Near Real Time Indexing Kafka Message to Apache Blur using Spark Streaming

by Dibyendu Bhattacharya

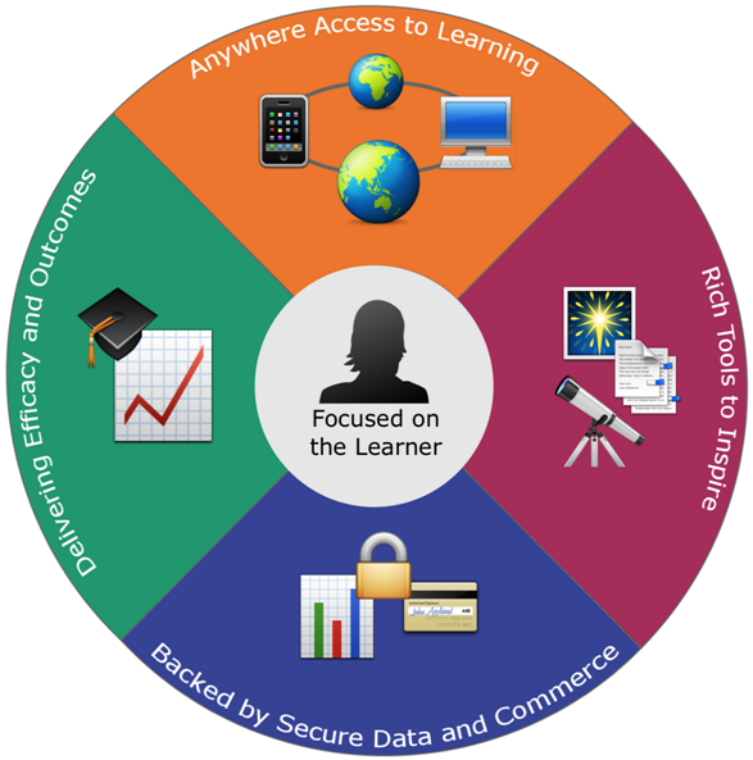
Pearson : What We Do ?

We are building a scalable, reliable cloud-based learning platform providing services to power the next generation of products for Higher Education.

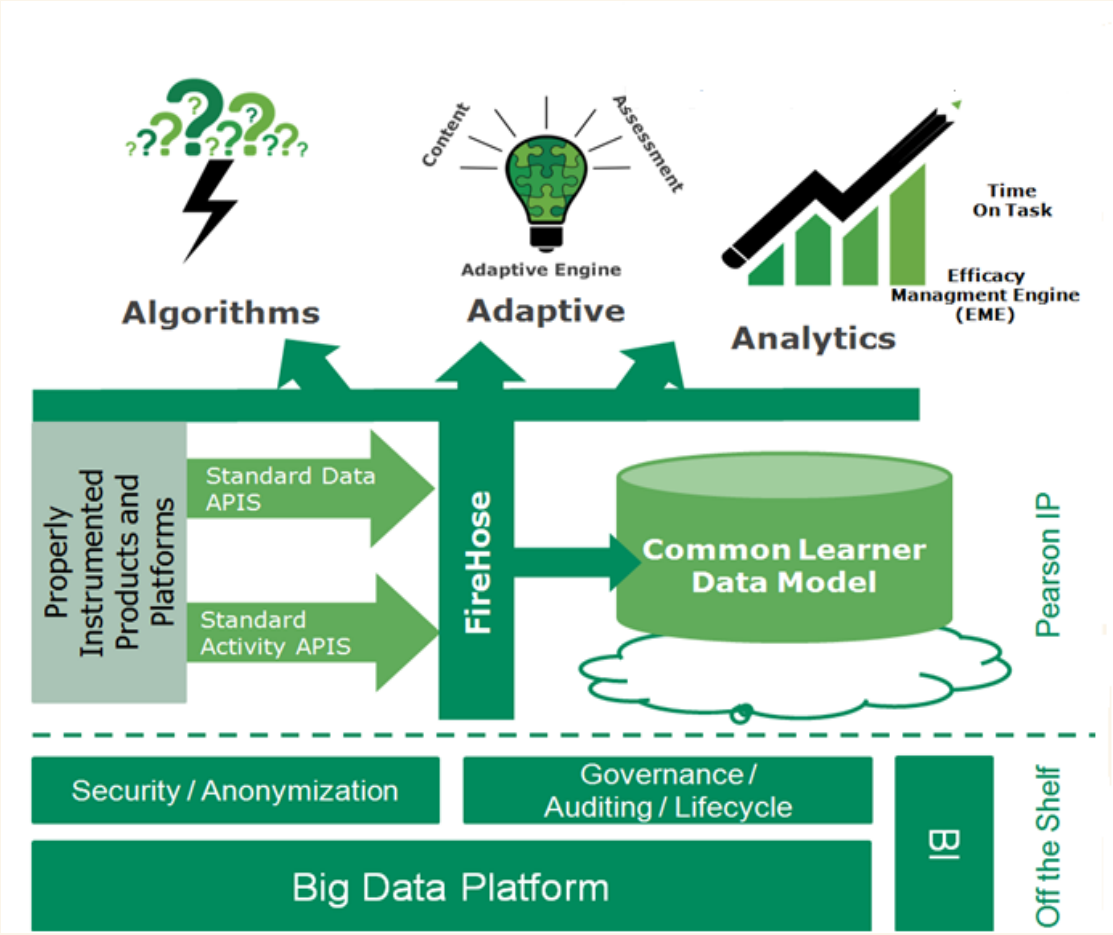
With a common data platform, we build up student analytics across product and institution boundaries that deliver efficacy insights to learners and institutions not possible before.

Pearson is building

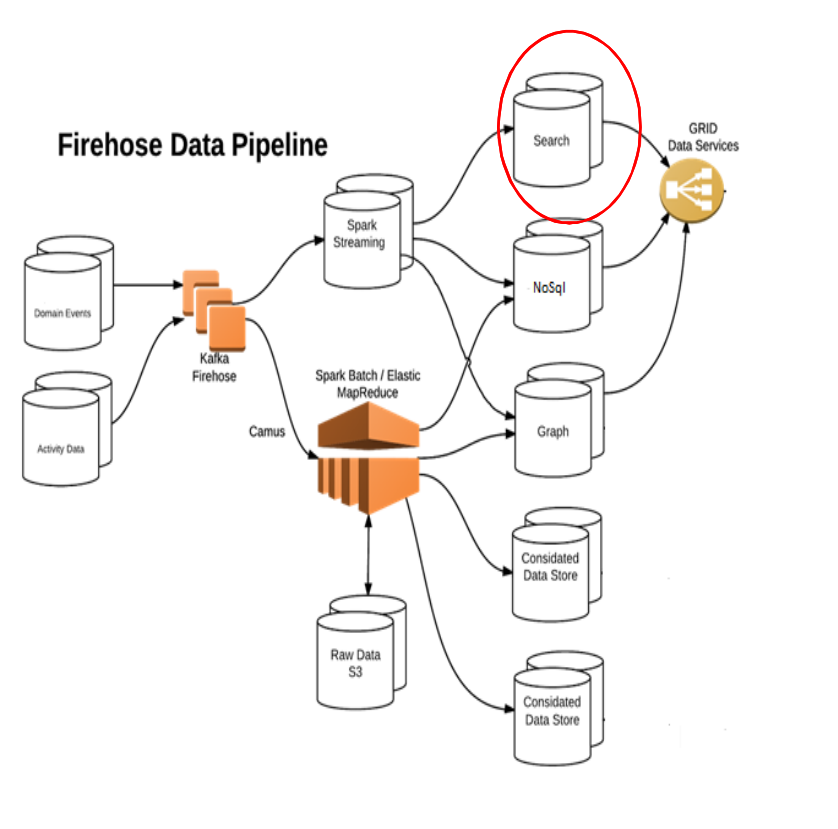
- The worlds greatest collection of educational content
- The worlds most advanced data, analytics, adaptive, and personalization capabilities for education



