DATA+AI SUMMIT 2022

Scaling Salesforce In-Memory Streaming Analytics Platform for Trillion Events Per Day

Dyno Fu, Lead Software Engineer, UIP

Kishore Reddipalli, Sr. Director of Engineering, UIP

ORGANIZED BY databricks

What is Salesforce?

The World's #1 Customer Relationship Management Platform

UIP – Unified Intelligence Platform

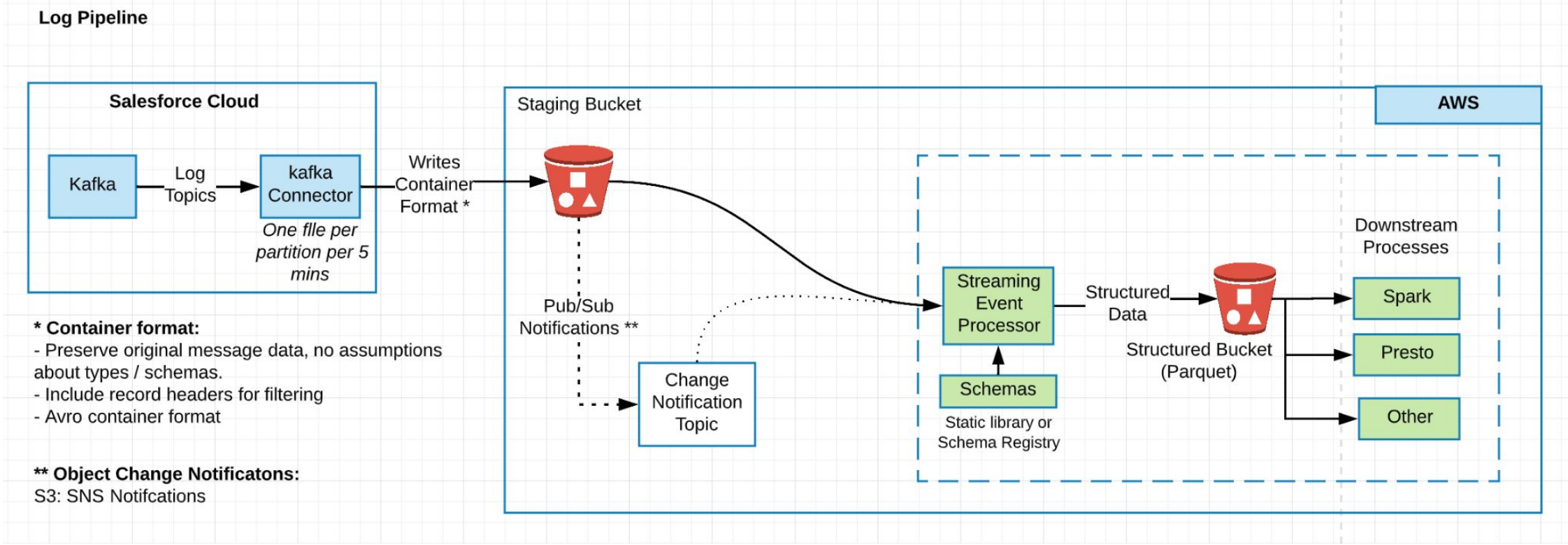
Internal Data Platform for Salesforce

- *UIP is the one-stop data destination for all Salesforce teams to better understand and improve their business areas.*
- *A **modern, trusted, turn-key environment for analytics/ML on big data.***
- *UIP* drives enormous value to Salesforce through it's Trust, Scale, and Network Effects; Reversing/preventing the insecure siloed data use.

