CLAIRVOYANT

design engineer deliver

Spark Streaming

Clairvoyant Big Data Team, Pune

CLAIRVOYANT

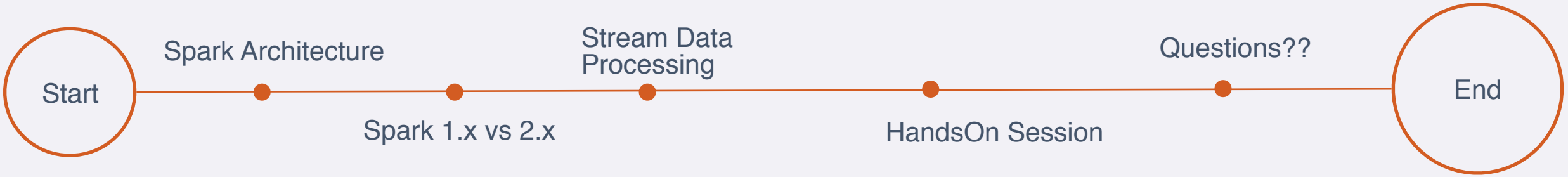
Scope of this talk

- What we will cover?
  - High Level Spark Architecture
  - Spark Streaming - Basic to Mid level concepts
  - Streaming Concepts in general
  - Enable the users to get started with Simple Spark Streaming Apps
- What we will not cover? (due to time constraints)
  - Low level details of the Spark Architecture and individual components
  - SparkSQL, DataFrames, DataSets, MLLib etc
  - Advanced streaming features in Spark
  - Details of Performance Tuning

Disclaimers

- Images: Most of the Images in the slide desk have been picked from existing online Spark books in the interest of saving time.
  - Thanks to the authors of the corresponding books

Presentation Agenda



Quick Glance

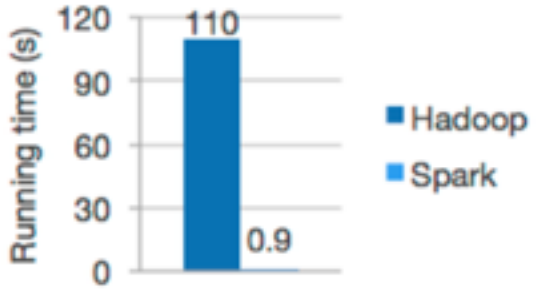
Spark and its Architecture

What is Spark??

Speed

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.



