How to performance-tune spark applications in large clusters

- Omkar Joshi

marmaray

ANY SOURCE, ANY SINK.

Omkar Joshi

ojoshi@netflix.com

- Software engineer @ Netflix
- Architect & author of Marmaray (Generic ingestion framework) @ Uber.
- Architected Object store & NFS solutions at Hedvig
- Hadoop Yarn committer
- Enjoy gardening & hiking in free time

Agenda

01 JVM Profiler

02 Hadoop Profiler

03 Spark Listener

04 Auto tune

05 Storage improvements

06 Cpu / Runtime improvements

07 Efficiency improvements

08 Memory improvements

JVM Profiler: Distributed Profiling at a Large Scale

- Help tracking memory/cpu/stacktrace for large amount of Spark executors

- (

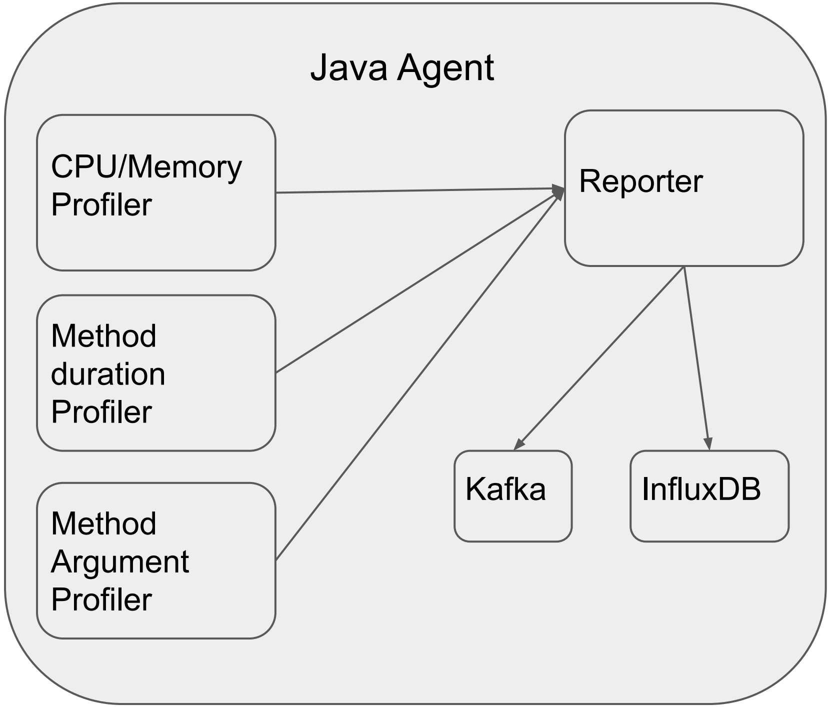
common/jvm-profiler)

- Presented in previous Spark Summit

- Some new update

Hadoop Profiler (Recap)

- **Java Agent attached to each executor**
- **Collects metrics via JMX and /proc**
- **Instruments arbitrary Java user code**
- **Emits to Kafka, InfluxDB, and Redis and other data sinks**



https://github.com/uber-common/jvm-profiler

Spark Listener

- Plugable Listener
  - spark.extraListeners=com.foo.MySparkListener
- Modify Spark Code and Send Execution Plan to Spark Listener
- Generate Data Lineage Information
- Offline Analysis for Spark Task Execution

Auto Tune

- **Problem: data scientist using team level Spark conf template**

- **Known Daily Applications**
  - **Use historical run to set Spark configurations (memory, vcore, etc.) ***

