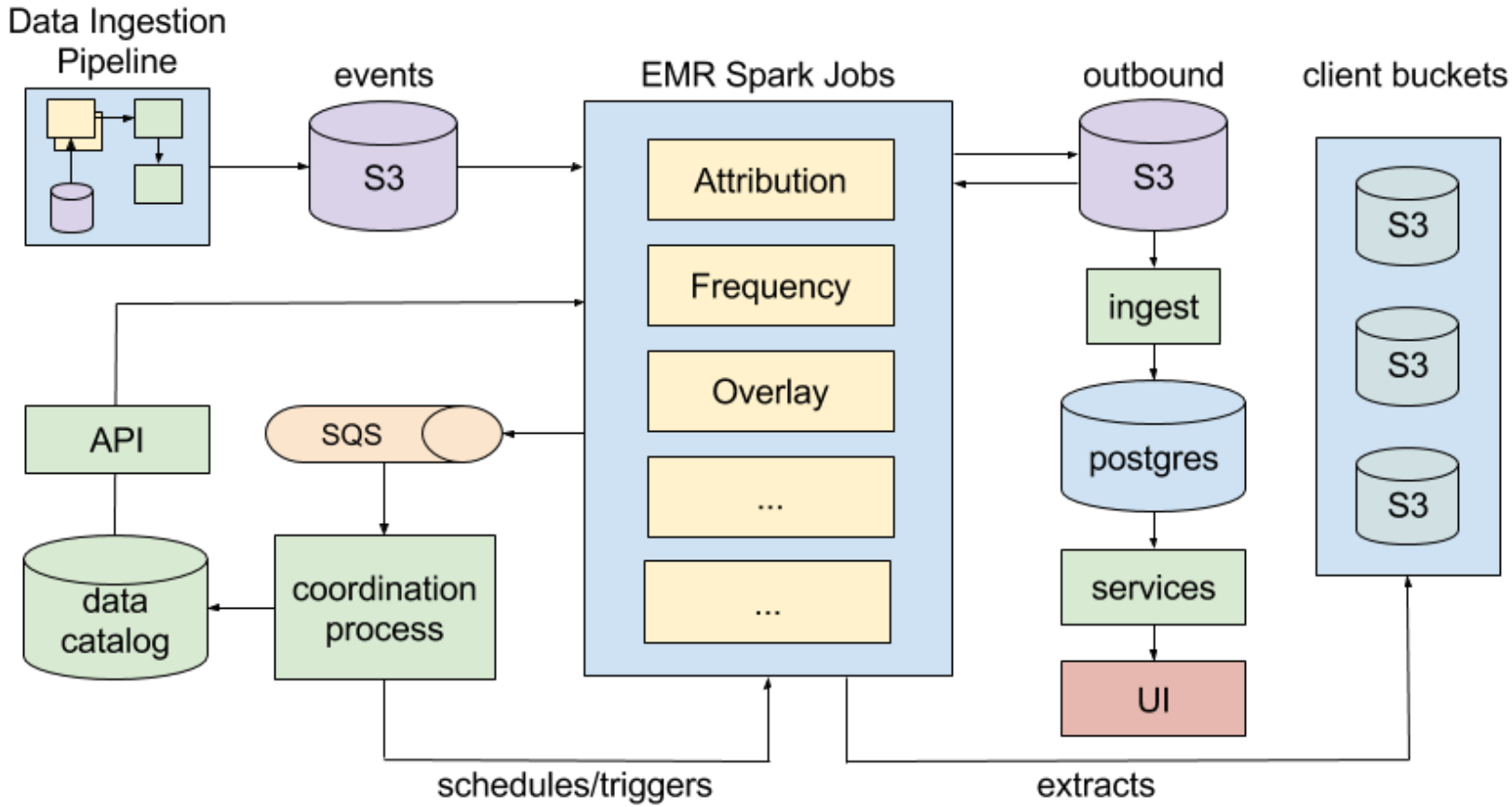
October, 2017


neustar
Marketing
Risk
Security
Registry
Communications
Industries
Clients
Resources
Marketing Solutions
In our modern connected world, the customer has taken center stage. And identity is the cornerstone to knowing your customer. Don't get identity right and everything else you do will be wrong. Be right.
https://www.neustar.biz/marketing
Customer Intelligence
Know your customers. Really.
Customer Activation
Better Connections at Every Touchpoint
Identity Data Management Platform (IDMP)
Activate Smarter across Your Stack
Marketing Analytics
Leverage the Insights of MTA and MMM

Neustar

- Help the world's most valuable brands understand and target their consumers both online and offline
- Maximize ROI on Ad spend
- Billions of user events per day, petabytes of data

Architecture (simplified view)



Batch ETL

- Runs on schedule/ programmatically triggered
- Aim for complete utilization of cluster resources, esp. memory and CPU

Why Batch?

- We care about historical state
- We don’t have SLA other than 1-3x daily delivery
- Efficient, tuned optimal use of resources, cost efficiency

What we will talk about today

- Issues at scale
- Skew
- Optimizations
- Ganglia

The attribution problem

- At Neustar, we process large quantities of ad events
- Familiar events like: impressions, clicks, conversions
- Which impression/click contributed to conversion?

Example attribution

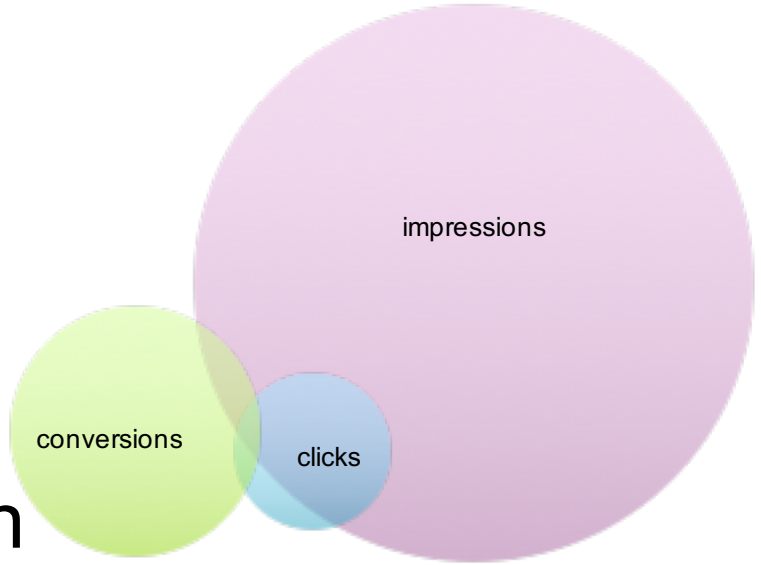
- Alice goes to her favorite news site, and sees 3 ads – impressions
- She clicks on one of them that leads to Macy's – click
- She buys something on Macy's – conversion
- Her purchase can be attributed to the click and impression events