Pearson Learning Platform : GRID

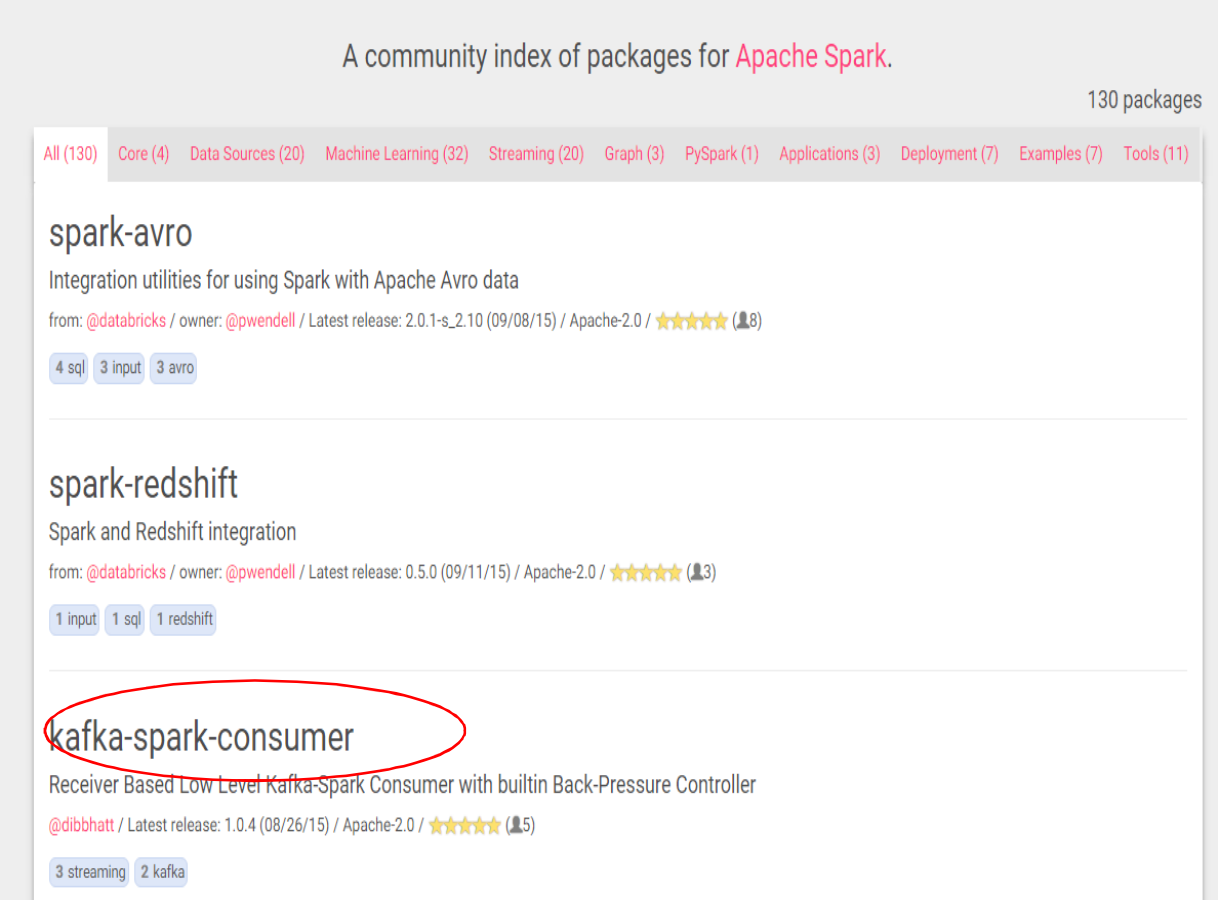


Data , Adaptive and Analytics

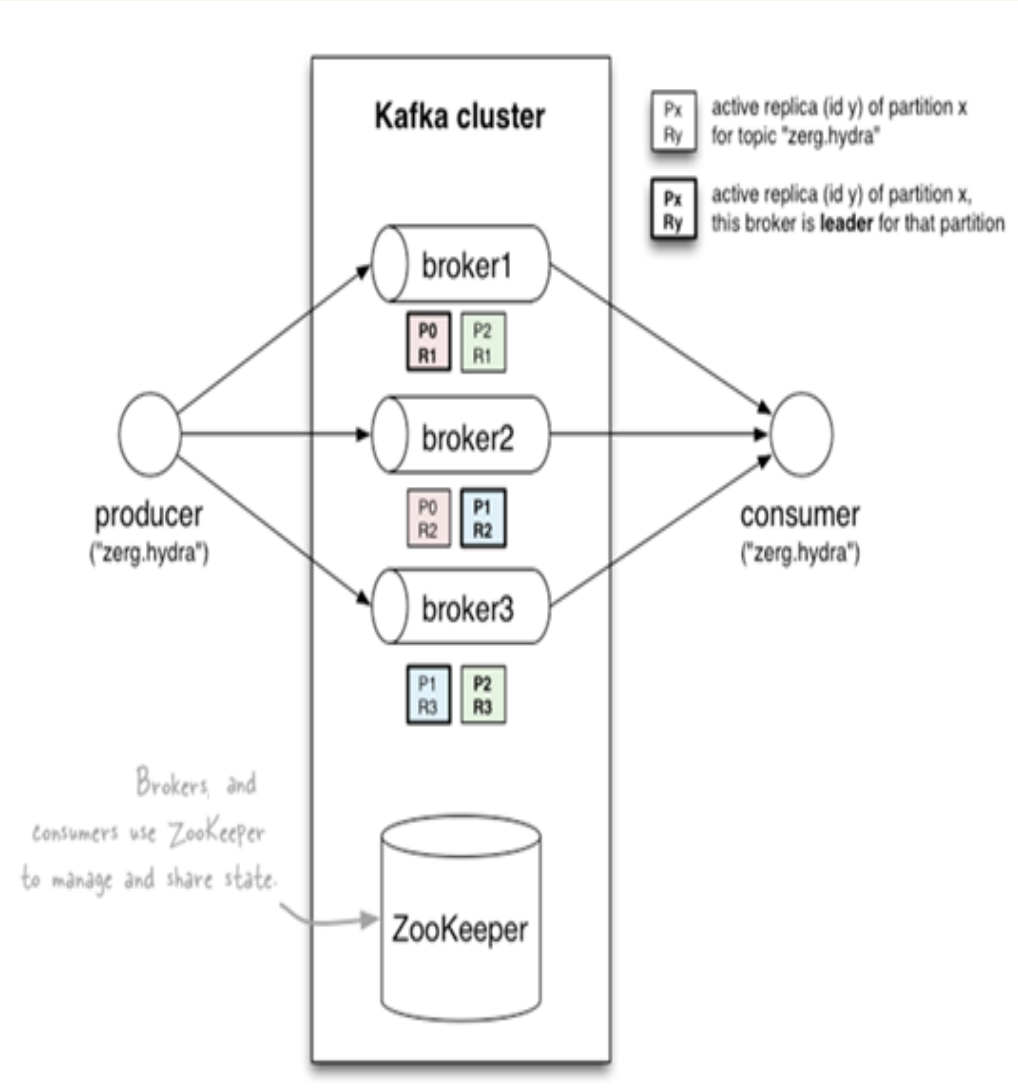


Kafka Consumer for Spark Streaming

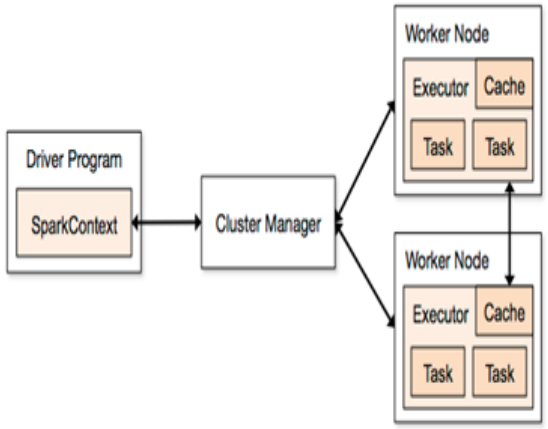
Implemented fault tolerant reliable Kafka consumer which is now part of spark packages (spark-packages.org)



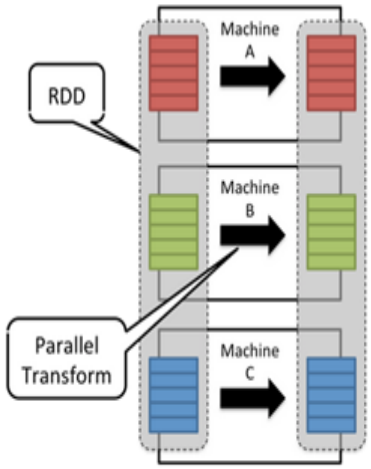
Anatomy of Kafka Cluster..



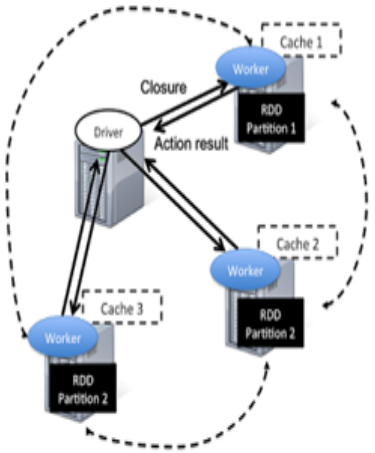
Spark in a Slide..



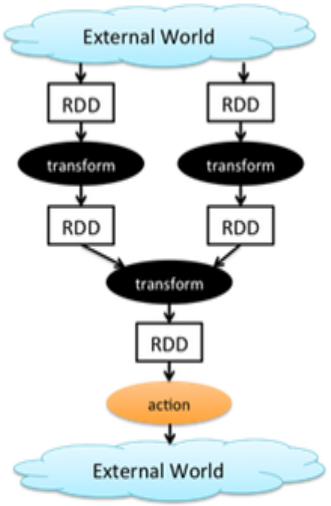
driver program that launches various parallel operations on a cluster . Driver programs access Spark through a SparkContext object which helps to create RDD.