- **Ad-hoc Repeated (Daily) Applications**
  - **Use Machine Learning to predict resource usage**
  - **Challenge: Feature Engineering**
    - **Execution Plan**

```
Project [user_id, product_id, price]
  Filter (date = 2019-01-31 and product_id = xyz)
    UnresolvedRelation datalake.user_purchase
```

marmaray

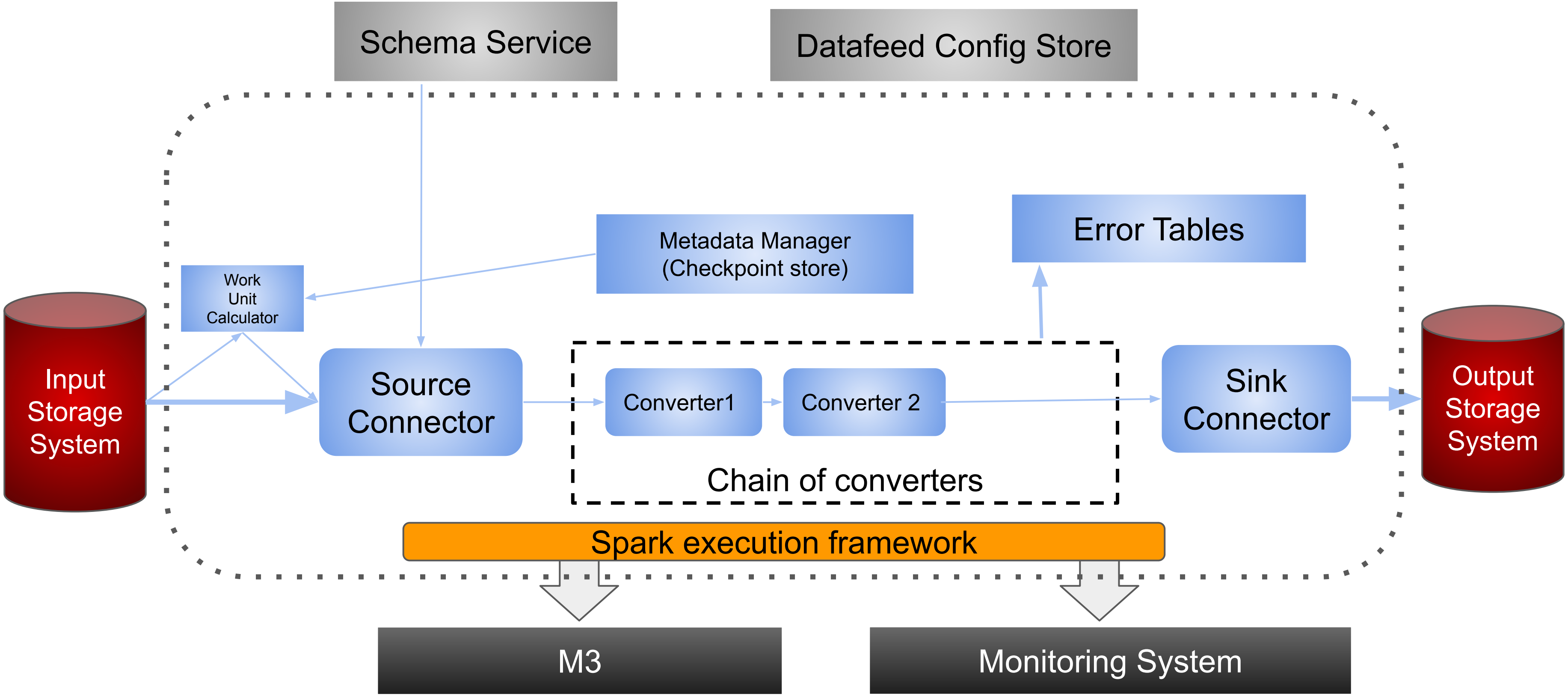
Open Sourced in September 2018

https://github.com/uber/marmaray

Blog Post:

https://eng.uber.com/marmaray-hadoop-ingestion-open-source/

High-Level Architecture



Spark job improvements

Storage improvements

Effective data layout (parquet)

- Parquet uses columnar compression
- Columnar compression savings outperform gz or snappy compression savings
- Align records such that adjacent rows have identical column values
  - Eg. For state column California.
- Sort records with increasing value of cardinality.
- Generalization sometimes is not possible; If it is a framework provide custom sorting.