Logistic regression in Hadoop and Spark

Ease of Use

Write applications quickly in Java, Scala, Python, R.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala, Python and R shells.

```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

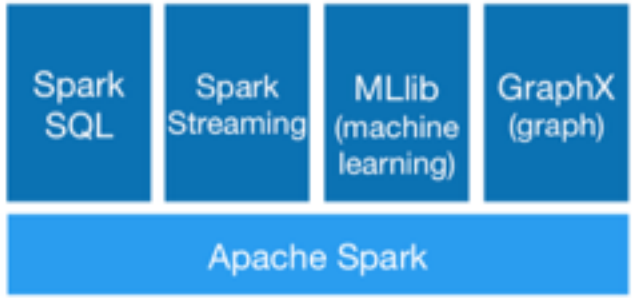
Word count in Spark's Python API

What is Spark??

Generality

Combine SQL, streaming, and complex analytics.

Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.

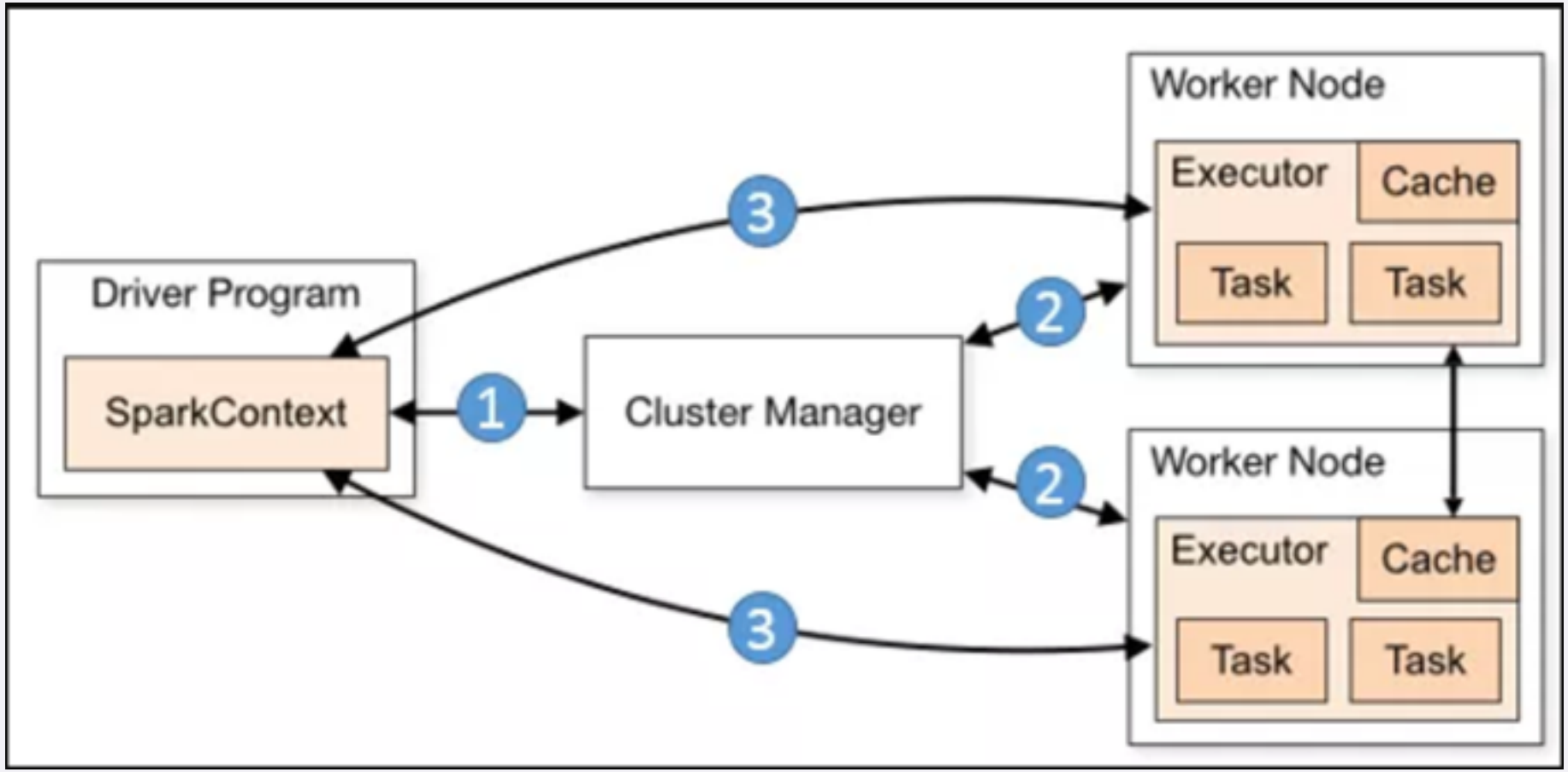


Runs Everywhere

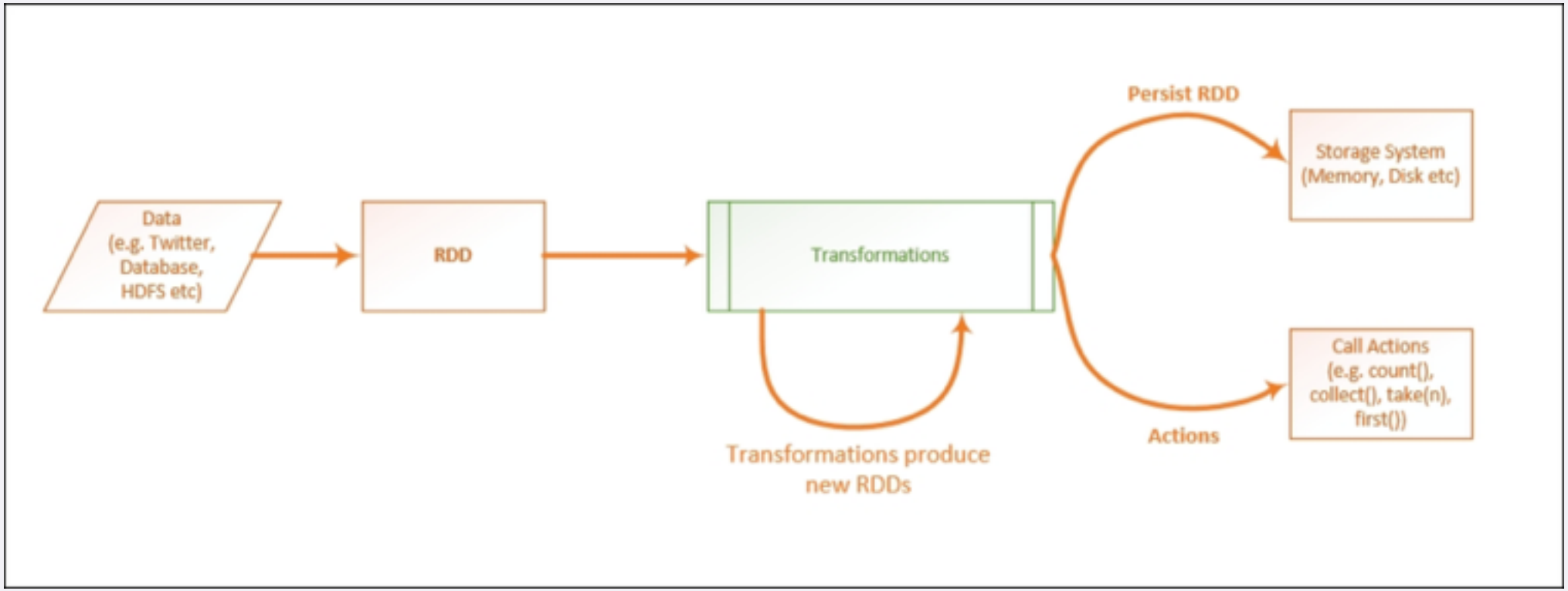
Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

You can run Spark using its standalone cluster mode, on EC2, on Hadoop YARN, or on Apache Mesos. Access data in HDFS, Cassandra, HBase, Hive, Tachyon, and any Hadoop data source.