UIP – Unified Intelligence Platform

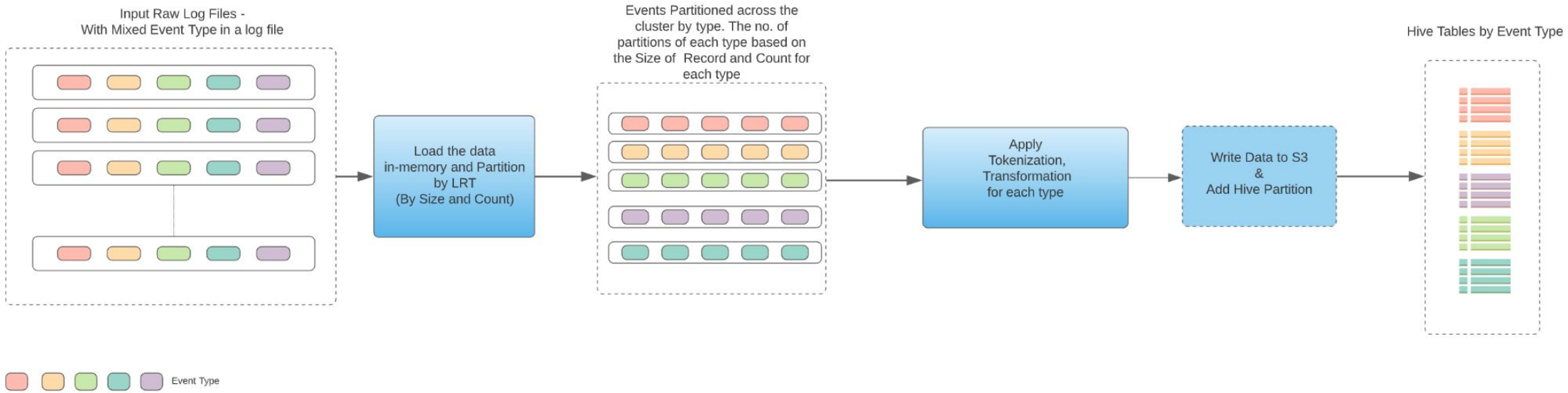
- **100+ PB Data Lake**
- **Ingestion workload**
  - **Volume** - billion events / minute; trillions of events per day
  - **Velocity** - peak 1.2 TB / 5 mins; PB/week
  - **Variety** - around 3k different Event Types