User Id	First Name	City	State	Rider score
abc123011101	John	New York City	New York	4.95
abc123011102	Michael	San Francisco	California	4.92
abc123011103	Andrea	Seattle	Washington	4.93
abc123011104	Robert	Atlanta City	Georgia	4.95

User Id	First Name	City	State	Rider score
cas123032203	Sheetal	Atlanta City	Georgia	4.97
dsc123022320	Nikki	Atlanta City	Georgia	4.95
ssd012320212	Dhiraj	Atlanta City	Georgia	4.94
abc123011104	Robert	Atlanta City	Georgia	4.95

CPU / Runtime improvements

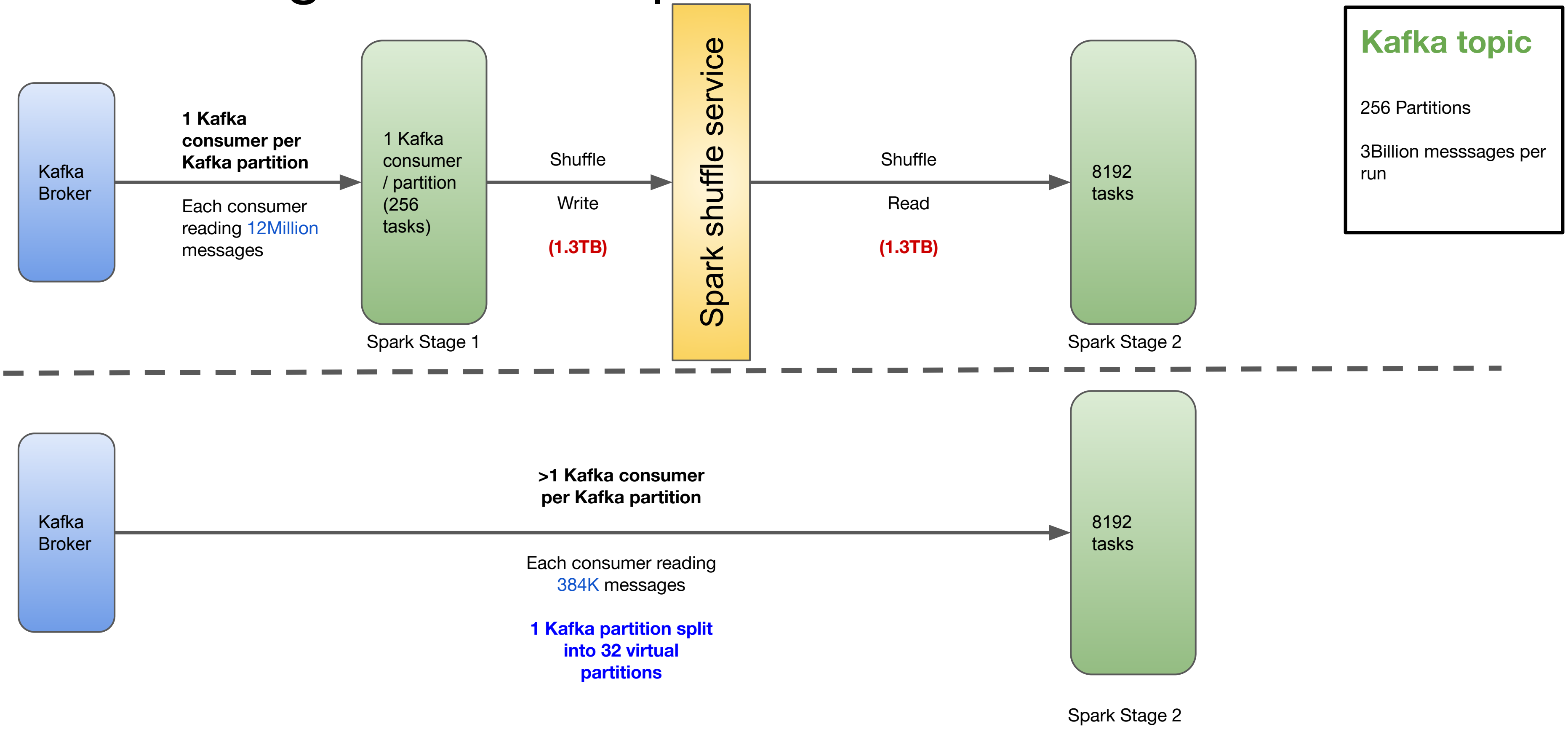
Custom Spark accumulators

- Problem
  - Given a set of ride records; remove duplicate ride records and also count duplicates per state

1. RDD<String> rideRecords = javaSparkContext.readParquet(some_path);
2. Map<String, Long> ridesPerStateBeforeDup = rideRecords.map(r -> getState(r)).countByKey();
3. RDD<String> dedupRideRecords = dedup(rideRecords);
4. Map<String, Long> ridesPerStateAfterDup = dedupRideRecords.map(r -> getState(r)).countByKey();
5. dedupRideRecords.write(some_hdfs_path);
6. **Duplicates = Diff(ridesPerStateAfterDup, ridesPerStateBeforeDup)**
7. **# spark stages = 5 (3 for countByKey)**

1. Class RidesPerStateAccumulator extends AccumulatorV2<Map<String, Long>>
2. RidesPerStateAccumulator riderPerStateBeforeDup, riderPerStateAfterDup;
3. dedup(javaSparkContext.readParquet(some_path).map(r -> {riderPerStateBeforeDup.add(r); return r;})).map(r -> {riderPerStateAfterDup.add(r); return r;}).write(some_hdfs_path);
4. **Duplicates = Diff(ridesPerStateAfterDup, ridesPerStateBeforeDup)**
5. **# spark Stages = 2 (no counting overhead!!)**