Application Execution model



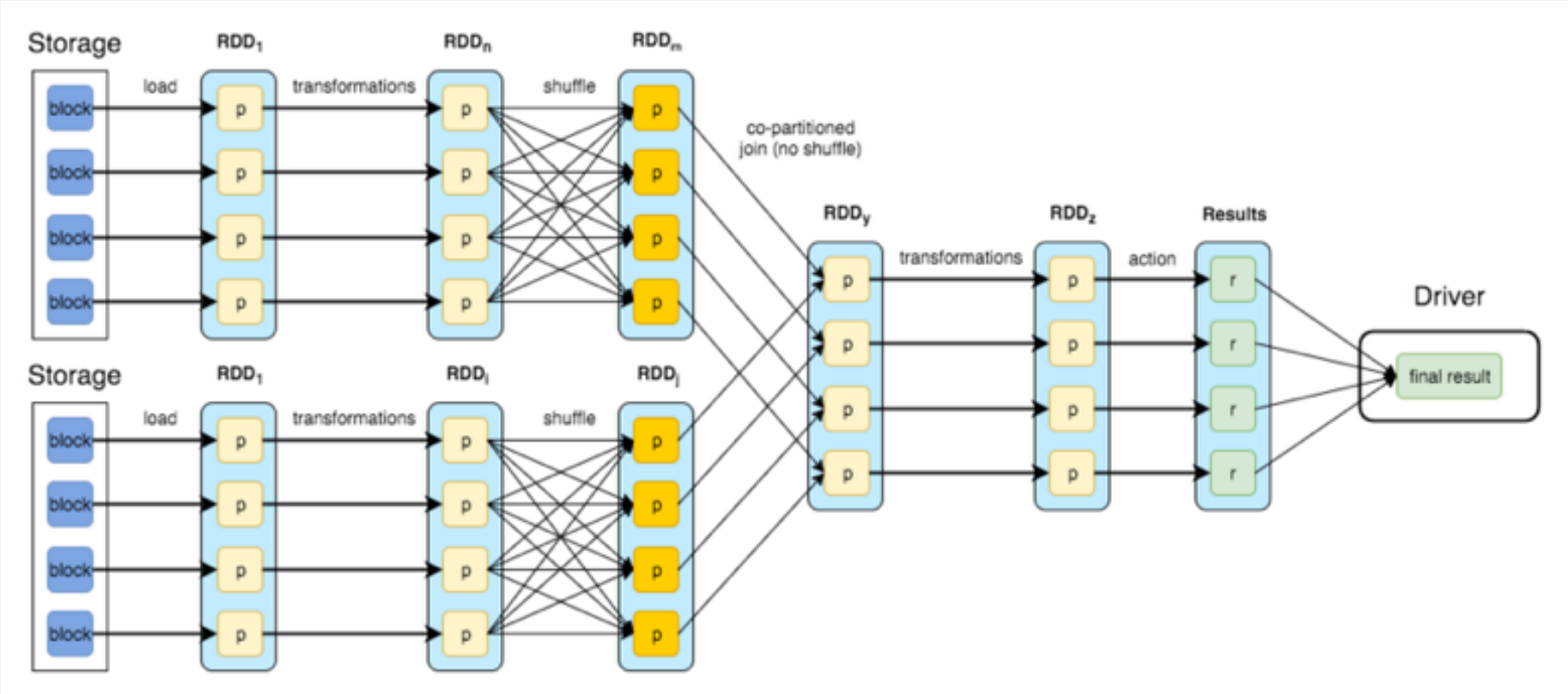
Data Computation Model



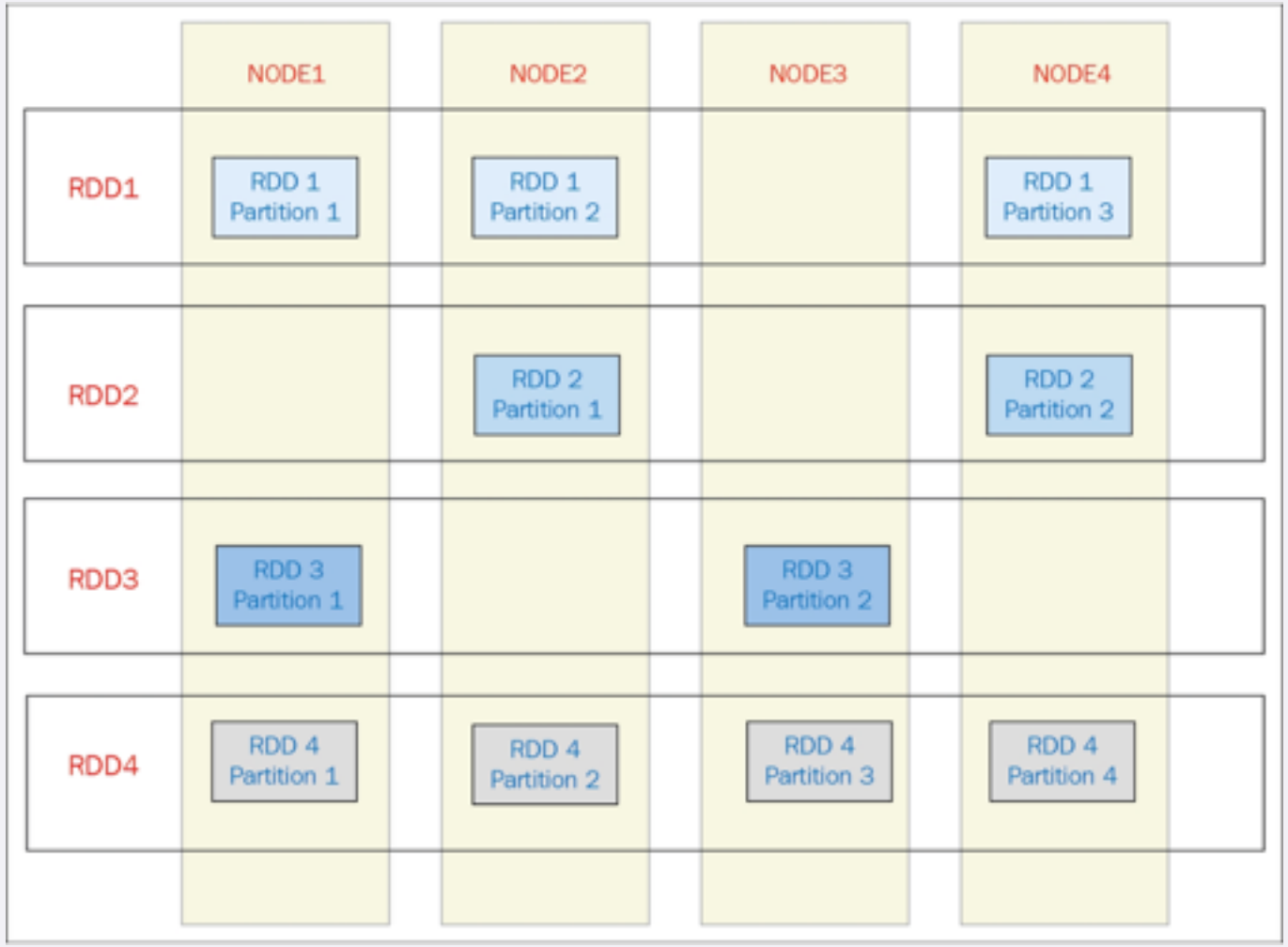
Resilient Distributed Datasets

- The basic abstraction to holding the data in a distributed manner along with the details of how it is computed
  - A list of splits (**partitions**)
  - A function for **computing** each split
  - A list of **dependencies** on other RDDs
  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)

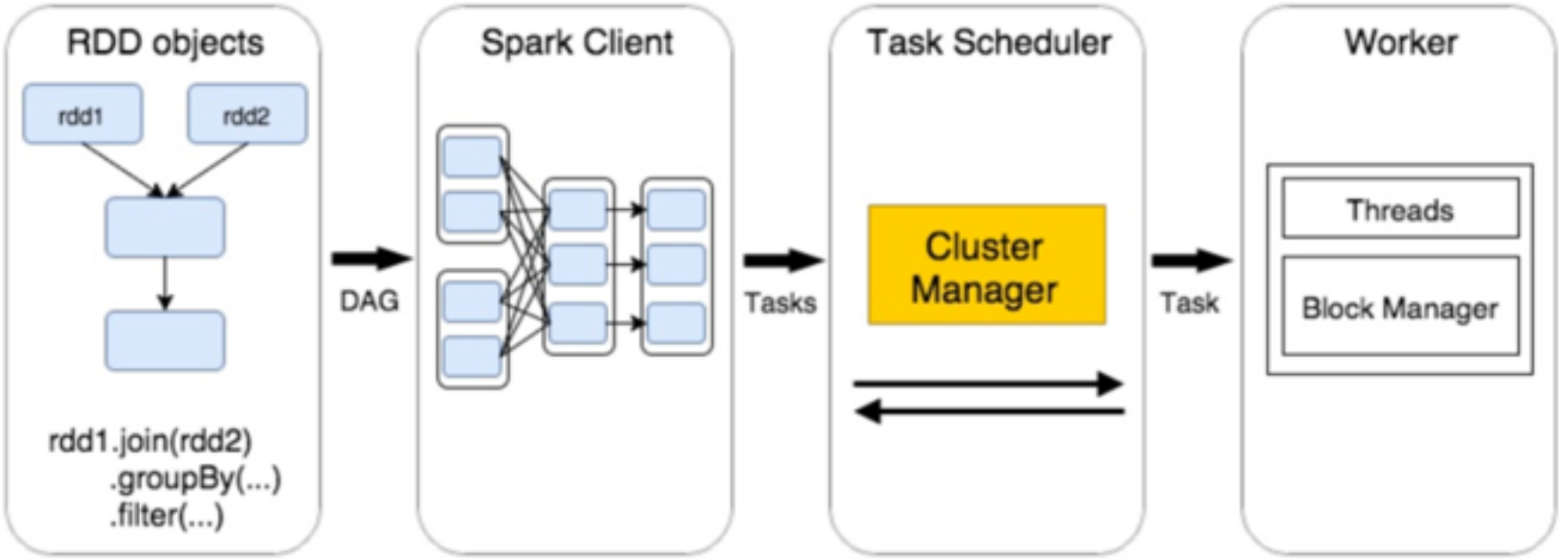
Resilient (split level computation and dependencies)



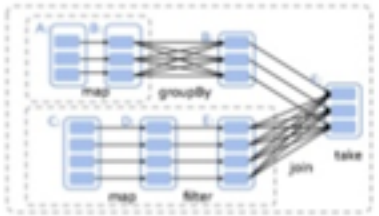
Distributed Datasets



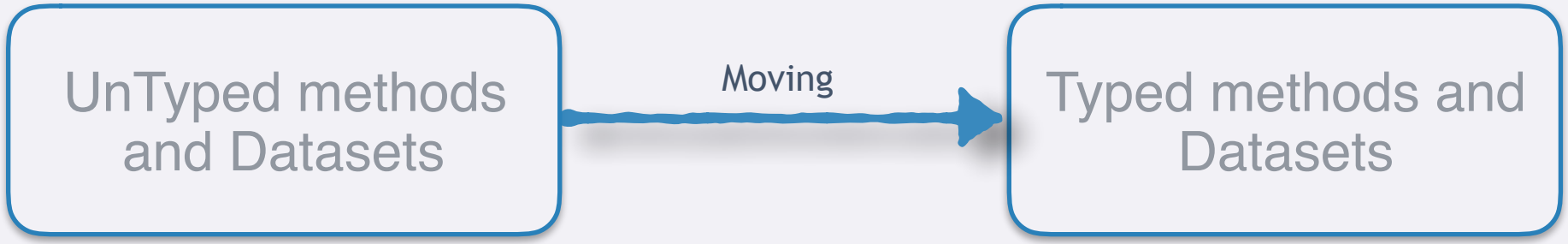
Architecture Spark