Ingestion Pipeline

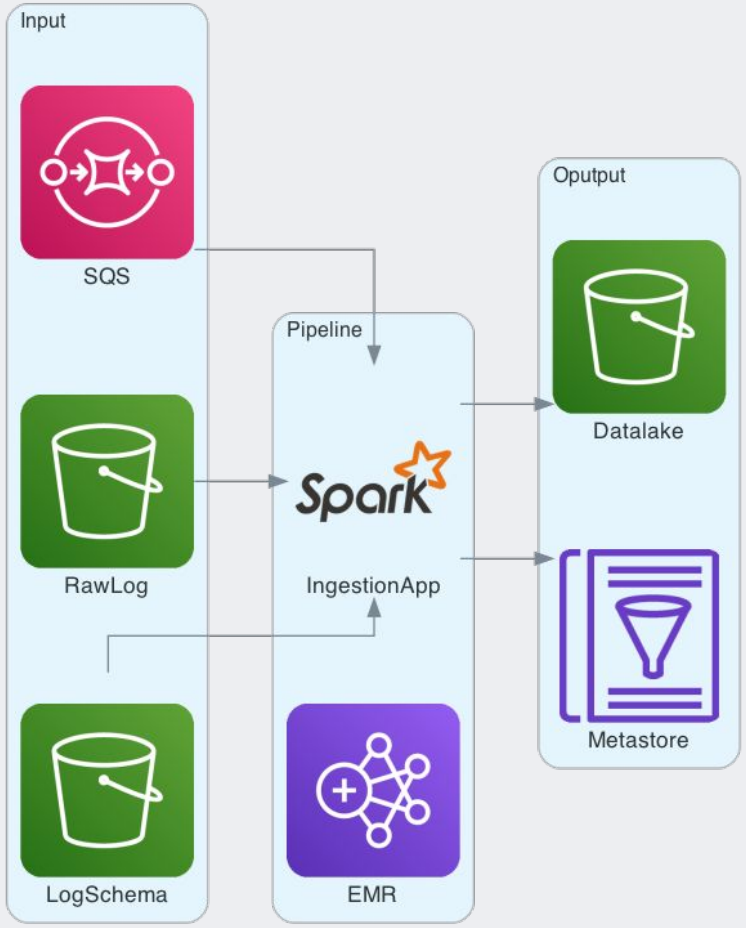


Ingestion Pipeline Data Flow

Read + Count + Sort + Parse + Write



Ingestion Requirements



- Input
  - mixed records of event type
  - Avro with Envelope Schema
- Output
  - Zstd compressed Parquet
  - Partitioned by **batchid, date, hour**
  - One table per event type in Metastore
- 10 minutes target processing time
  - Apply Schema and Exploding Columns
  - Add Partition to Metastore tables

Implementation Challenges

Oversimplified Pipeline

```
val filesInBatch = sqs.receiveMessage(messageNum)
val inputDF = spark.read.avro(filesInBatch: _*)
val eventTypes = input.select("event_type").distinct.collect()
eventTypes.foreach { eventType =>
  val eventDF = inputDF.where($"event_type" === eventType)
  tokenizeAndExplode(eventDF).write.parquet(output)
  metastore.addPartition(eventType, batchId)
}
```

Issues

- Reading data for 40k+ files takes minutes
- Imbalanced data across event types
- Unique schema needs one DF per event type, but filtering on DF is way too expensive
  - Execution plan shows filtered DF has as many partitions as parent DF
  - Too many tasks generated even they are empty

Pipeline Read with Spark DStream

Comparing with Batch Mode

- Faster read - 20~30 seconds v.s. Minutes
  - Why not spark.read.format("avro").load()?
- Amortized Spark Application startup and setup cost