Increasing kafka read parallelism



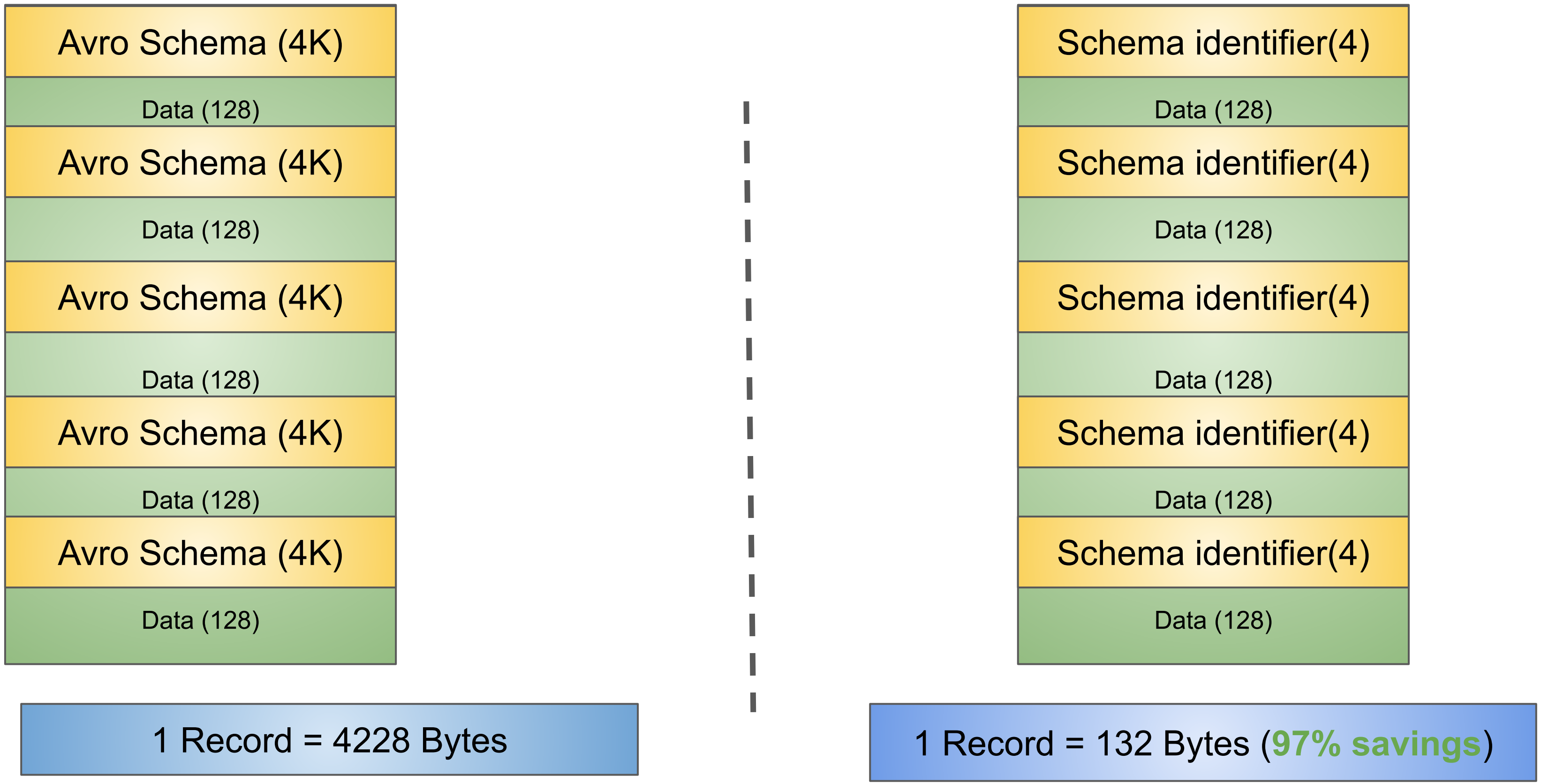
Increasing kafka read parallelism contd..

```java
final RDD<ConsumerRecord<byte[], byte[]>> kafkaRDD = new KafkaRDD<byte[], byte[]>(
    this.jsc.get().sc(),
    kafkaParams,
    workUnits.toArray(new OffsetRange[0]),
    Collections.emptyMap(),
     useConsumerCache: true) {
    // It is overridden to ensure that we don't pin topic+partition consumer to only one executor. This allows
    // us to do parallel reads from kafka brokers.
    @Override
    public Seq<String> getPreferredLocations(final Partition thePart) { return new ArrayBuffer<>(); }

    // We are updating client.id on executor per task to ensure we assign unique ids for it.
    @Override
    public scala.collection.Iterator<ConsumerRecord<byte[], byte[]>> compute(final Partition thePart,
        final TaskContext context) {
        super.kafkaParams().put(KafkaConfiguration.CLIENT_ID, String
            .format(getConf().getClientIDFormat(),
                KafkaClientIDGenerator.getClientId(getConf().getConf())));
        return super.compute(thePart, context);
    }
};
```

Kryo serialization

- Why Kryo?
  - Lesser memory footprint than Java serializer.
  - Faster and supports custom serializer
- Bug fix to truly enable kryo serialization
  - Spark kryo config prefix change.
- What all is needed to take advantage of that
  - Set “spark.serializer” to “org.apache.spark.serializer.KryoSerializer”
  - Registering avro schemas to spark conf (sparkConf.registerAvroSchemas())
    - Useful if you are using Avro GenericRecord (Schema + Data)
  - Register all classes which are needed while doing spark transformation
    - Use “spark.kryo.classesToRegister”
    - Use “spark.kryo.registrationRequired” to find missing classes

Kryo serialization contd..