- RDD operations(transformations and actions) form DAG
- DAG is split into stages of tasks which are then submitted to cluster manager
- stages combine tasks which don't require shuffling/repartitioning
- tasks run on workers and results then return to client



Spark 1.x vs 2.x



- Making Spark Structure aware (DataSets)
  - Make the required optimisations
  - Unifications across modules

Spark 1.x vs 2.x

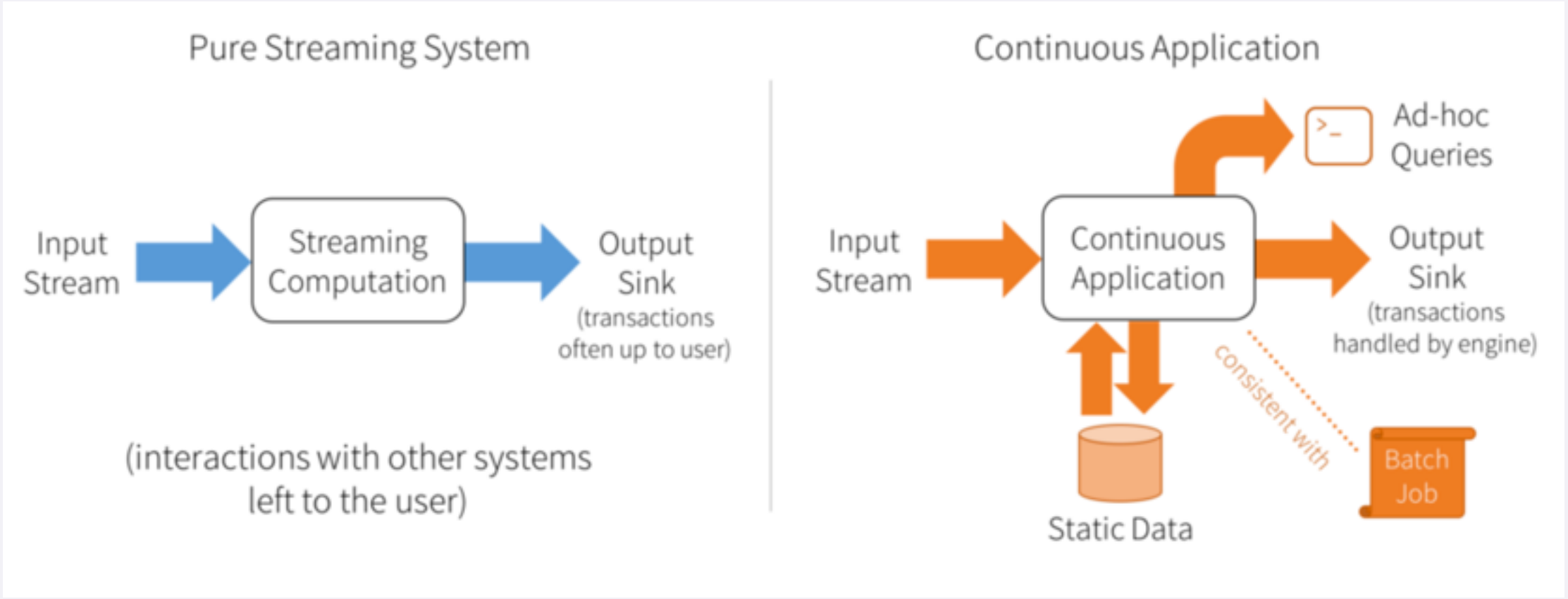
- Usability
  - Standard SQL support (SQL:2003 features)
  - Unifying DataFrame/Dataset API.
  - SparkSession (get rid of SQL/Hive contexts)
- Performance
  - whole-stage code generation
    - optimized bytecode at runtime
    - leveraging CPU registers for intermediate data
  - Catalyst optimizer

Spark 1.x vs 2.x

- Intelligent
  - Structured Streaming
    - integration of the batch stack and the streaming stack
    - integration with external storage systems
    - ability to cope with changes

continuous applications

Continuous Applications



Structured Streaming

- Strong guarantees about consistency with batch jobs
  - write computations using Spark's DataFrame/Dataset API
  - engine automatically incrementalizes this computation
  - Structured Streaming output = Batch job on a prefix of the input data
- Transactional integration with storage systems
  - process data exactly once
  - update output sinks transactionally
- Tight integration with the rest of Spark
  - interactive queries on streaming state using SparkSQL/JDBC etc

Quick Example

Batch