Pipeline Read with Spark Streaming

Spark DStream with SQS/S3 as Source

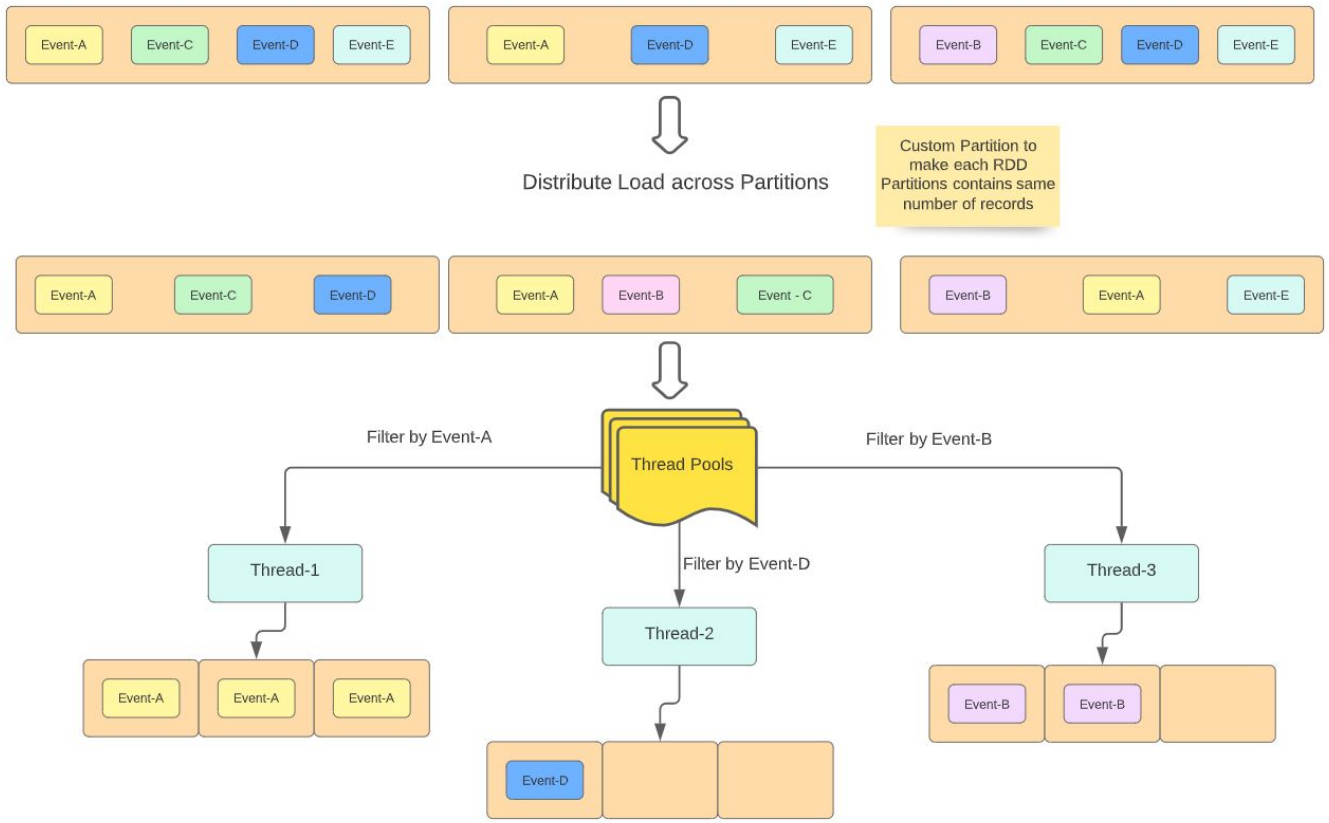
```
class SqsRDD(_sc: SparkContext)(val paths: Seq[String]) extends RDD[(String, Array[Byte])](_sc, Nil) {
  override protected def getPartitions: Array[Partition] =
    paths.zipWithIndex
      .map({ case (path, index) => SqsRDDPartition(index, path) })
      .toArray

  override def compute(split: Partition, context: TaskContext): Iterator[(String, Array[Byte])] = {
    val p = split.asInstanceOf[SqsRDDPartition]
        val uri = new AmazonS3URI(p.path)
        val inputStream = appLogS3Client.getObject(uri.getBucket, uri.getKey).getObjectContent
        val byteArray = IOUtils.toByteArray(inputStream)
        val seekableInput = new SeekableByteArrayInput(byteArray)
        val dataFileReader = new DataFileReader[GenericRecord](seekableInput, new GenericDatumReader[GenericRecord])

        new SqsRDDIterator(dataFileReader)
  }
}
```

Pipeline Event Processing

Partitioning + Tokenization + Parsing + Writing



Pipeline Event Processing

In Memory Partitioning with Range Partitioner

- Range Partitioner & filterByRange
- Construct RDD instead of DataFrame
  - Filtering on DataFrame is very costly but not RDD
  - Writing parquet requires DataFrame API
- RDD with Range Partitioner Optimization
  - Output stats based sizing (average record size for an event type is constant)
  - Range partitioner without sampling
  - Range Mapping Algorithm: partition key(event_type:date:hour) <-> integer salt

Pipeline Event Processing

Schema Management

- Schema for each event types (3k+)
- Schema version for daily patch
- Schema refresh & version fallback without pipeline restart
  - *@transient lazy val …* and Guava *LoadingCache*

Pipeline Event Processing

Spark Scheduling Assistant

- Spread the job submission
  - Easy the pressure on spark scheduler
- Limit the concurrency based on available cores
  - Serialize multiple DataFrame writes for the same event type
- Size the partition based on processing time
  - Identical Machine Scheduling Problem
-