Reduce ser/deser time by restructuring payload

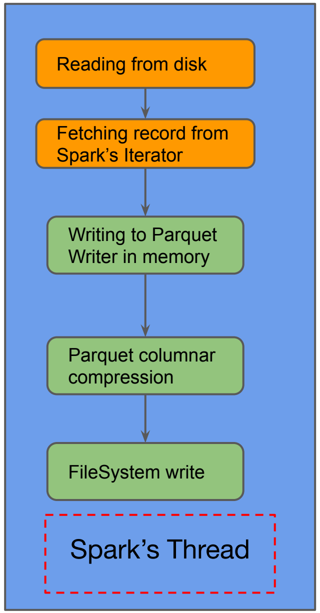
```java
@AllArgsConstructor
@Getter
private class SparkPayload {
  private final String sortingKey;
  // Map with 1000+ entries.
  private final Map<String, GenericRecord> data;
}
```

```java
@Getter
private class SparkPayload {
  private final String sortingKey;
  // Map with 1000+ entries.
  private final byte[] serializedData;

  public SparkPayload(final String sortingKey, Map<String, GenericRecord> data) {
    this.sortingKey = sortingKey;
    this.serializedData = KryoSerializer.serialize(data);
  }

  public Map<String, GenericRecord> getData() {
    return KryoSerializer.deserialize(this.serializedData, Map.class);
  }
}
```

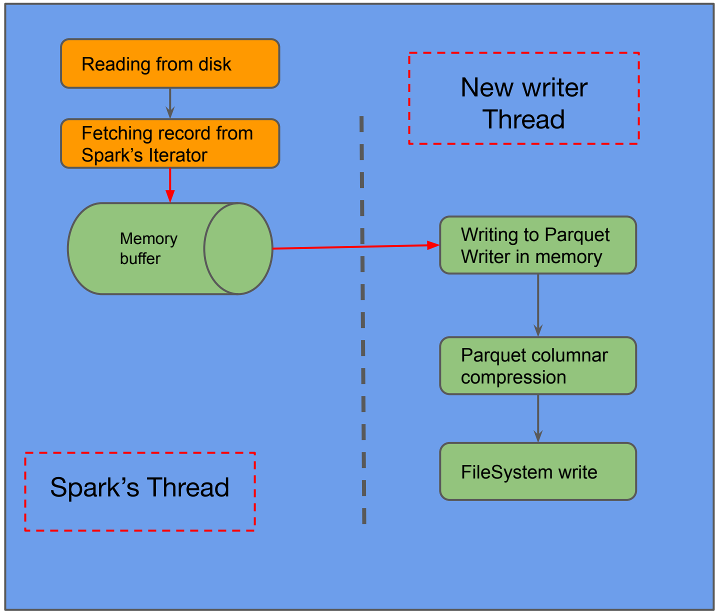
Parallelizing spark's iterator

jsc.mapPartitions (iterator -> while (i.hasnext) { parquetWriter.write(i.next); })

MapPartitions Stage (~45min)



MapPartitions Stage (~25min)



Efficiency improvements

Improve utilization by sharing same spark resources

JavaSparkContext.readFromKafka("topic 1").writeToHdfs();