RDD (Resilient Distributed Dataset), which is a logically centralized entity but physically partitioned across multiple machines inside a cluster based on some notion of key

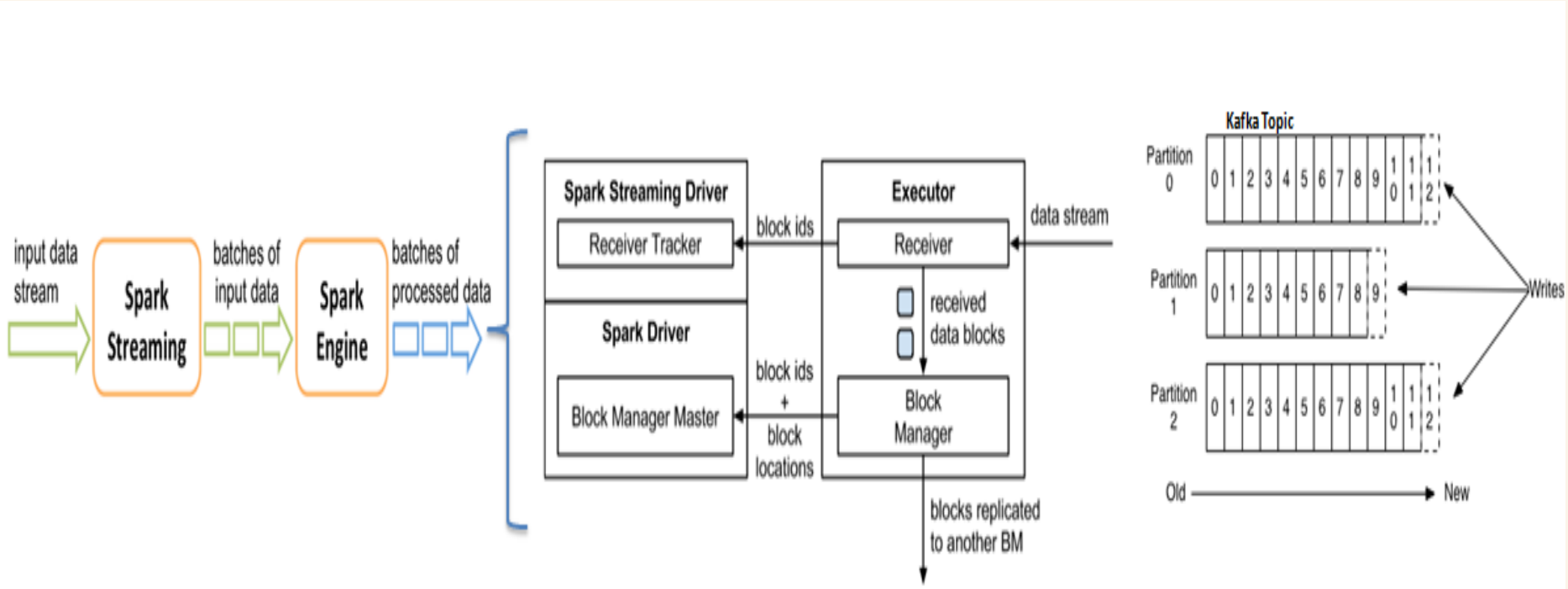


RDD can optionally be cached in memory and hence providing fast access. RDD can also be checkpointed to Disk .

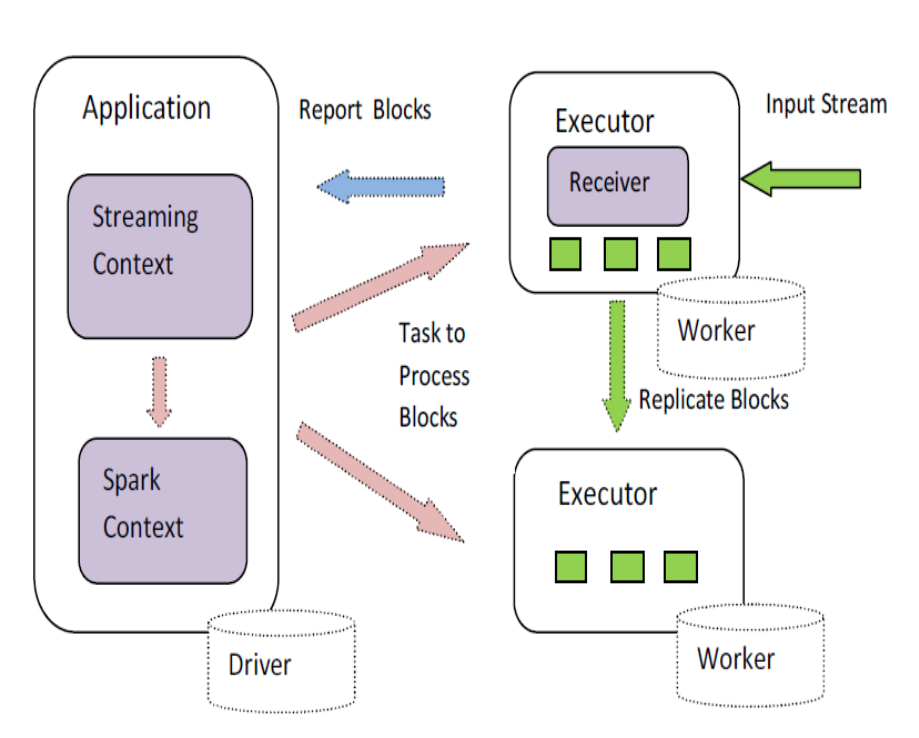


application logic are expressed in terms of a sequence of Transformation and Action. "Transformation" specifies the processing dependency among RDDs and "Action" specifies what the output will be

Spark + Kafka

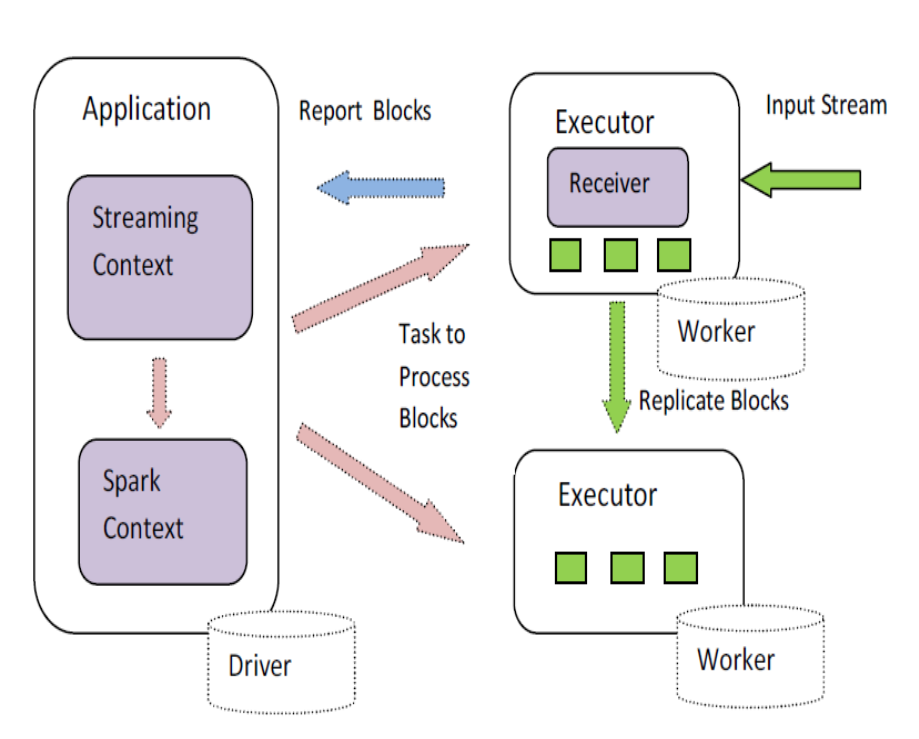

input data stream
Spark Streaming
batches of input data
Spark Engine
batches of processed data
Spark Streaming Driver
Receiver Tracker
Spark Driver
Block Manager Master
block ids
block ids + block locations
Executor
Receiver
data stream
received data blocks
Block Manager
blocks replicated to another BM
Kafka Topic
Partition 0
Partition 1
Partition 2
Writes
Old
New

Spark + Kafka



1. Streaming application uses Streaming Context which uses Spark Context to launch Jobs across the cluster.
2. Receivers running on Executors process
3. Receiver divides the streams into Blocks and writes those Blocks to Spark BlockManager.
4. Spark BlockManager replicates Blocks
5. Receiver reports the received blocks to Streaming Context.
6. Streaming Context periodically (every Batch Intervals) take all the blocks to create RDD and launch jobs using Spark context on those RDDs.
7. Spark will process the RDD by running tasks on the blocks.
8. This process repeats for every batch intervals.

Failure Scenarios..