The approach

- Join conversions with impressions and clicks on userId
- Go through each user and attribute conversions to correct target event (impressions/clicks)
- Latest target events are valued more, so timestamp matters

The scale

- Impression: 250 billion
- Clicks: 20 billion
- Conversions: 50 billion



- Join 50 billion x 250 billion

What we will talk about today

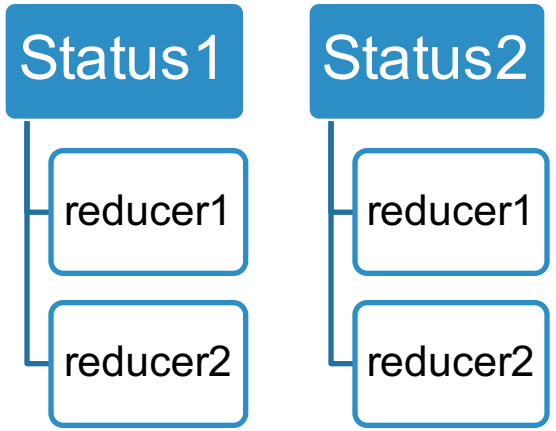
- **Issues at scale**
- Skew
- Optimizations
- Ganglia

Driver OOM

```
Exception in thread "map-output-dispatcher-12" java.lang.OutOfMemoryError
  at java.io.ByteArrayOutputStream.hugeCapacity(ByteArrayOutputStream.java:123)
  at java.io.ByteArrayOutputStream.grow(ByteArrayOutputStream.java:117)
  at java.io.ByteArrayOutputStream.ensureCapacity(ByteArrayOutputStream.java:93)
  at java.io.ByteArrayOutputStream.write(ByteArrayOutputStream.java:153)
  at java.util.zip.DeflaterOutputStream.deflate(DeflaterOutputStream.java:253)
  at java.util.zip.DeflaterOutputStream.write(DeflaterOutputStream.java:211)
  at java.util.zip.GZIPOutputStream.write(GZIPOutputStream.java:145)
  at java.io.ObjectOutputStream$BlockDataOutputStream.writeBlockHeader(ObjectOutputStream.java:1894)
  at java.io.ObjectOutputStream$BlockDataOutputStream.drain(ObjectOutputStream.java:1875)
  at java.io.ObjectOutputStream$BlockDataOutputStream.setBlockDataMode(ObjectOutputStream.java:1786)
  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1189)
  at java.io.ObjectOutputStream.writeArray(ObjectOutputStream.java:1378)
  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1174)
  at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)
  at org.apache.spark.MapOutputTracker$$anonfun$serializeMapStatuses$1.apply$mcV$sp(MapOutputTracker.scala:615)
  at org.apache.spark.MapOutputTracker$$anonfun$serializeMapStatuses$1.apply(MapOutputTracker.scala:614)
  at org.apache.spark.MapOutputTracker$$anonfun$serializeMapStatuses$1.apply(MapOutputTracker.scala:614)
  at org.apache.spark.util.Utils$.tryWithSafeFinally(Utils.scala:1287)
  at org.apache.spark.MapOutputTracker$.serializeMapStatuses(MapOutputTracker.scala:617)
```

Driver OOM

- Array of mapStatuses of size m, each status contains info about how the block is used by each reducer (n).
- m (mappers) x n (reducers)



Driver OOM

- 2 types of mapStatus: highly compressed vs compressed
- HighlyCompressedMapStatus tracks reduce partition average size, with bitmap tracking which blocks are empty for each reducer

Driver OOM

- Reduce number of partitions on either side
- 300k x 75k → 100k x 75k

Disable unnecessary GC

- spark.cleaner.periodicGC.interval
- GC cycles “stop the world”.
- Large heaps means longer GC
- Set to a long period (e.g. twice the length of your job)

Disable unnecessary GC

- ContextCleaner uses weak references to keep track of every RDD, ShuffleDependency, and Broadcast, and registers when the objects go out of scope
- periodicGCService is a single-thread executor service that calls the JVM garbage collector periodically

Allow extra time

- spark.rpc.askTimeout
- spark.network.timeout
- in case of GC, our heap size is so large, we will exceed the timeout.

```
org.apache.spark.SparkException: Error communicating with MapOutputTracker
  at org.apache.spark.MapOutputTracker.askTracker(MapOutputTracker.scala:114)
  at org.apache.spark.MapOutputTracker.getStatuses(MapOutputTracker.scala:212)
  at org.apache.spark.MapOutputTracker.getMapSizesByExecutorId(MapOutputTracker.scala:152)
  at org.apache.spark.shuffle.BlockStoreShuffleReader.read(BlockStoreShuffleReader.scala:47)
```

Spurious failures

- Reading from s3 can be flaky, especially when reading millions of files
- Set spark.task.maxFailures higher than default of 3
- We set to < 10 to ensure true errors propagate out quickly