Pipeline Event Processing

Late Arriving Data

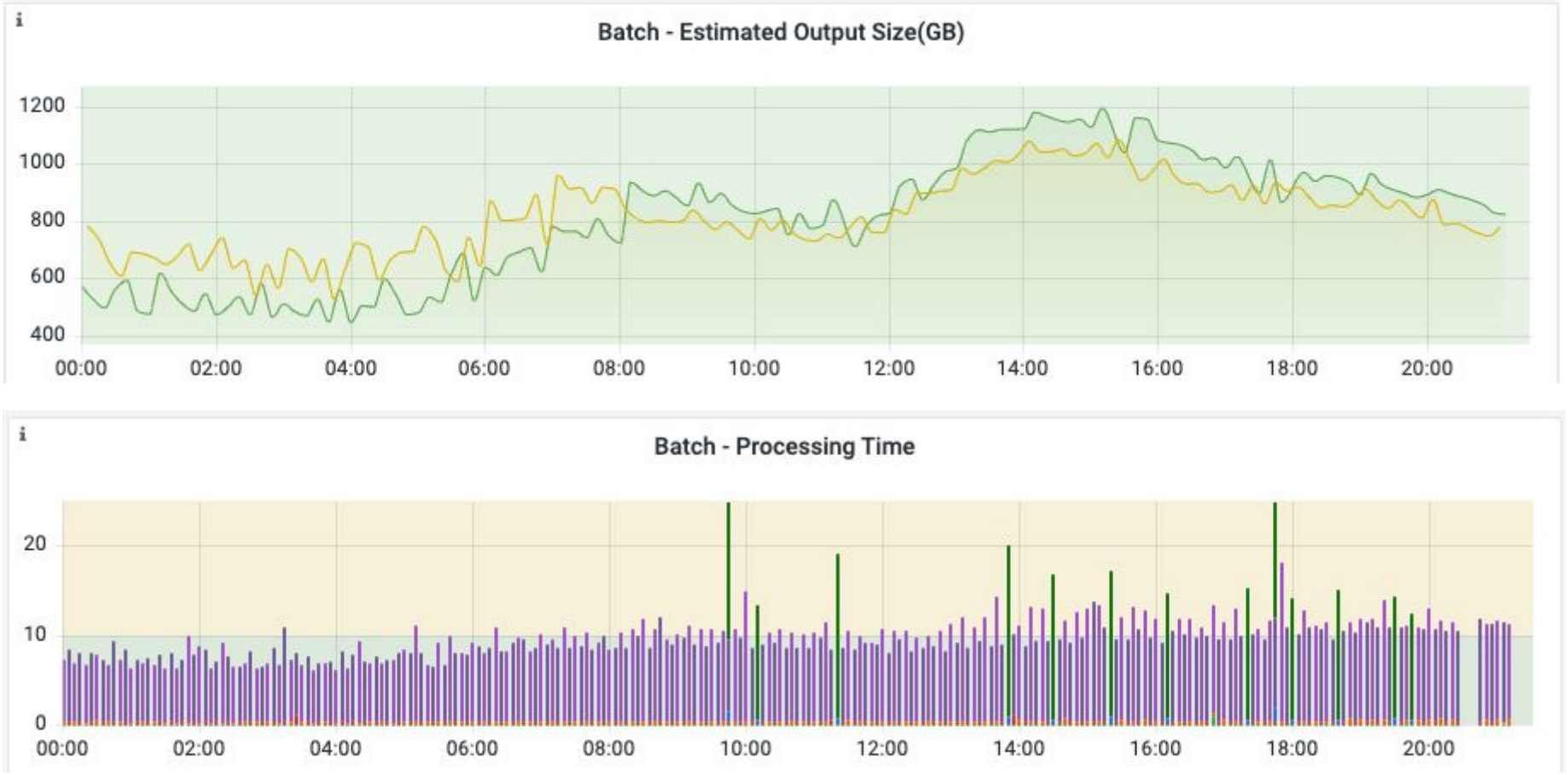
- Late Arriving Data create too many tasks with small partition
- Short period (by hour) - in memory consolidation
  - Output with *partitionBy*
- Long period (by date) - diverge from the source
  - Buffer and consolidate

Pipeline Event Processing

Optimization towards the cloud

- S3 Performance Optimization
  - EMR S3 committer - Don't set partitionOverwriteMode = "dynamic"
  - Prefix randomization with salted batchid batchid=xxx/date=yyy/hour=zzz
  - Avoid prefix **listing** with cache
- EMR Cost optimization
  - Use Spot Instance
  - Adjust resource allocation based on time with EMR scaling policy