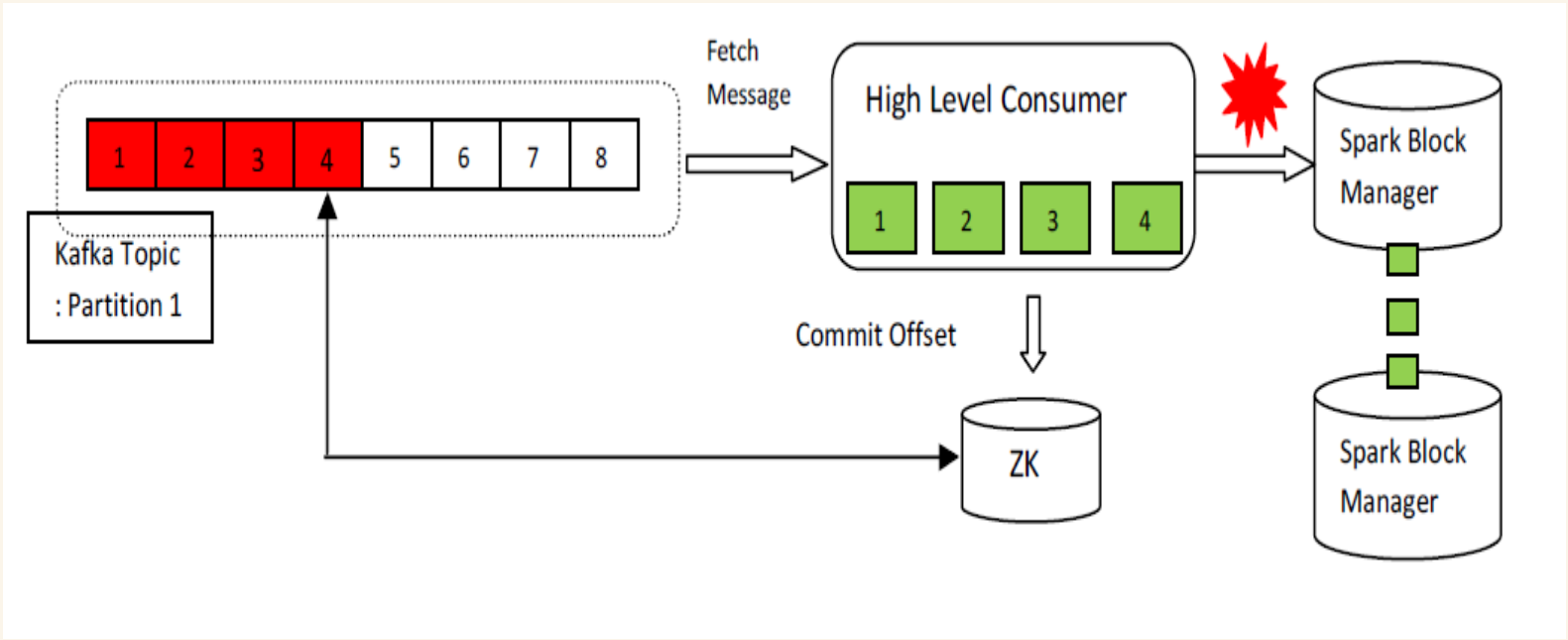
Receiver failed

Driver failed

Data Loss in both cases

Failure Scenarios..Receiver

Un-Reliable Receiver



✓ Need a Reliable Receiver

Kafka Receivers..

Reliable Receiver can use ..

✓ Kafka High Level API (Spark Out of the box Receiver)

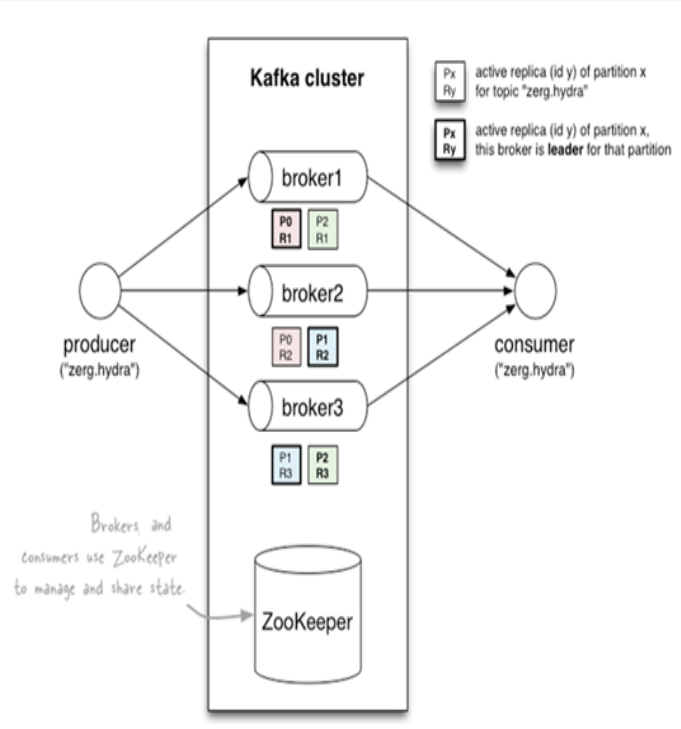
✓ Kafka Low Level API (part of Spark-Packages)

http://spark-packages.org/package/dibbhatt/kafka-spark-consumer

❖ High Level Kafka API has SERIOUS issue with Consumer Re-Balance...Can not be used in Production

https://cwiki.apache.org/confluence/display/KAFKA/Consumer+Client+Re-Design

Low Level Kafka Receiver Challenges

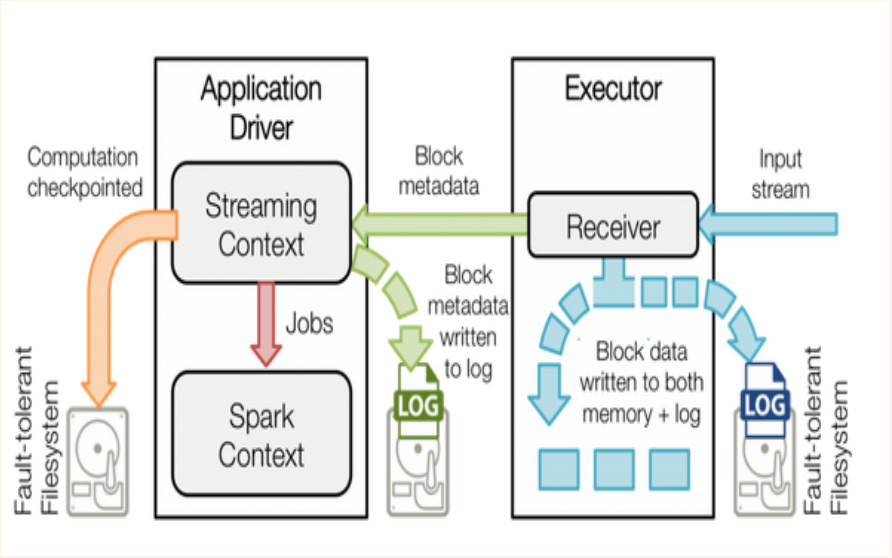


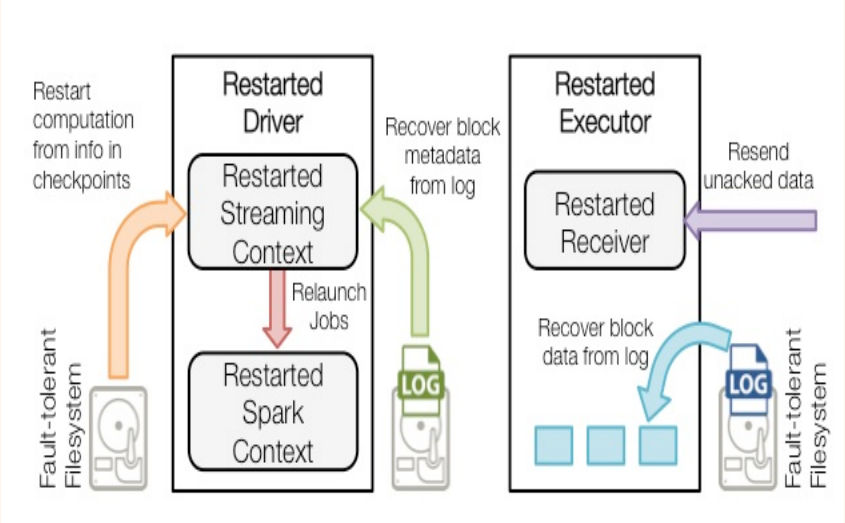
Consumer implemented as Custom Spark Receiver where..

- Consumer need to know Leader of a Partition.
- Consumer should aware of leader changes.
- Consumer should handle ZK timeout.
- Consumer need to manage Kafka Offset.
- Consumer need to handle various failovers.

Failure Scenarios..Driver

- Data which is buffered but not processed are lost ...why ?