What we will talk about today

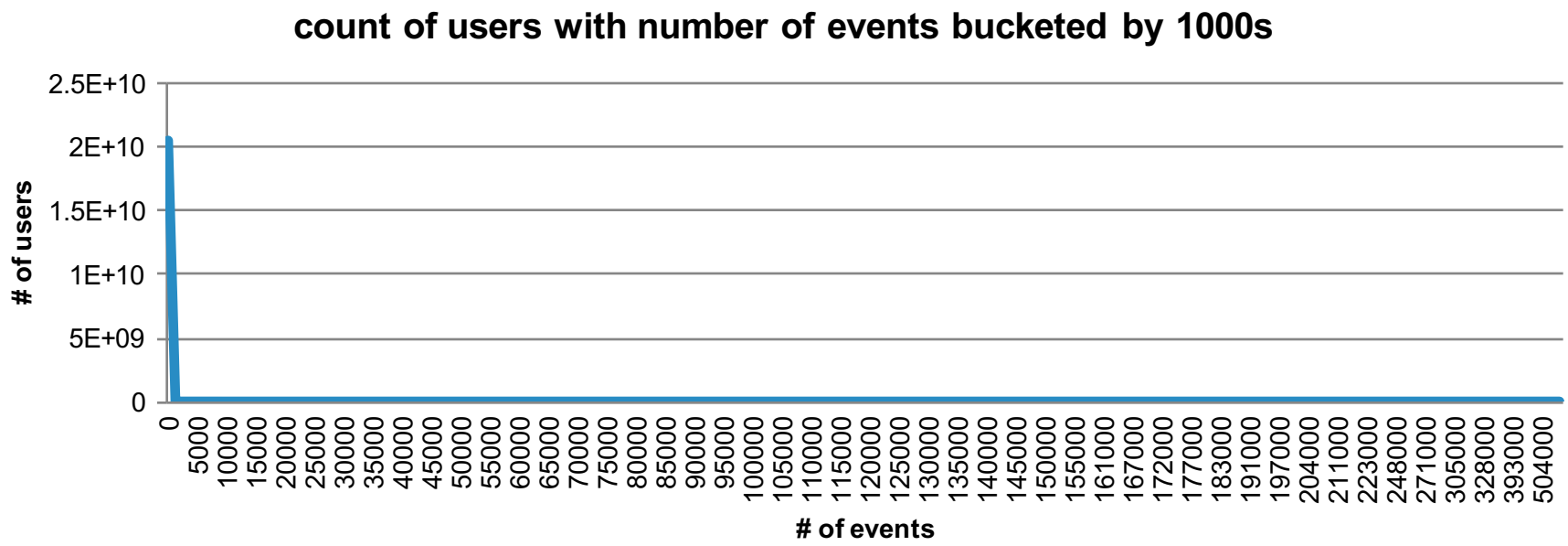
- Issues at scale
- **Skew**
- Optimizations
- Ganglia

The skew

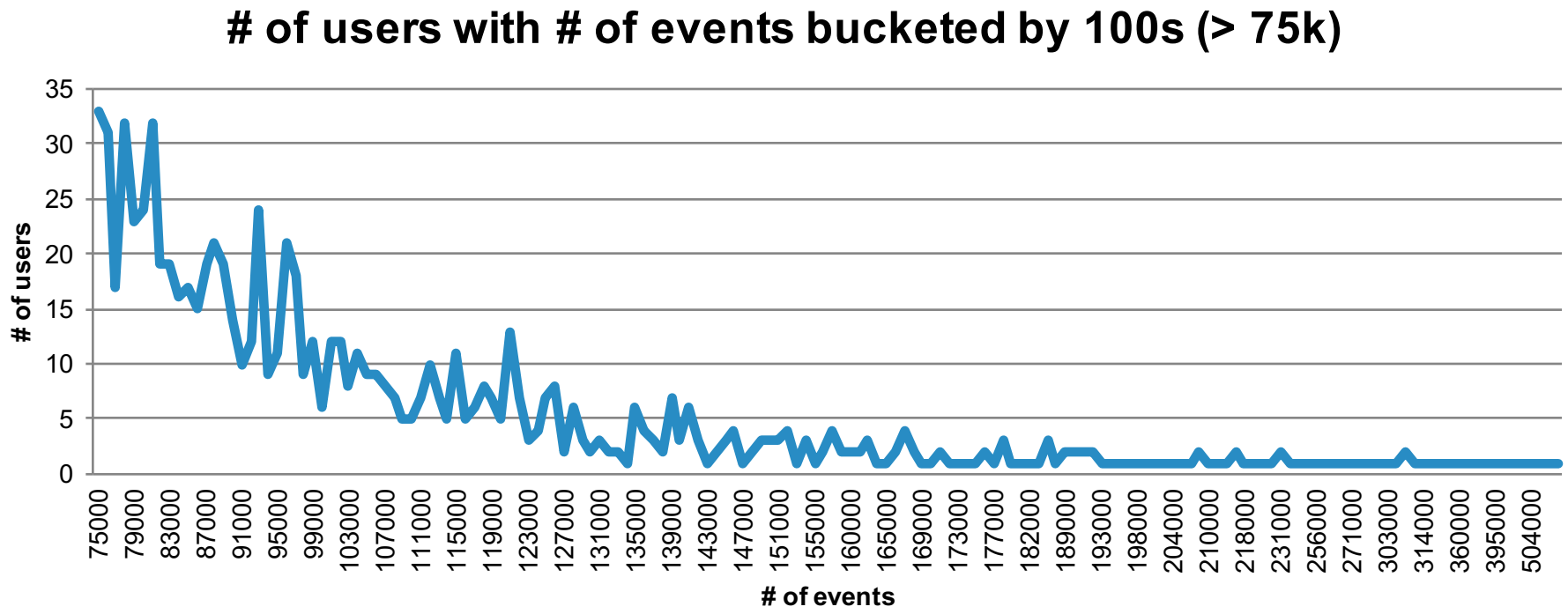
- Extreme skew in data
- A few users have 100k+ events for 90 days. The average user has < 50
- Executors dying due to handful of extremely large partitions