```
// Read JSON once from S3
logsDF = spark.read.json("s3://logs")

// Transform with DataFrame API and save
logsDF.select("user", "url", "date")
      .write.parquet("s3://out")
```

Streaming

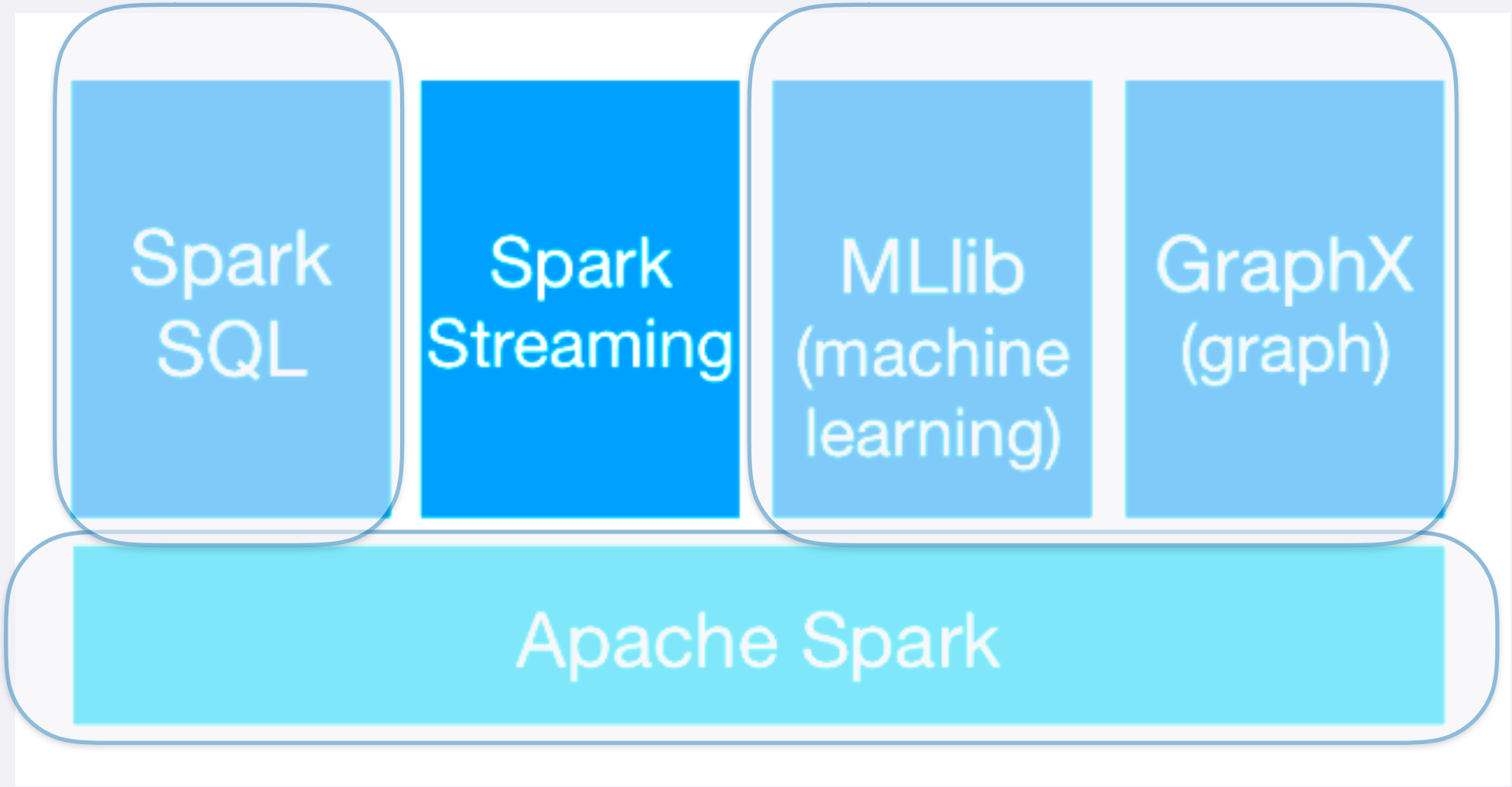
```
// Read JSON continuously from S3
logsDF = spark.readStream.json("s3://logs")