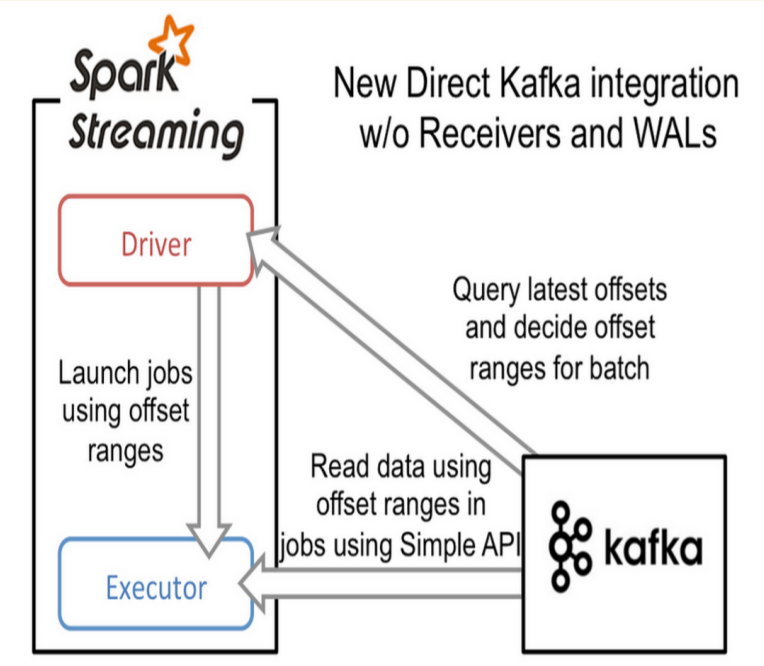


- ✓ When Driver crashed , it lost all its Executors ..and hence Data in BlockManager
- ✓ Need to enable WAL based recovery for both Data and Metadata
- ✓ Can we use Tachyon ?

Some pointers on this Kafka Consumer

- Inbuilt PID Controller to control Memory Back Pressure.
- Rate limiting by size of Block, not by number of messages . Why its important ?
- Can save ZK offset to different Zookeeper node than the one manage the Kafka cluster.
- Can handle ALL failure recovery .
  - Kafka broker down.
  - Zookeeper down.
  - Underlying Spark Block Manager failure.
  - Offset Out Of Range issues.
  - Ability to Restart or Retry based on failure scenarios.

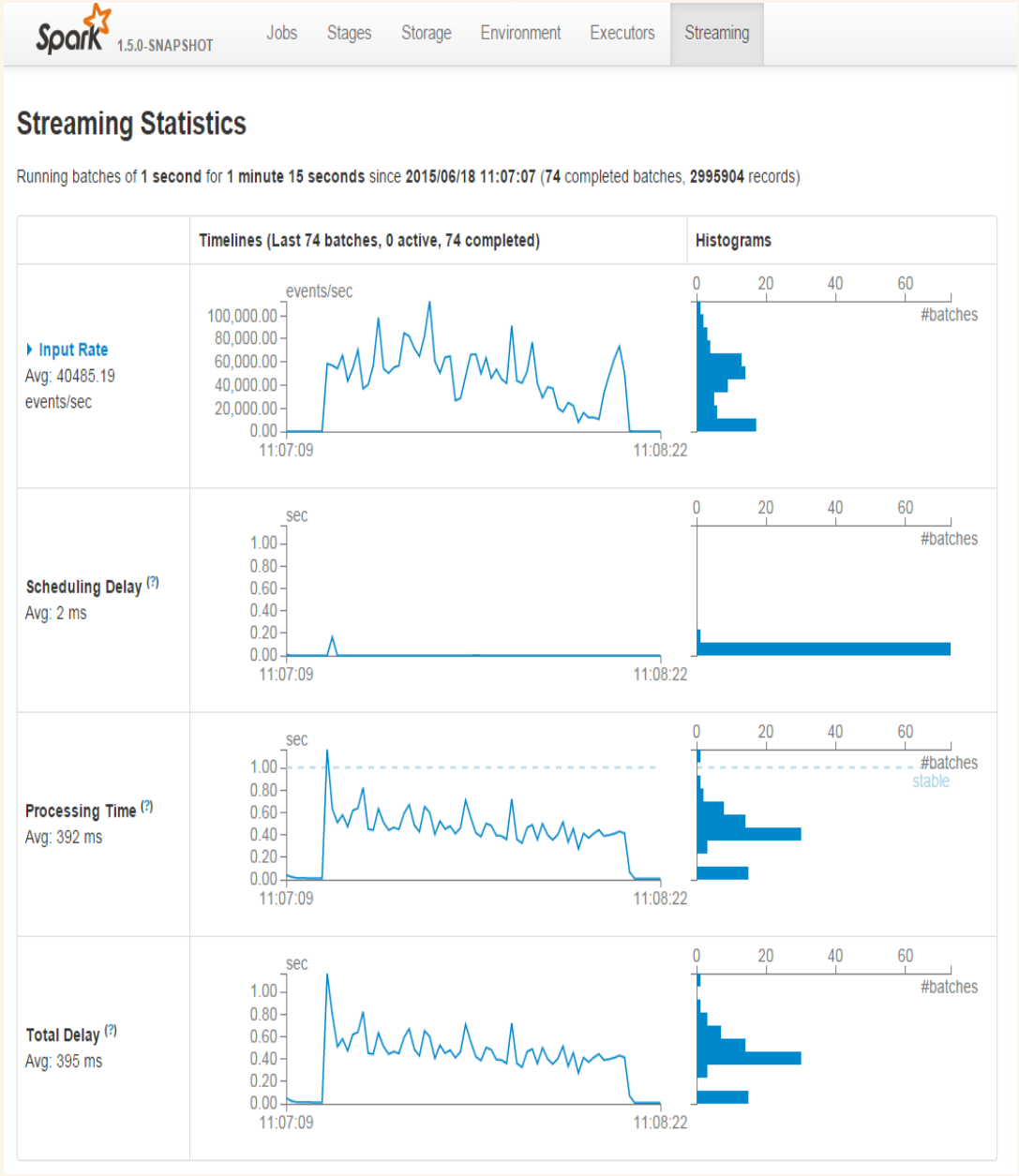
Direct Kafka Stream Approach



1. Read the Offset ranges and save in Checkpoint Dir
2. During RDD Processing data is fetched from Kafka

Primary Issue :

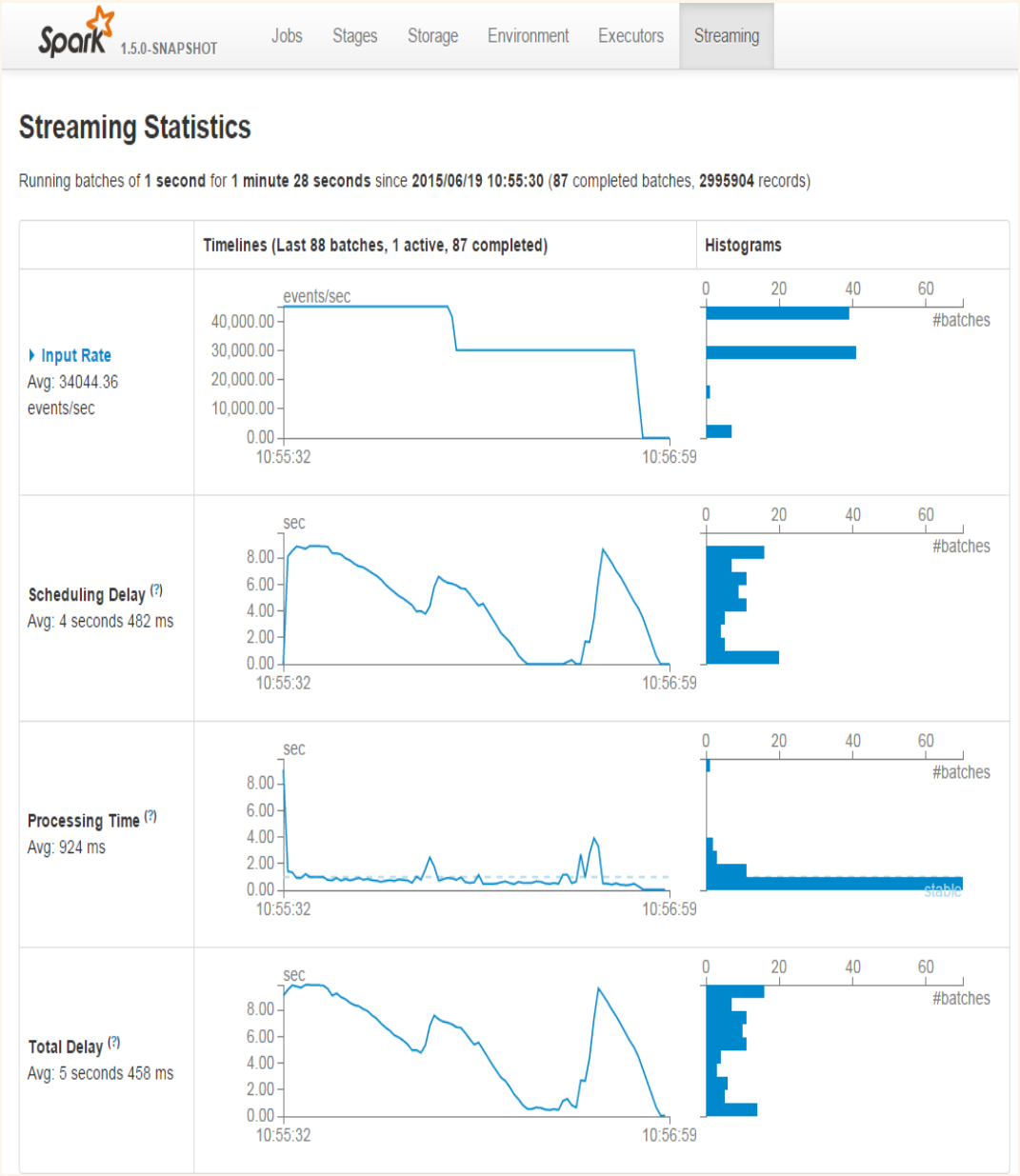
1. If you modify Driver code , Spark can not recover Checkpoint Dir
2. You may need to manage own offset in your code (complex)



dibbhatt/kafka-spark-consumer

Date Rate : 10MB/250ms per Receiver

40 x 3 MB / Sec



Direct Stream Approach

What is the primary reason for higher delay ?