The skew

- Out of 20.5B users, 20.2B have < 50 events



The skew: zoom



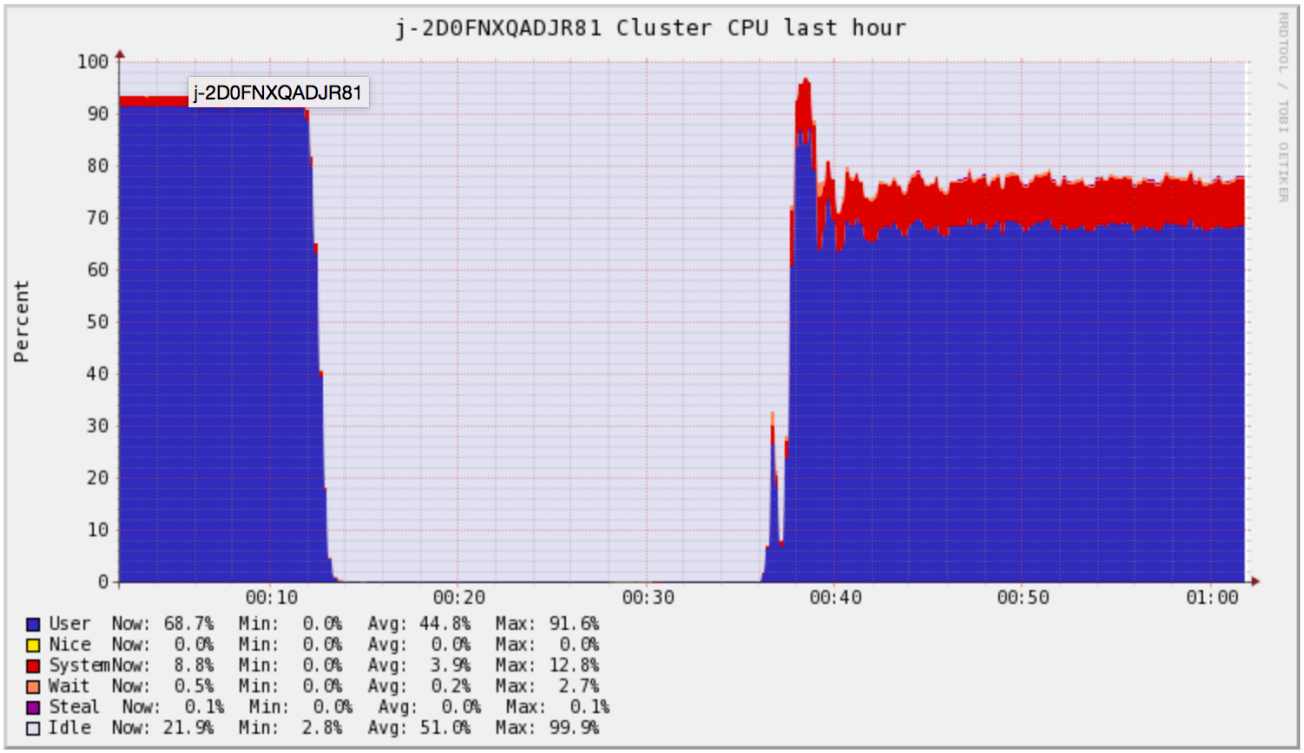
Strategy: increase # of partitions

- First line of defense - increase number of partitions so skewed data is more spread out

Strategy: Nest

- Group conversions by userId, group target event by userId, then join the lists
- Avoid cartesian joins

Long tail: ganglia



Long tail: Spark UI

- 50 min long tail, median 24 s

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	10 s	21 s	24 s	28 s	50 min
Scheduler Delay	3 ms	8 ms	9 ms	14 ms	0.9 s
Task Deserialization Time	0 ms	1 ms	2 ms	2 ms	4 s
GC Time	0.6 s	2 s	3 s	3 s	1.3 min
Result Serialization Time	0 ms	0 ms	0 ms	0 ms	0.2 s
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	181.2 MB	294.2 MB	340.2 MB	406.1 MB	3.4 GB
Shuffle Read Blocked Time	4 ms	19 ms	25 ms	35 ms	0.3 s
Shuffle Read Size / Records	50.4 MB / 179161	53.0 MB / 188604	53.6 MB / 190736	54.3 MB / 193106	117.5 MB / 217526
Shuffle Write Size / Records	33.2 MB / 65672	34.2 MB / 66511	34.4 MB / 66685	34.7 MB / 66858	83.4 MB / 67983

Long tail: what else to do?

- If you have domain specific knowledge of your data, use it to filter “bad” data out
- Salt your data, and shuffle twice (but shuffling is expensive)
- Use bloom filter if one side of your join is much smaller than the other

Bloom Filter

- Space-efficient probabilistic data structure to test whether an element is a member of a set
- Size mainly determined by number of items in the filter, and the probability of false positives
- No false negatives!
- Broadcast filter out to executors

Bloom Filter

- Using a high false positive rate, still very good filter
- P = 5% -> 80% filtered out
- Subsequent join much faster

Bloom Filter

- Tradeoff between accuracy & size
- We've had great success with Bloom Filters with size of < 5G
- Experiment with Bloom Filters