// Transform with DataFrame API and save
logsDF.select("user", "url", "date")
      .writeStream.parquet("s3://out")
      .start()
```

Spark 2.x

In the next Spark Workshop (in a couple weeks)

Stream Data Processing


Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)
Apache Spark

What is Spark Streaming?

Core API Extension for stream processing of live data streams

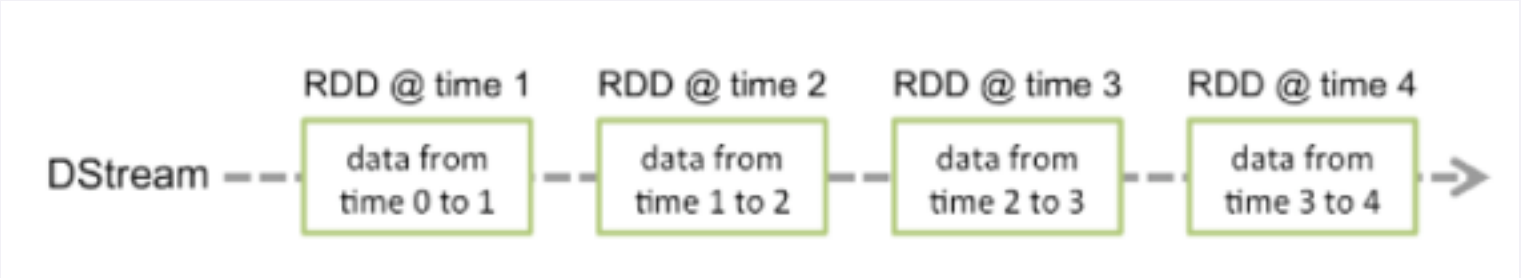
- scalable
- high-throughput
- fault-tolerant

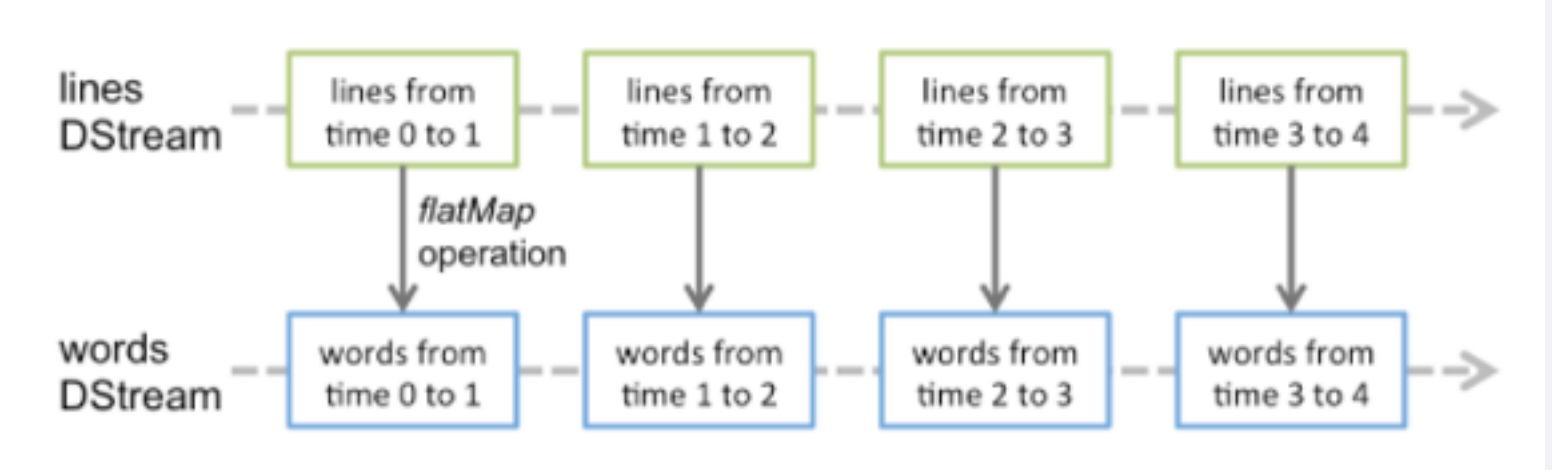
DStreams

A DStream internally is characterized by a few basic properties:

- A list of other DStreams that the DStream depends on
- A time interval at which the DStream generates an RDD
- A function that is used to generate an RDD after each time interval

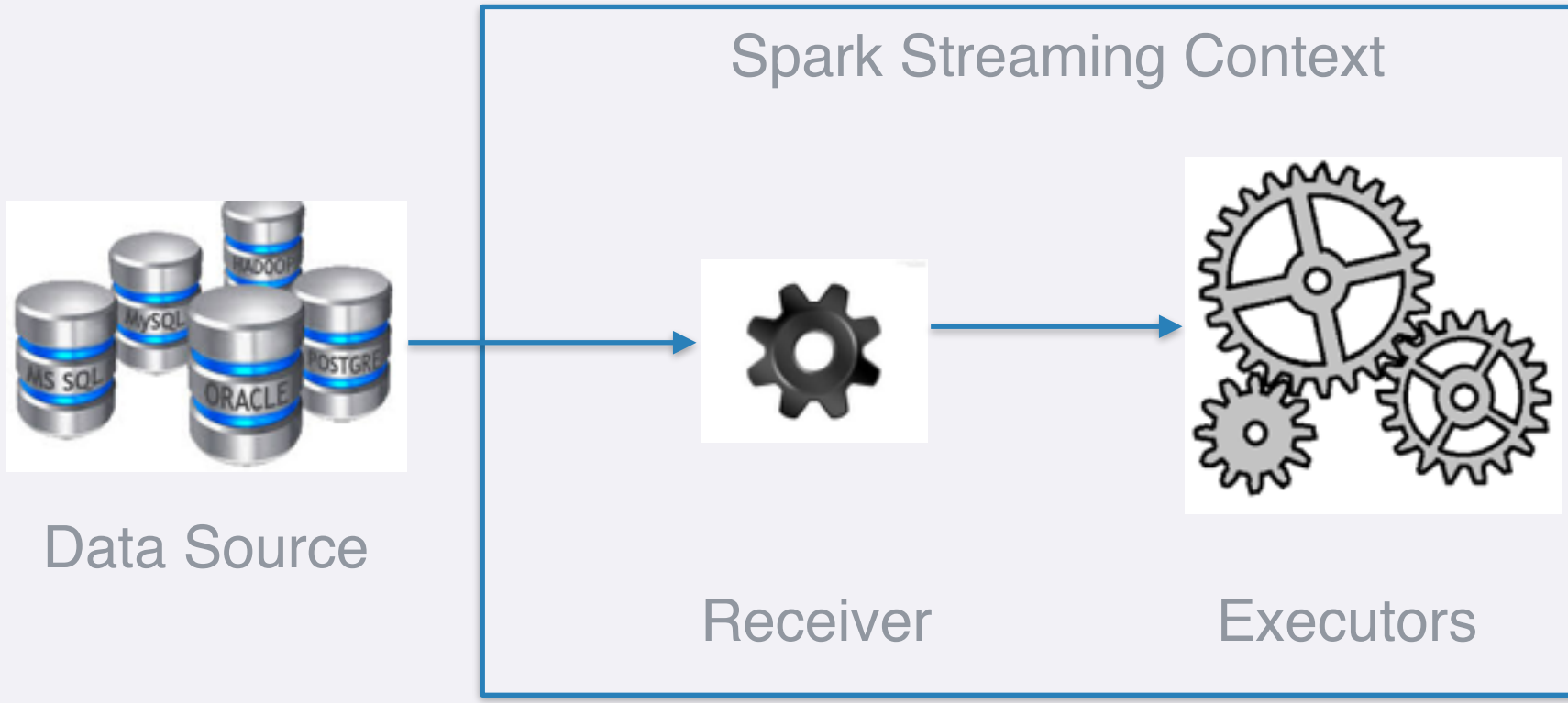
DStreams


RDD @ time 1
RDD @ time 2
RDD @ time 3
RDD @ time 4
DStream