Pipeline Operation Dashboard



DATA+AI SUMMIT 2022

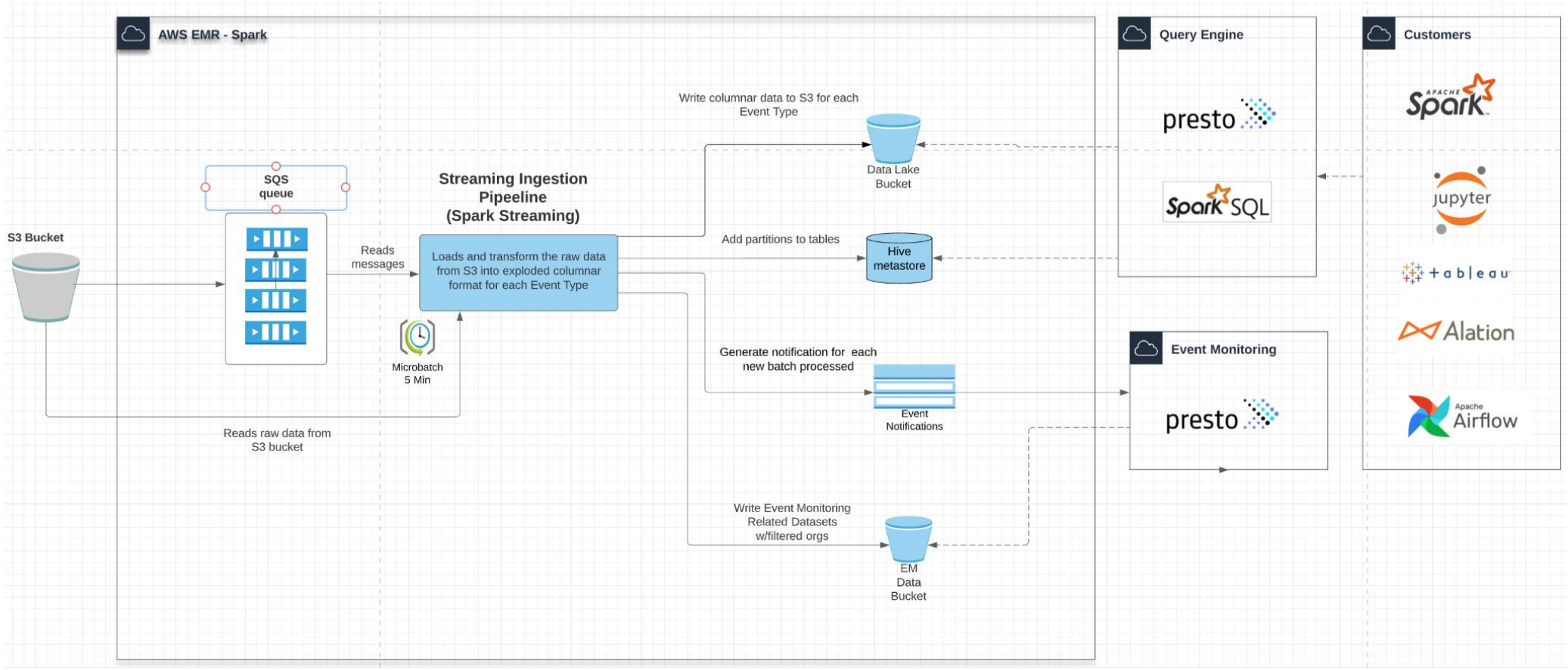
Thank you

Dyno Fu, Lead Software Engineer, UIP

Kishore Reddipalli, Sr. Director of Engineering, UIP

Backup Slides

Ingestion Pipeline Architecture



Pipeline Read – Spark Streaming – SQS/S3

```
// Get the DStream
val stream = new SqsInputDStream(ssc, platformContext, batchSize, numBatches)

//Iterate ove each RDD of the Stream
stream.foreachRDD { rdd =>
    val batchRequests = rdd.asInstanceOf[HasBatchRequest].batchRequests
    // Process RDD

    // RDD is processed. Ack SQS
    stream.asInstanceOf[CanAckMessages].ackMessages(batchRequests)
  }
}
```