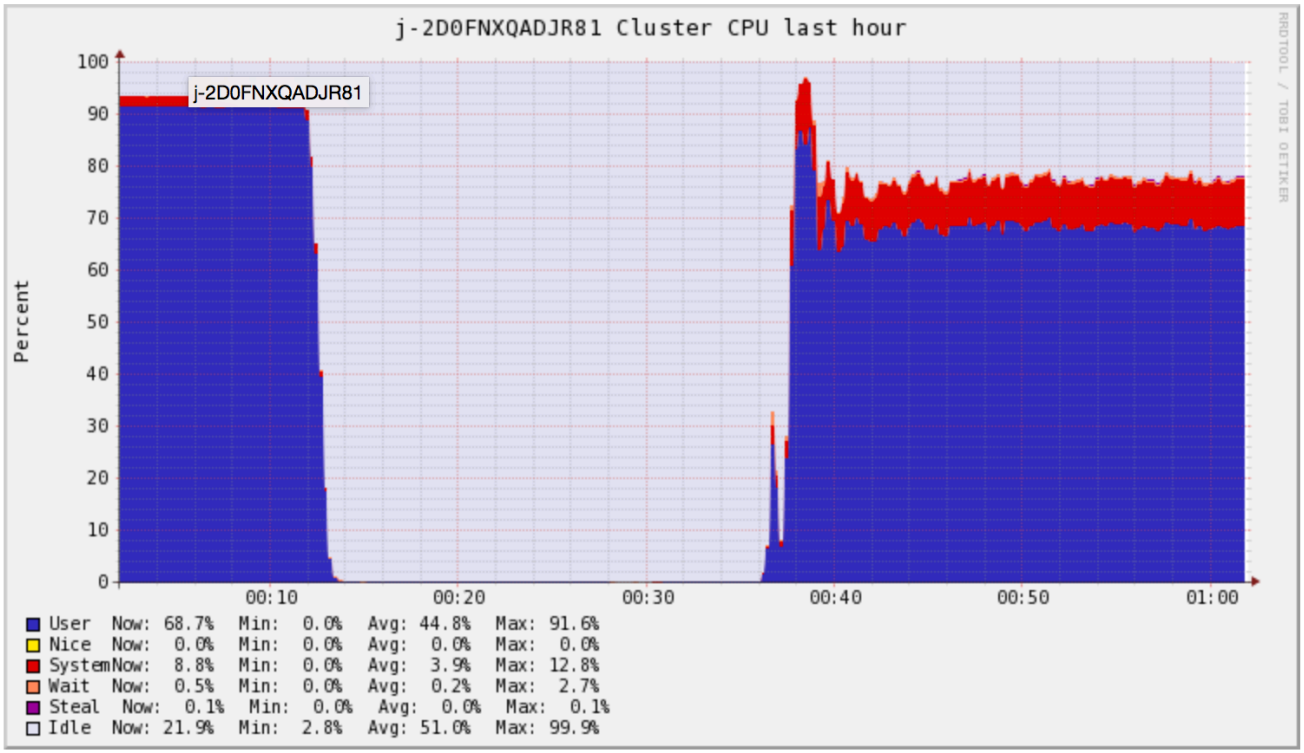
Bloom Filter Applied

- For conversions of 50 billion, false positive rate of 0.1%, filter size is 80GB
- False positive rate of 5%, filter size is 35GB
- Still too big

Long tail: what else to do?

- If you have domain specific knowledge of your data, use it to filter “bad” data out
- Salt your data, and shuffle twice (but shuffling is expensive)
- Use bloom filter if one side of your join is much smaller than the other

Long tail: ganglia



Long tail: what is it doing?

- Look at executor threads during long tail

```
com.esotericsoftware.kryo.util.IdentityObjectIntMap.clear(IdentityObjectIntMap.java:382)
com.esotericsoftware.kryo.util.MapReferenceResolver.reset(MapReferenceResolver.java:65)
com.esotericsoftware.kryo.Kryo.reset(Kryo.java:865)
com.esotericsoftware.kryo.Kryo.writeClassAndObject(Kryo.java:630)
org.apache.spark.serializer.KryoSerializationStream.writeObject(KryoSerializer.scala:209
)
org.apache.spark.serializer.SerializationStream.writeValue(Serializer.scala:134)
org.apache.spark.storage.DiskBlockObjectWriter.write(DiskBlockObjectWriter.scala:239)
org.apache.spark.util.collection.WritablePartitionedPairCollection$$anon$1.writeNext(Wri
tablePartitionedPairCollection.scala:56)
org.apache.spark.util.collection.ExternalSorter.writePartitionedFile(ExternalSorter.scal
a:699)
```

Long tail: what is it doing?

- Mappers writing to shuffle space taking long
- Need to reduce data size **before** going into shuffle

Long tail: what is it doing?

- Events in the long tail had almost identical information, spread over time.
- For each user, if we retain just 1 event per hour, at 90 days, it is around 2k events.
- However, this means we need to group by user first, which requires a shuffle, which defeats the whole purpose of this exercise, right?

Strategy: Filter during map side combine

- Use combineByKey and maximize map side combine
- Thin collection out during map side combine -> less is written to shuffle space

```
eventRDD.combineByKey(
    event -> {
        final List<TargetEvent> list = new ArrayList<>();
        list.add(event);
        return list;
    },
    (listOfEvents, event) -> {
        listOfEvents.add(event);
        if (listOfEvents.size() < MAX_USER_CHAIN_SIZE) {
            return listOfEvents;
        }
        final List<TargetEvent> rateLimitedList = rateLimit(listOfEvents);
        return rateLimitedList;
    },
    (listOfEvents1, listOfEvents2) -> {
        listOfEvents1.addAll(listOfEvents2);
        return listOfEvents1;
    }, partitioner
    );
```

Still slow…

- What else can I do?