Threadpool with # threads = parallelism needed

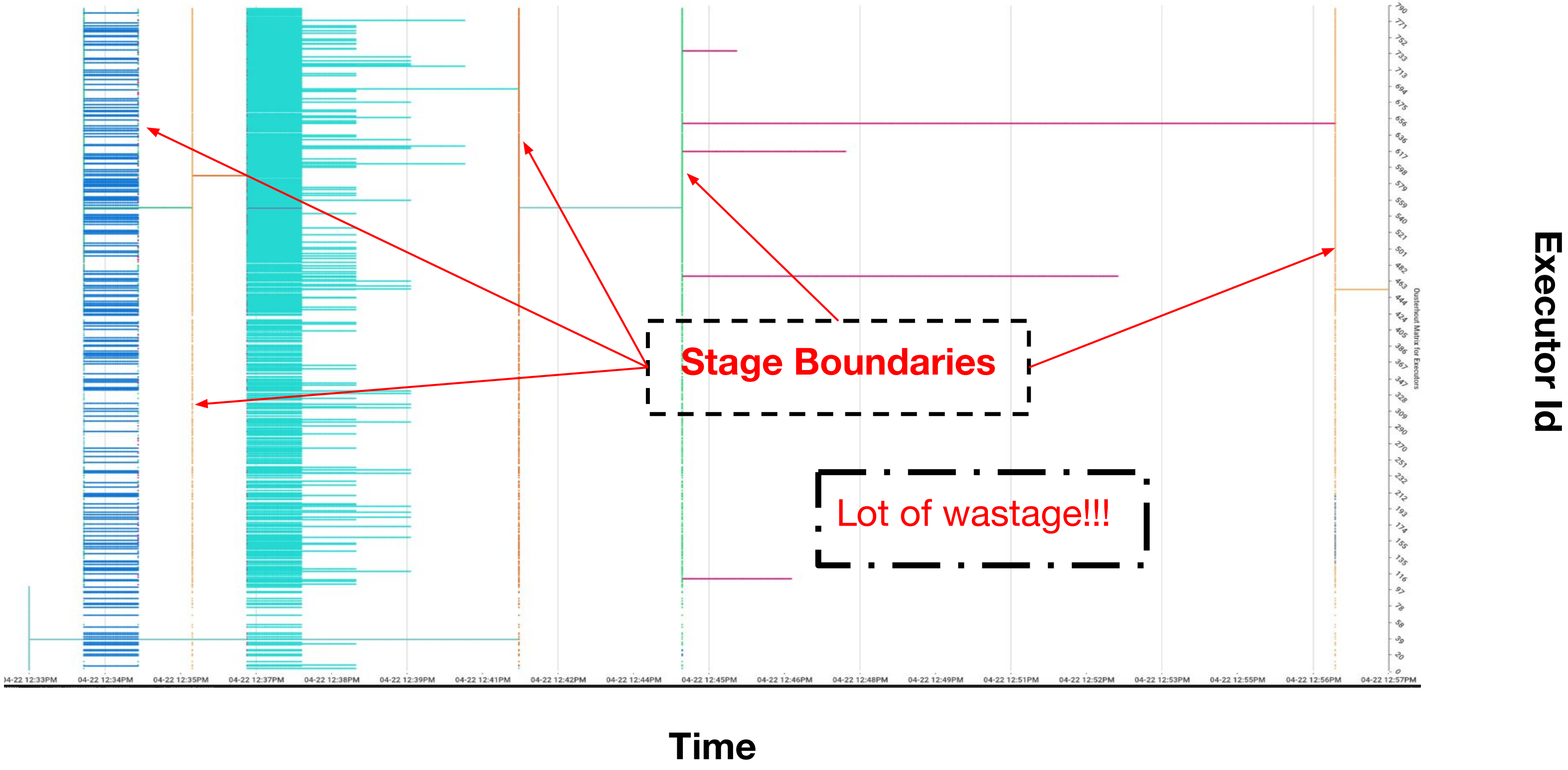
JavaSparkContext.readFromKafka("topic 1").writeToHdfs();

JavaSparkContext.readFromKafka("topic 2").writeToHdfs();