Pipeline Read with Spark Streaming

Spark DStream with SQS/S3 as Source

```
private class SqsRDD(sc: SparkContext,
                             val platformContext: PlatformContext,
                             val batchRequests: Array[BatchRequest]
                            ) extends RDD[(String, Array[Byte])](sc, Nil)
  with Logging with HasBatchRequest {

  override def compute(thePart: Partition, context: TaskContext): Iterator[(String, Array[Byte])] = {
    val part = thePart.asInstanceOf[SqsRDDPartition]
    val inputStream: InputStream = platformContext.getBucket.readFile(part.bucketName, part.fileName)
    val byteArray = IOUtils.toByteArray(inputStream)
    val seekableInput = new SeekableByteArrayInput(byteArray)
    val dataFileReader = new DataFileReader[GenericRecord](seekableInput,
      new GenericDatumReader[GenericRecord])
    new SqsRDDIterator(dataFileReader)
  }

  override protected def getPartitions: Array[Partition] = {
    batchRequests.zipWithIndex.map { case (o, i) =>
      new SqsRDDPartition(i, o.bucketName, o.fileName)
    }.toArray
  }
}

private class SqsRDDIterator(dataFileReader: DataFileReader[GenericRecord])
  extends Iterator[(String, Array[Byte])] {
```

```
// Get the DStream
val stream = new SqsInputDStream(ssc, platformContext, batchSize, numBatches)

//Iterate ove each RDD of the Stream
stream.foreachRDD { rdd =>
    val batchRequests = rdd.asInstanceOf[HasBatchRequest].batchRequests
    // Process RDD

    // RDD is processed. Ack SQS
    stream.asInstanceOf[CanAckMessages].ackMessages(batchRequests)
  }
}
```

Pipeline Read – Spark Streaming – SQS/S3

```scala
private class SqsRDDPartition(val index: Int,
                              val bucketName: String,
                              val fileName: String) extends Partition {

  override def toString(): String = {
    s"SqsRDDPartition(index: '$index', bucketName: $bucketName, fileName: '$fileName')"
  }
}
```

```scala
trait HasBatchRequest {
  def batchRequests: Array[BatchRequest]
}

trait CanAckMessages {
  def ackMessages(batchRequests: Array[BatchRequest]): Unit
}

final class BatchRequest private(val bucketName: String,
                                 val fileName: String,
                                 val id: String,
                                 val receiptHandle: String) extends Serializable {
  import BatchRequest.BatchRequestTuple
```

```scala
private[ingestion] class SqsInputDStream(_ssc: StreamingContext,
                                         platformContext: PlatformContext,
                                         batchSize: Integer,
                                         numBatches: Integer)
  extends InputDStream[(String, Array[Byte])](_ssc) with Logging with CanAckMessages {

  override def ackMessages(batchRequests: Array[BatchRequest]): Unit = {
    // ACK every messages in the Batch
  }

  override def compute(validTime: Time): Option[RDD[(String, Array[Byte])]] = {
    // Query SQS, receive N messages
    // Construct BatchRequest for each message. [ Note each message is one file in S3]
    // Construct RDD with Array of BatchRequest. N messages will be N partitions in RDD
    // Construct the Pair RDD
    val pubSub: PubSub = platformContext.getPubSub()
    val batchRequests = new ListBuffer[BatchRequest]()
    pubSub.pullMessages(batchSize).asScala.toList.map { rm =>
      batchRequests += BatchRequest(rm.getBucket, rm.getObject, rm.getId, rm.getReceiptHandle)
    }
    Some(new SqsRDD(context.sparkContext, platformContext, batchRequests.toArray))
  }
```