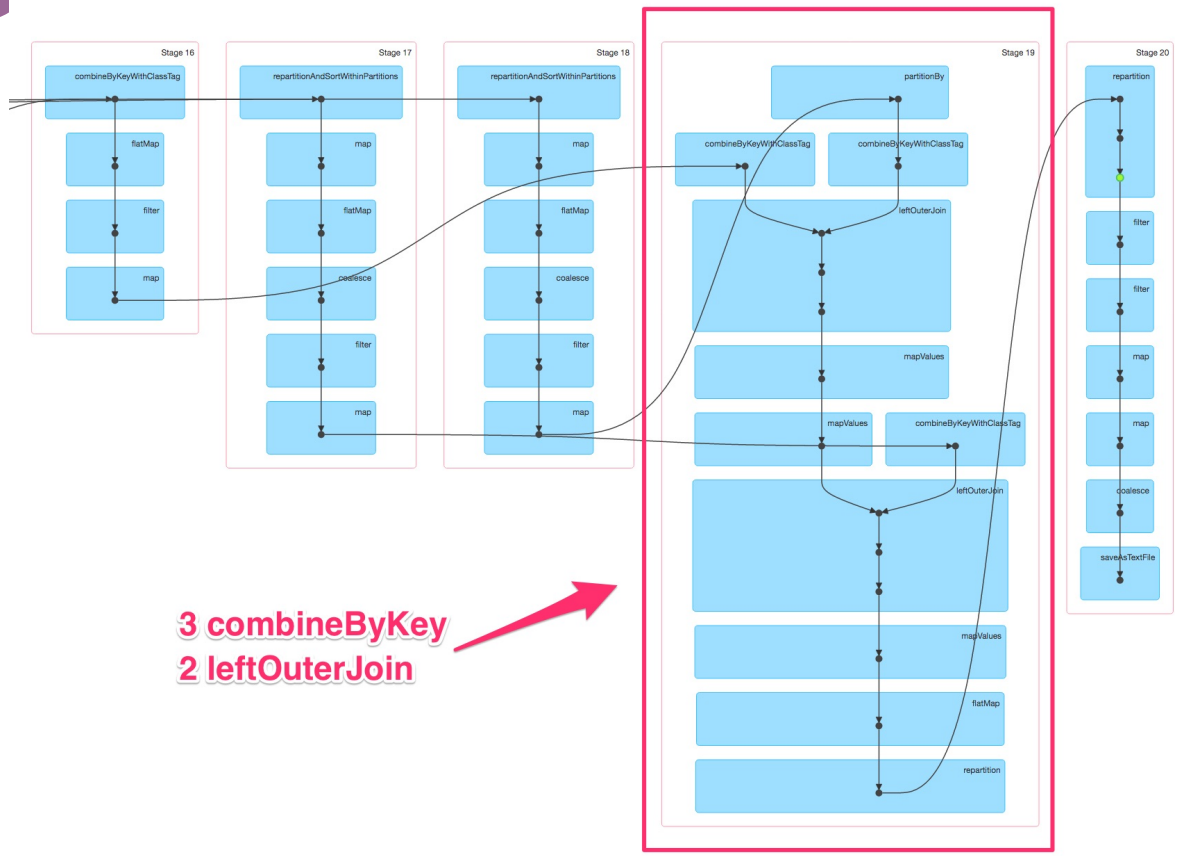
What we will talk about today

- Issues at scale
- Skew
- **Optimizations**
- Ganglia

Avoid shuffles

- Reuse the same partitioner instance

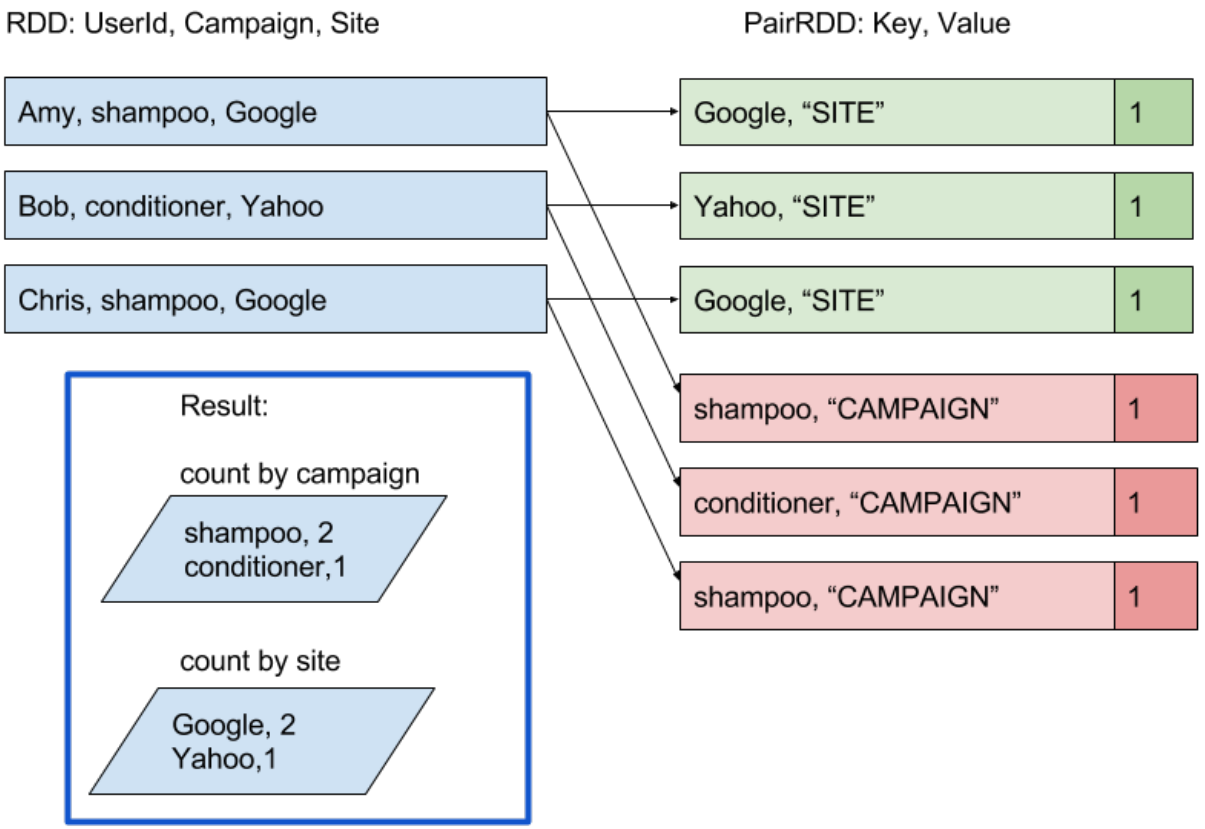
The DAG


Stage 16
combineByKeyWithClassTag
flatMap
filter
map
Stage 17
repartitionAndSortWithinPartitions
map
flatMap
coalesce
filter
map
Stage 18
repartitionAndSortWithinPartitions
map
flatMap
coalesce
filter
map
Stage 19
partitionBy
combineByKeyWithClassTag
combineByKeyWithClassTag
leftOuterJoin
mapValues
mapValues
combineByKeyWithClassTag
leftOuterJoin
mapValues
flatMap
repartition
Stage 20
repartition
filter
filter
map
map
coalesce
saveAsTextFile
3 combineByKey
2 leftOuterJoin

Avoid shuffles

- Denormalize data or union data to minimize shuffle
- Rely on the fact we will reduce into a highly compressed key space.
- For example, we want count of events by campaign, also count of events by site

Avoid shuffle



Coalesce partitions when loading

- Loading many small files – coalesce down # of partitions
- No shuffle
- Reduce task overhead, greatly improve speed
- Going from 300k partitions to 60k, cut time by half

Coalesce partitions when loading

```
final JavaRDD<Event> eventRDD= loadDataFromS3(); // load data
final int loadingPartitions = eventRDD.getNumPartitions(); // inspect
how many partitions
final int coalescePartitions = loadingPartitions / 5; // use algorithm
to calculate new #

eventRDD
  .coalesce(coalescePartitions) // coalesce to smaller #
  .map(e -> transform(e)) // faster subsequent operations
```