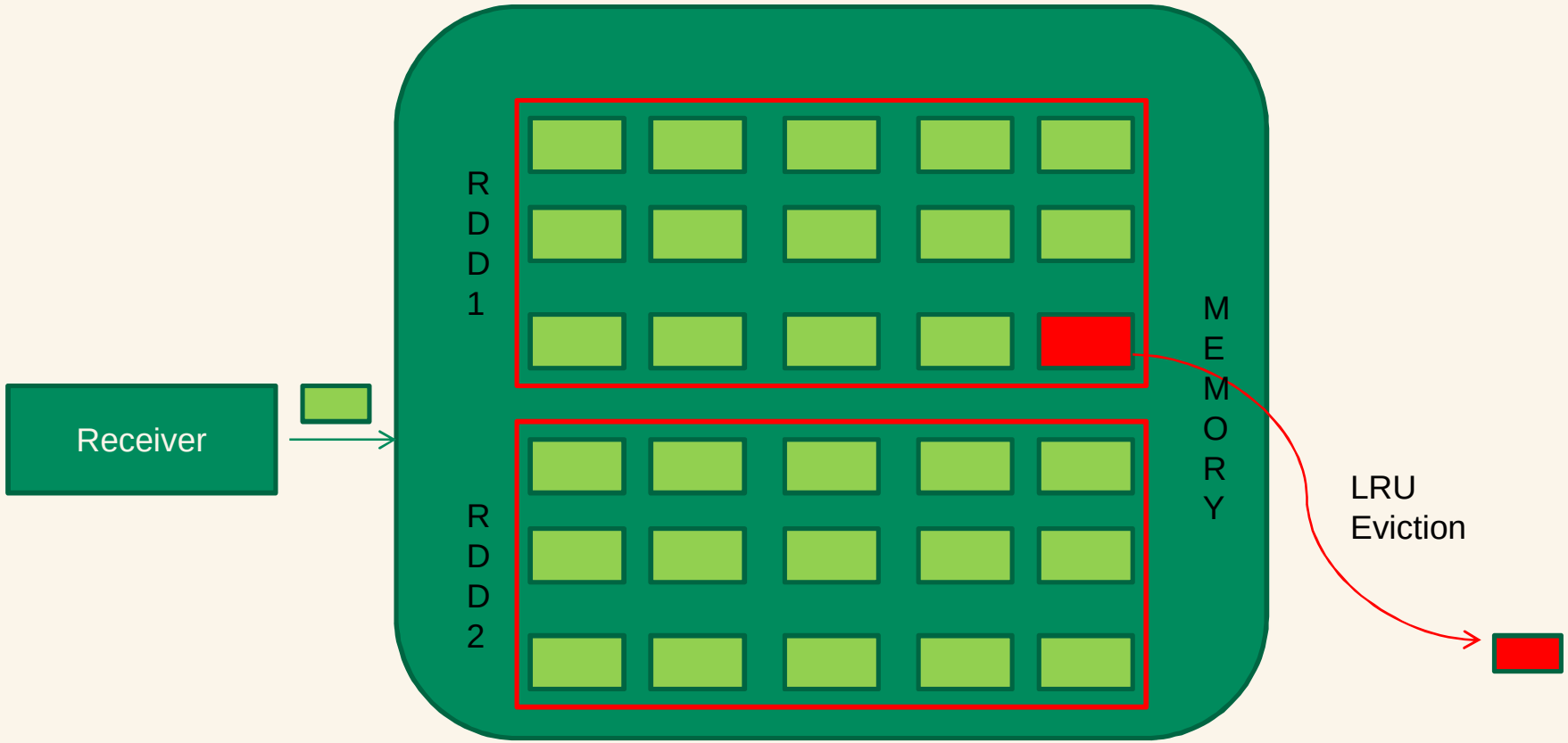
What is the primary reason for higher processing time ?

PID Controller - Spark Memory Back Pressure

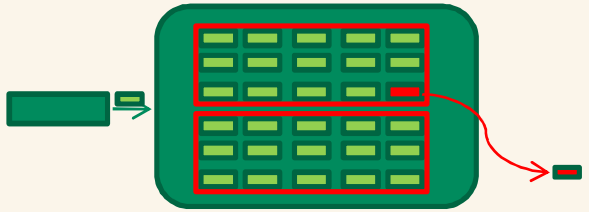
200 Ms Block Interval and 3 Second Batch Interval ..Let assume there is No Replication

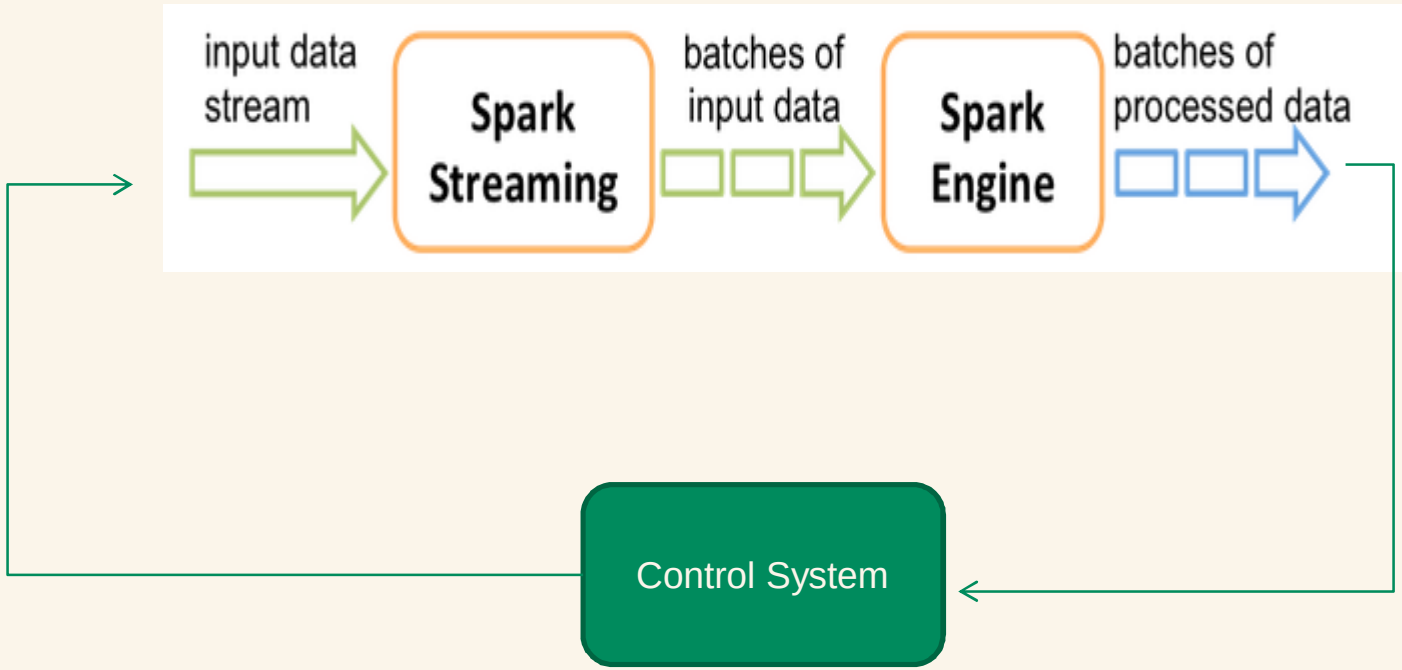
Every RDD is associated with a StorageLevel. MEMORY_ONLY, MEMORY_DISK, OFF_HEAP

BlockManager Storage Space is Limited , so as the Disk space..



Control System





PID Controller

Primary difference between PID Controller in this Consumer and what comes within Spark 1.5

Spark 1.5 Control the number of messages ..

dibbhatt/kafka-spark-consumer control the size of every block fetch from Kafka .