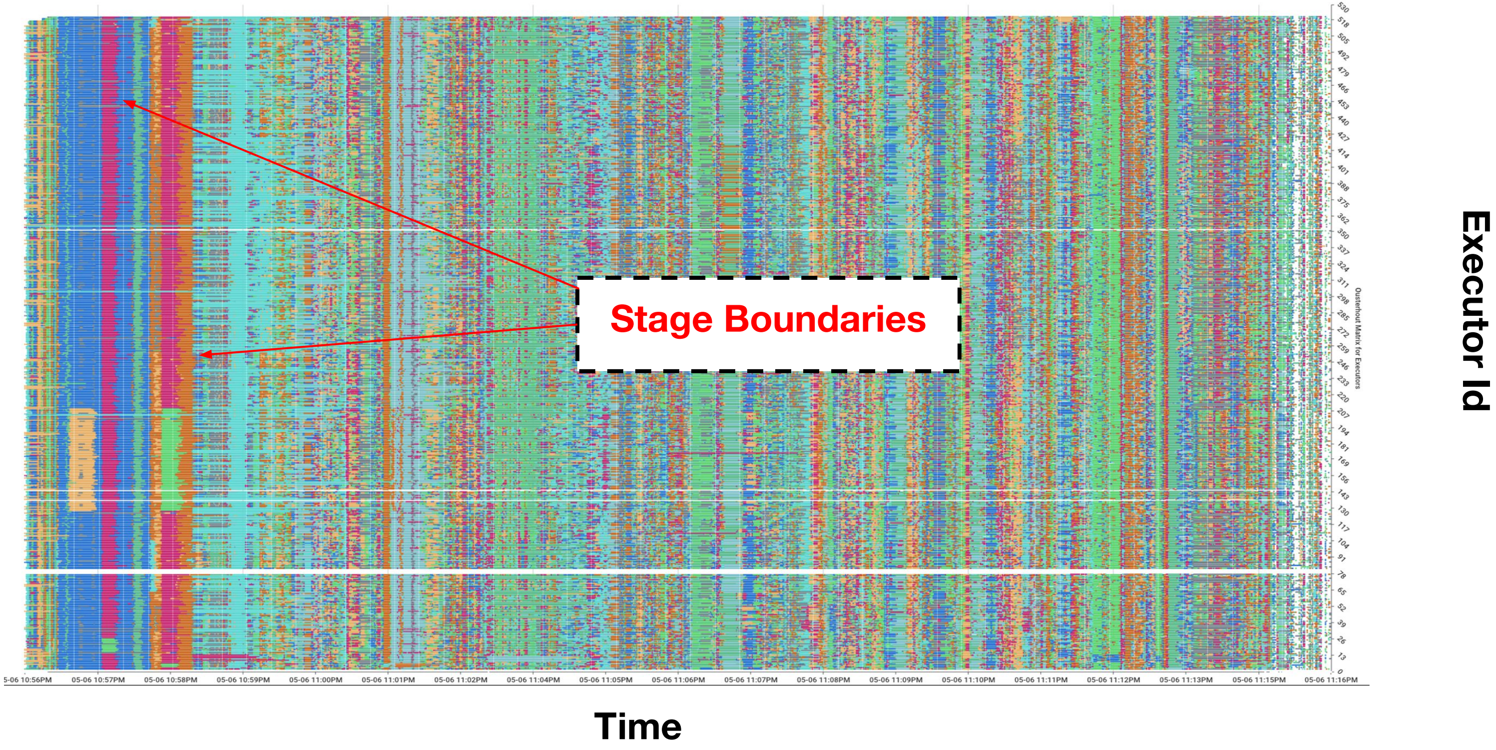
JavaSparkContext.readFromKafka("topic 3").writeToHdfs();

JavaSparkContext.readFromKafka("topic N").writeToHdfs();

Improve utilization by sharing same spark resources



Improve utilization by sharing same spark resources



Memory improvements

Off heap memory improvements (work in progress)

- Symptom
    - Container killed by YARN for exceeding memory limits. 10.4 GB of 10.4 GB physical memory used. Consider boosting spark.yarn.executor.memoryOverhead
- Solution as per stack overflow :)
    - Increase "**spark.yarn.executor.memoryOverhead**"
    - **Result - Huge memory wastage.**
- Spark memory distribution
    - Heap & off-heap memory (direct & memory mapped)
- Possible solutions
    - Avoid memory mapping or perform memory mapping chunk by chunk instead of entire file (4-16MB vs 1GB)
- Current vs Target
    - Current - 7GB [Heap(4gb) + Off-heap(3gb)]
    - Target - 5GB [Heap(4gb) + Off-heap(1gb)] - ~28% memory reduction (per container)

Thank You!!

We are hiring!!

Please reach out to us if you would like to work on the amazing problems.

Omkar Joshi (ojoshi@netflix.com)