Materialize data

- Large chunk of data persisted in memory
- Large RDD used to calculate small RDD
- Use an Action to materialize the smaller calculated result so larger data can be unpersisted

Materialize data

```
parent.cache() // persist large parent PairRDD to memory
child1 = parent.reduceByKey(a).cache() // calculate child1 from parent
child2 = parent.reduceByKey(b).cache() // calculate child2 from parent
child1.count()// perform an Action
child2.count()// perform an Action
parent.unpersist() // safe to mark parent as unpersisted
// rest of the code can use memory
```

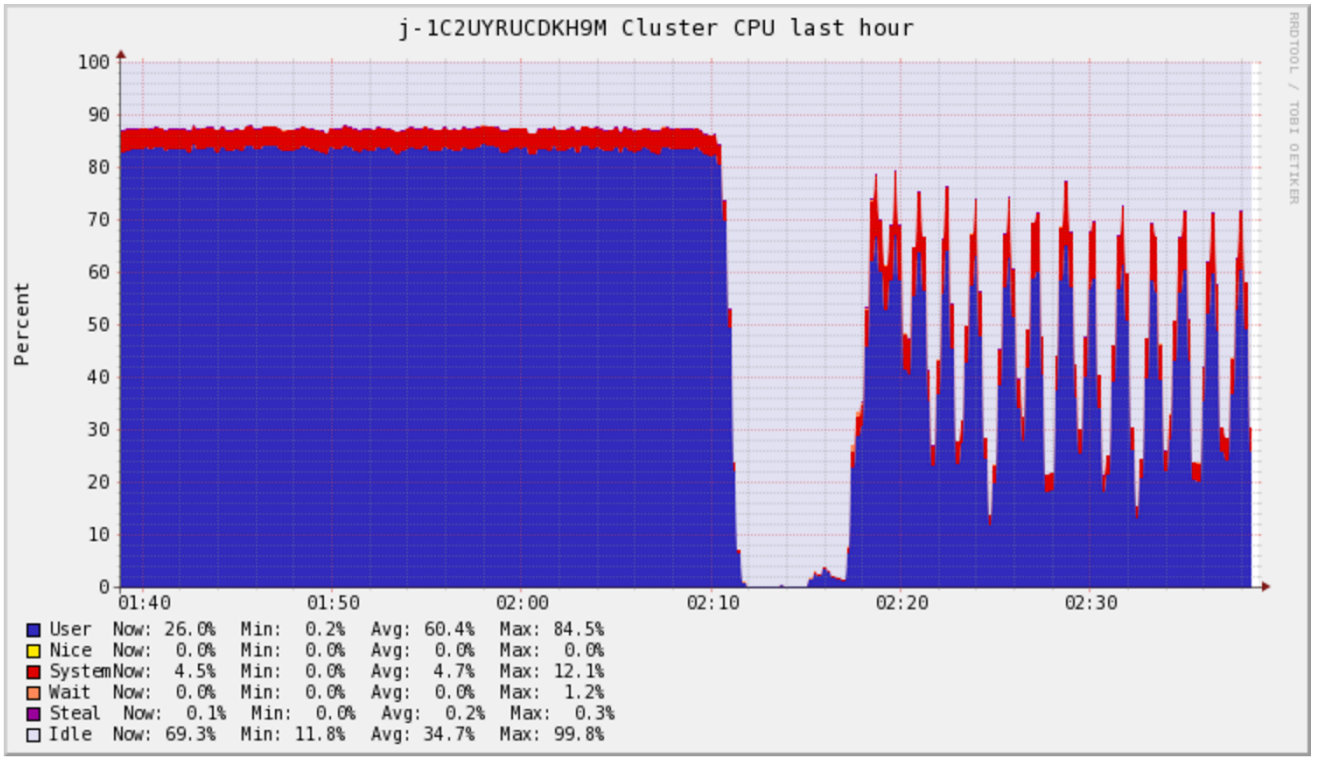
What we will talk about today

- Issues at scale
- Skew
- Optimizations
- **Ganglia**

Ganglia

- Ganglia is an extremely useful tool to understand performance bottlenecks, and to tune for highest cluster utilization