How does that matter ..?

Can throttling by number of messages will guarantee Memory size reduction ? What if you have messages of varying size

Throttling the number of messages after consuming large volume of data from Kafka caused unnecessary I/O. My consumer throttle at the source.

Apache Blur

Pearson Search Services : Why Blur

We are presently evaluating Apache Blur which is Distributed Search engine built on top of Hadoop and Lucene.

Primary reason for using Blur is ..

- Distributed Search Platform stores Indexes in HDFS.
- Leverages all goodness built into the Hadoop and Lucene stack

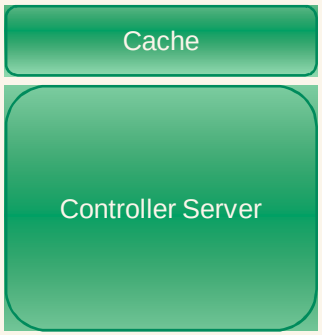
Benefit	Description
Scalable	Store , Index and Search massive amount of data from HDFS
Fast	Performance similar to standard Lucene implementation
Durable	Provided by built in WAL like store
Fault Tolerant	Auto detect node failure and re-assigns indexes to surviving nodes
Query	Support all standard Lucene queries and Join queries

"HDFS-based indexing is valuable when folks are also using Hadoop for other purposes (MapReduce, SQL queries, HBase, etc.). There are considerable operational efficiencies to a shared storage system. For example, disk space, users, etc. can be centrally managed." - Doug Cutting

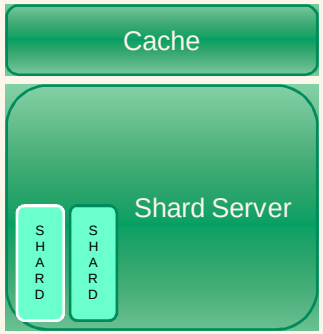
Blur Architecture

Components	Purpose
Lucene	Perform actual search duties
HDFS	Store Lucene Index
Map Reduce	Use Hadoop MR for batch indexing
Thrift	Inter Process Communication
Zookeeper	Manage System State and stores Metadata

Blur uses two types of Server Processes
- Controller Server
- Shard Server

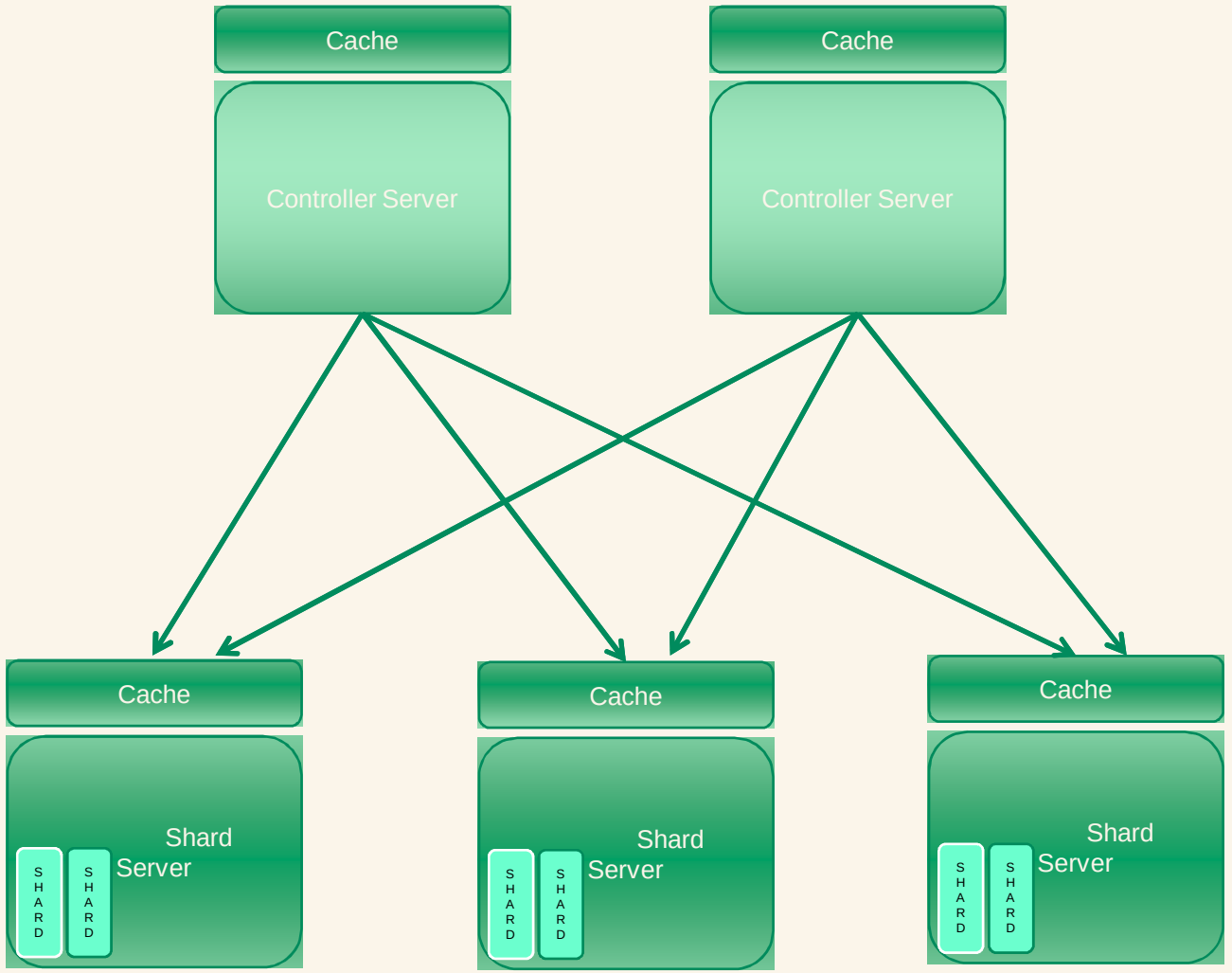


Orchestrate Communication between all Shard Servers for communication



Responsible for performing searches for all shard and returns results to controller

Blur Architecture



Major Challenges Blur Solved

Random Access Latency w/HDFS

Problem :

HDFS is a great file system for streaming large amounts data across large scale clusters. However the random access latency is typically the same performance you would get in reading from a local drive if the data you are trying to access is not in the operating systems file cache. In other words every access to HDFS is similar to a local read with a cache miss. Lucene relies on file system caching or MMAP of index for performance when executing queries on a single machine with a normal OS file system. Most of time the Lucene index files are cached by the operating system's file system cache.

Solution:

Blur have a Lucene Directory level block cache to store the hot blocks from the files that Lucene uses for searching. a concurrent LRU map stores the location of the blocks in pre allocated slabs of memory. The slabs of memory are allocated at start-up and in essence are used in place of OS file system cache.

Blur Data Structure

Blur is a table based query system. So within a single cluster there can be many different tables, each with a different schema, shard size, analyzers, etc. Each table contains Rows. A Row contains a row id (Lucene StringField internally) and many Records. A record has a record id (Lucene StringField internally), a family (Lucene StringField internally), and many Columns. A column contains a name and value, both are Strings in the API but the value can be interpreted as different types. All base Lucene Field types are supported, Text, String, Long, Int, Double, and Float.

Row Query :

- execute queries across Records within the same Row.
- similar idea to an inner join.

find all the Rows that contain a Record with the family "author" and has a "name" Column that has that contains a term "Jon" and another Record with the family "docs" and has a "body" Column with a term of "Hadoop".

+<author.name:Jon> +<docs.body:Hadoop>

```
Row {
  "id" => "r-5678",
  "records" => [
    Record {
      "recordId" => "1234",
      "family" => "family1",
      "columns" => [
        Column {"column1" => "value1"},
        Column {"column2" => "value2"},
        Column {"column2" => "value3"},
        Column {"column3" => "value4"}
      ]
    },
    Record {
      "recordId" => "9012",
      "family" => "family1",
      "columns" => [
        Column {"column1" => "value1"}
      ]
    },
    Record {
      "recordId" => "4321",
      "family" => "family2",
      "columns" => [
        Column {"column16" => "value1"}
      ]
    }
  ]
}
```