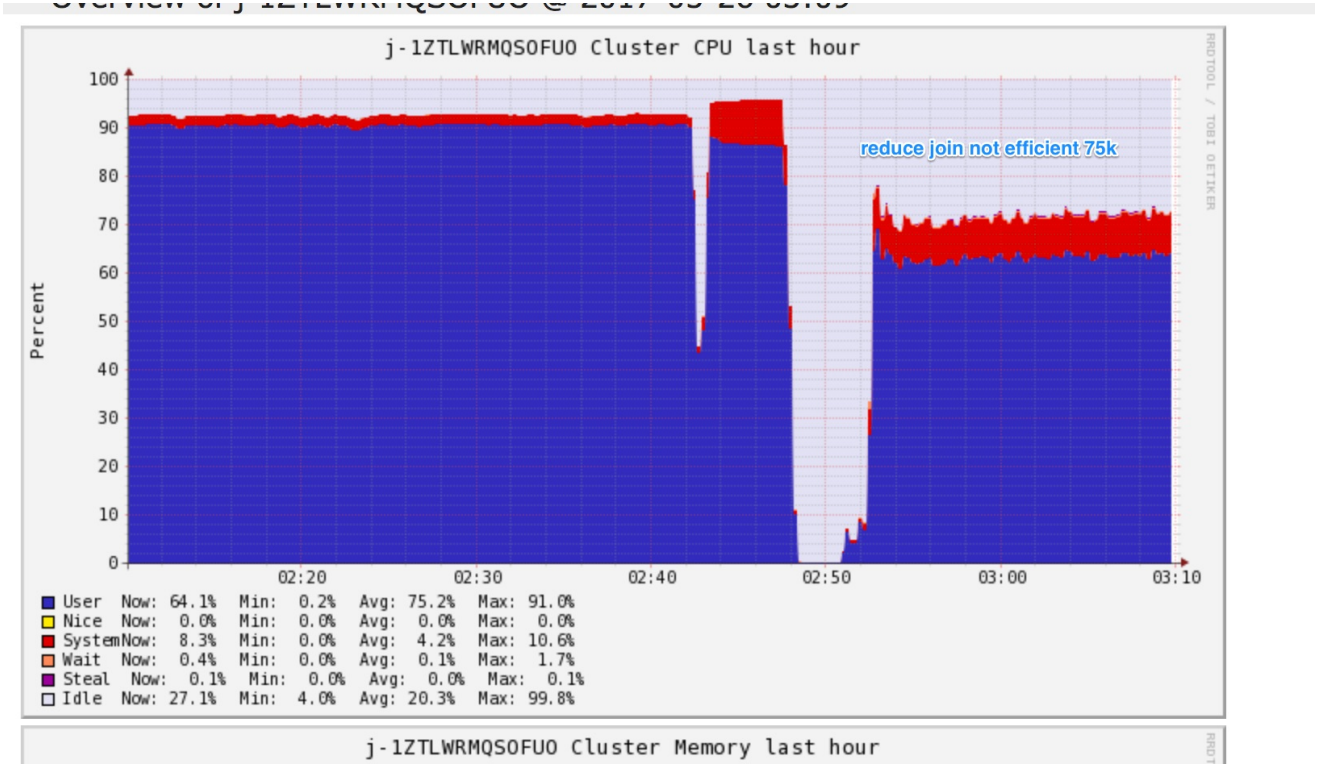
Ganglia: CPU wave



Ganglia: CPU wave

- Executors are going into GC multiple times in the same stage
- Running out of execution memory
- Persist to StorageLevel.DISK_ONLY()

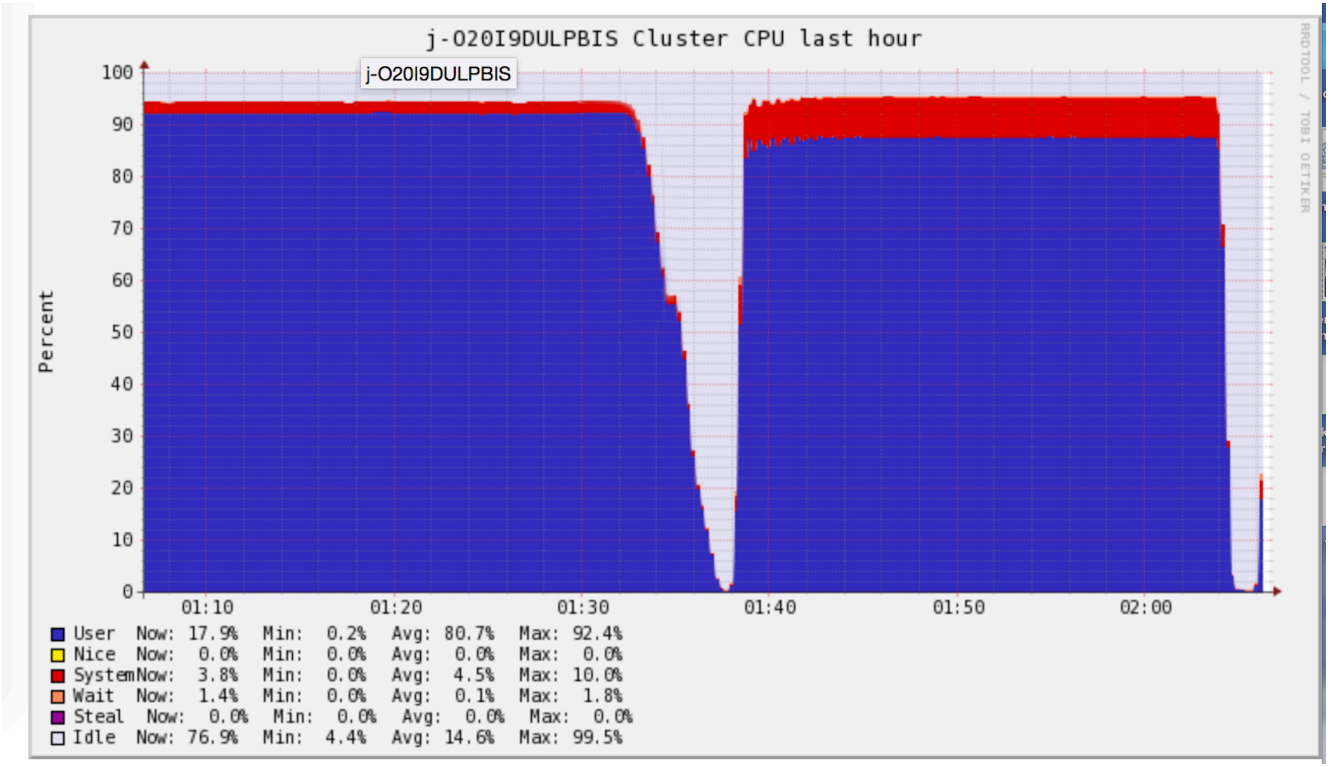
Ganglia: inefficient use



Ganglia: inefficient use

- Decrease # of partitions of RDDs used in this stage

Ganglia: much better



Final Configuration

- Master 1 r3.4xl
- Executors 110 r3.4xl
- Configurations:

spark	maximizeResourceAllocation	TRUE
spark-defaults	spark.executor.cores	16
spark-defaults	spark.dynamicAllocation.enabled	FALSE
spark-defaults	spark.driver.maxResultSize	8g
spark-defaults	spark.rpc.message.maxSize	2047
spark-defaults	spark.rpc.askTimeout	300
spark-defaults	spark.network.timeout	300s
spark-defaults	spark.executor.heartbeatInterval	20s
spark-defaults	spark.executor.memory	92540m
spark-defaults	spark.yarn.executor.memoryOverhead	23300
spark-defaults	spark.task.maxFailures	10
spark-defaults	spark.executor.extraJavaOptions	-XX:+UseG1GC
spark-defaults	spark.cleaner.periodicGC.interval	600min

Summary

- Large jobs are special, use special settings
- Outsmart the skew
- Use Ganglia!

Thank you

Emma Tang

@emmayolotang

@Neustar