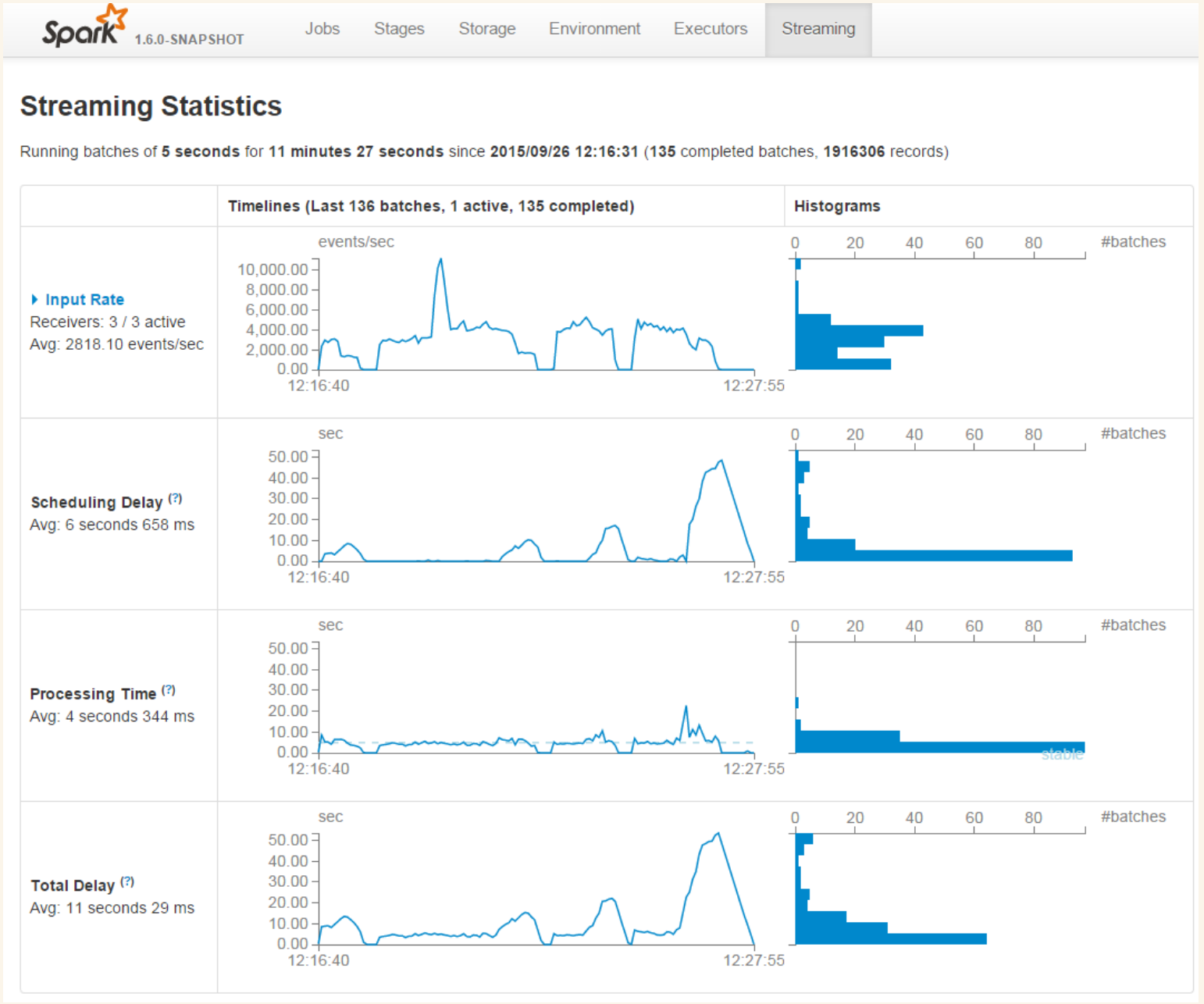
Spark Streaming Indexing to Blur

```
final String checkpointDirectory = "hdfs://10.252.5.113:9000/user/hadoop/spark/wal";
JavaStreamingContextFactory contextFactory = new JavaStreamingContextFactory() {

      @Override
      public JavaStreamingContext create() {

            int numberOfReceivers = 3;

            Properties props = new Properties();
            props.put("zookeeper.hosts", "10.252.1.136");
            props.put("zookeeper.port", "2181");
            props.put("zookeeper.broker.path", "/brokers");
            props.put("kafka.topic", "valid_subpub_2.0");
            props.put("kafka.consumer.id", "daalt-subpub");
            props.put("zookeeper.consumer.connection", "10.252.5.113:2182");
            props.put("zookeeper.consumer.path", "/spark-kafka");
            props.put("consumer.forcefromstart", "true");
            props.put("consumer.fetchsizebytes", "524288");
            props.put("consumer.fillfreqms", "200");
            props.put("consumer.backpressure.enabled", "true");

            SparkConf conf = new SparkConf().setAppName("BlurIndexer")
                .set("spark.executor.extraClassPath","/home/apache-blur-0.2.4/lib/*")
                .set("spark.serializer","org.apache.spark.serializer.KryoSerializer");

            JavaStreamingContext jsc = new JavaStreamingContext(conf, new Duration(5000));
            jsc.checkpoint(checkpointDirectory);
            JavaDStream<MessageAndMetadata> unionStreams = ReceiverLauncher.launch
                    (jsc, props, numberOfReceivers, StorageLevel.MEMORY_AND_DISK());

            BlurPairFunction pairFunc = new BlurPairFunction();
            JavaPairDStream<Text, BlurMutate> pairDStream = unionStreams.mapToPair(pairFunc);
            BlurIndexFunction indexFunc = new BlurIndexFunction();
            pairDStream.foreachRDD(indexFunc);
            return jsc;
      }
    };

JavaStreamingContext jsc = JavaStreamingContext.getOrCreate(checkpointDirectory, contextFactory);
jsc.start();
jsc.awaitTermination();
```

```
public class BlurPairFunction implements PairFunction<MessageAndMetadata, Text, BlurMutate>{

    public Tuple2<Text, BlurMutate> call(MessageAndMetadata mmeta) {

        String message = new String(mmeta.getPayload());
        String keyStr = DigestUtils.shaHex(message);
        BlurMutate mutate = new BlurMutate(BlurMutate.MUTATE_TYPE.REPLACE, keyStr, keyStr, "revel");

        Text key = new Text((keyStr).getBytes());
        mutate.addColumn("raw", message);
        mutate.addColumn("topic", mmeta.getTopic());

        return new Tuple2<Text, BlurMutate>(key, mutate);
    }

}
```

```
public class BlurIndexFunction implements Function2<JavaPairRDD<Text, BlurMutate>, Time, Void>{

    private static final long serialVersionUID = 88875777435L;

    @Override
    public Void call(JavaPairRDD<Text, BlurMutate> rdd,
            Time time) throws Exception {

        TableDescriptor tableDescriptor = new TableDescriptor();
        String tableUri = new Path("hdfs://10.252.5.113:9000/blur/tables/nrt").toString();
        tableDescriptor.tableUri = tableUri;
        tableDescriptor.cluster = "pearson";
        tableDescriptor.name = "nrt";
        tableDescriptor.shardCount = 6;
        Configuration conf = new Configuration();

        final JavaPairRDD<Text, BlurMutate> pRdd = rdd.partitionBy(new BlurSparkPartitioner(tableDescriptor.shardCount));

        BlurOutputFormat.setIndexLocally(conf, false);
        conf.setClass("mapreduce.reduce.class", DefaultBlurReducer.class, Reducer.class);
        conf.setClass("mapreduce.outputformat.class", BlurOutputFormat.class, OutputFormat.class);
        conf.setClass("mapreduce.partitioner.class", BlurPartitioner.class, Partitioner.class);
        conf.set("mapred.output.committer.class", BlurOutputCommitter.class.getName());
        conf.setInt("blur.output.max.document.buffer.size", 10000);
        conf.set("blur.output.path", tableDescriptor.getTableUri());

        BlurOutputFormat.setTableDescriptor(conf, tableDescriptor);

        JobConf jobConf = new JobConf(conf);

        jobConf.setNumReduceTasks(tableDescriptor.shardCount);
        jobConf.setOutputKeyClass(Text.class);
        jobConf.setOutputValueClass(BlurMutate.class);

        BlurMapReduceUtil.addAllJarsInBlurLib(conf);

        BlurMapReduceUtil.addDependencyJars(conf, org.apache.zookeeper.ZooKeeper.class,
                        org.apache.lucene.codecs.lucene42.Lucene42Codec.class,
                        jobConf.getOutputKeyClass(),
                        jobConf.getOutputValueClass());


        if (pRdd.count() > 0){
            try{

                pRdd.saveAsNewAPIHadoopFile(tableUri, Text.class,
                        BlurMutate.class, BlurOutputFormat.class,
                        jobConf);

            }catch(Exception e){

                e.printStackTrace();
            }

        }
```

Demo


Spark 1.6.0-SNAPSHOT
Jobs
Stages
Storage
Environment
Executors
Streaming
Streaming Statistics
Running batches of 5 seconds for 11 minutes 27 seconds since 2015/09/26 12:16:31 (135 completed batches, 1916306 records)
Timelines (Last 136 batches, 1 active, 135 completed)
Histograms
Input Rate
Receivers: 3 / 3 active
Avg: 2818.10 events/sec
events/sec
10,000.00
8,000.00
6,000.00
4,000.00
2,000.00
0.00
12:16:40
12:27:55
0
20
40
60
80
#batches
Scheduling Delay (?)
Avg: 6 seconds 658 ms
sec
50.00
40.00
30.00
20.00
10.00
0.00
Processing Time (?)
Avg: 4 seconds 344 ms
stable
Total Delay (?)
Avg: 11 seconds 29 ms

Thank You !