data from time 0 to 1
data from time 1 to 2
data from time 2 to 3
data from time 3 to 4


lines DStream
lines from time 0 to 1
lines from time 1 to 2
lines from time 2 to 3
lines from time 3 to 4
flatMap operation
words DStream
words from time 0 to 1
words from time 1 to 2
words from time 2 to 3
words from time 3 to 4

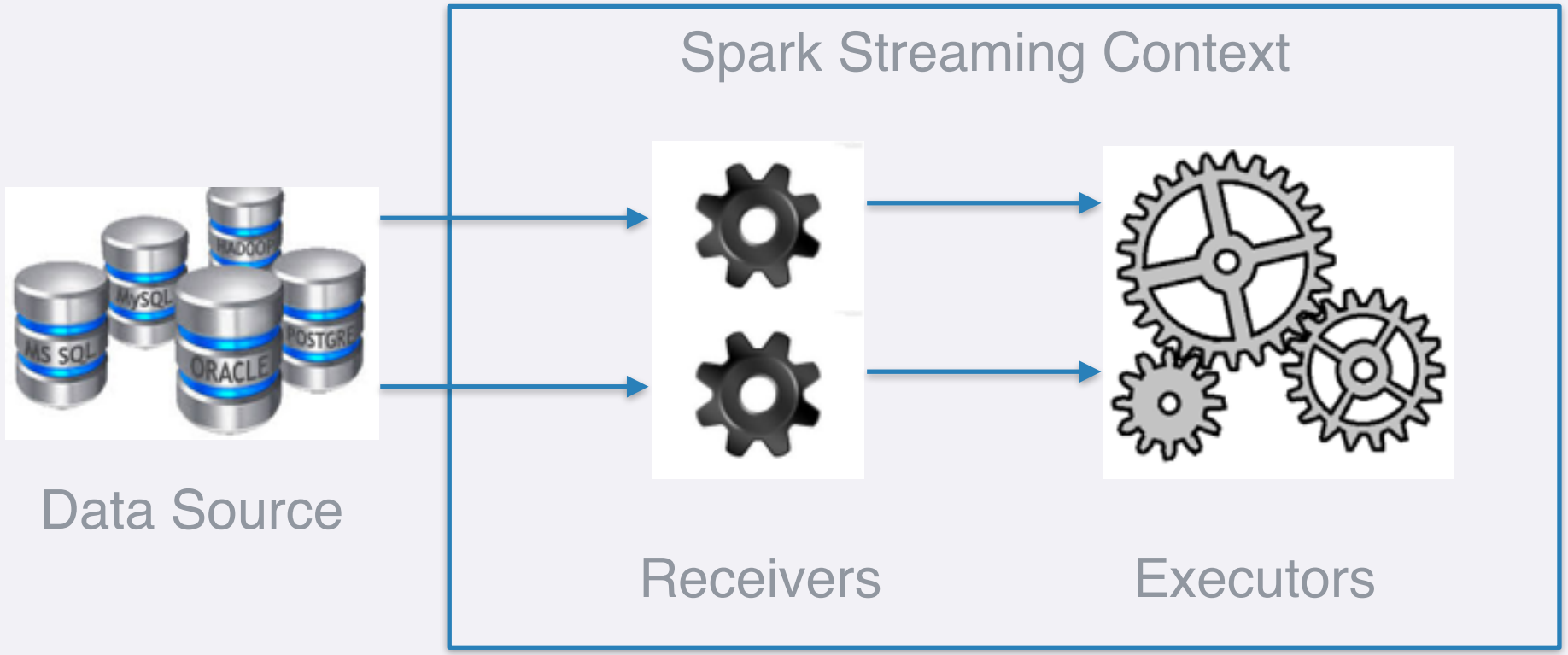
Data Sources (Single Receiver)

Receivers not applicable for File Streams



Data Sources (Multiple Receiver)

Num.Of Cores > Num.Of Receivers

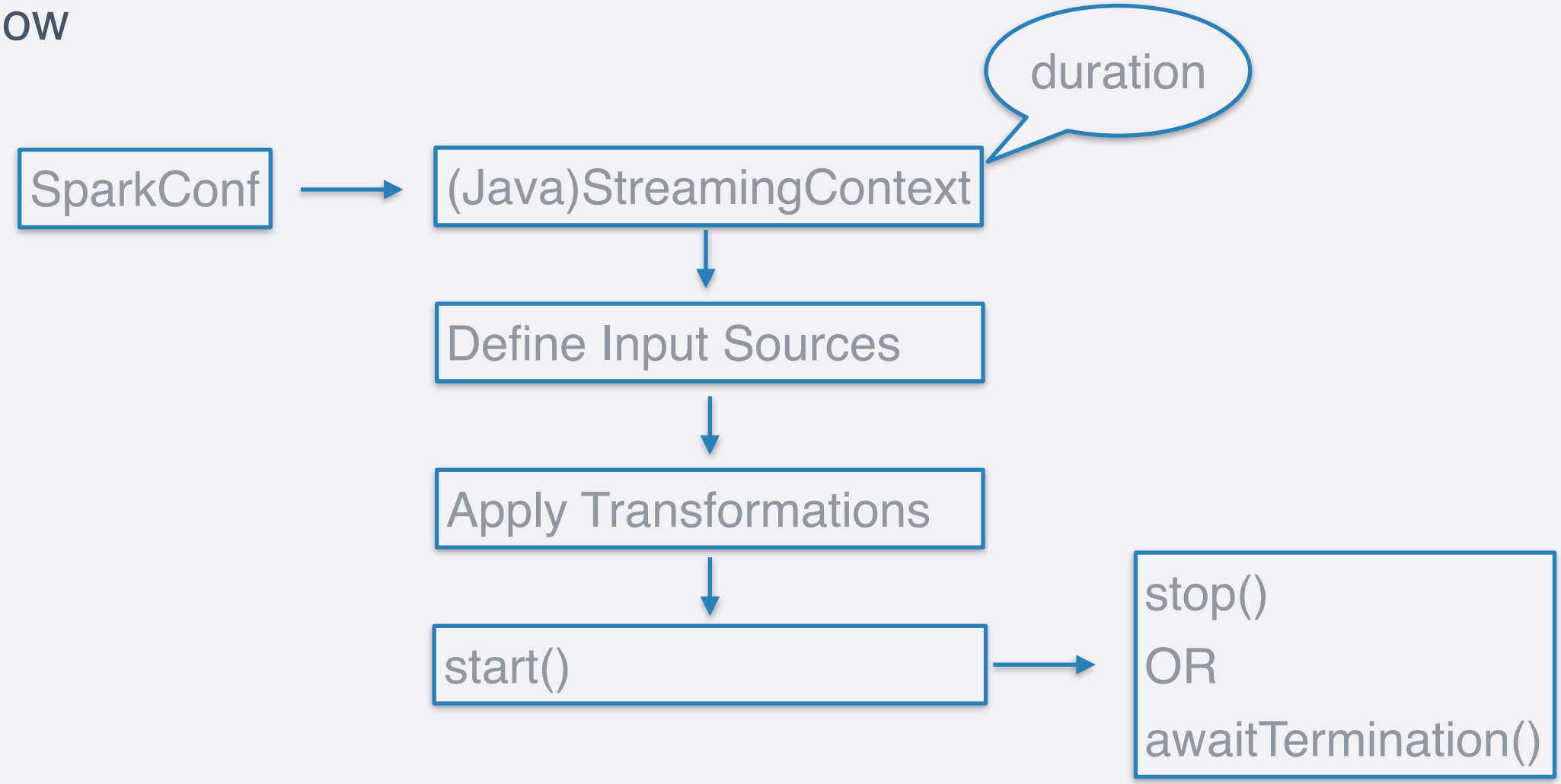


Custom Receivers

- Extend Abstract class Receiver
- Implement
  - onStart()
  - store(data)
  - onStop()
- restart() - calls stop() and then start()

Usage: ssc.receiverStream(new JavaCustomReceiver(host, port));

Basic Flow


duration
SparkConf
(Java)StreamingContext
Define Input Sources
Apply Transformations
start()
stop()
OR
awaitTermination()

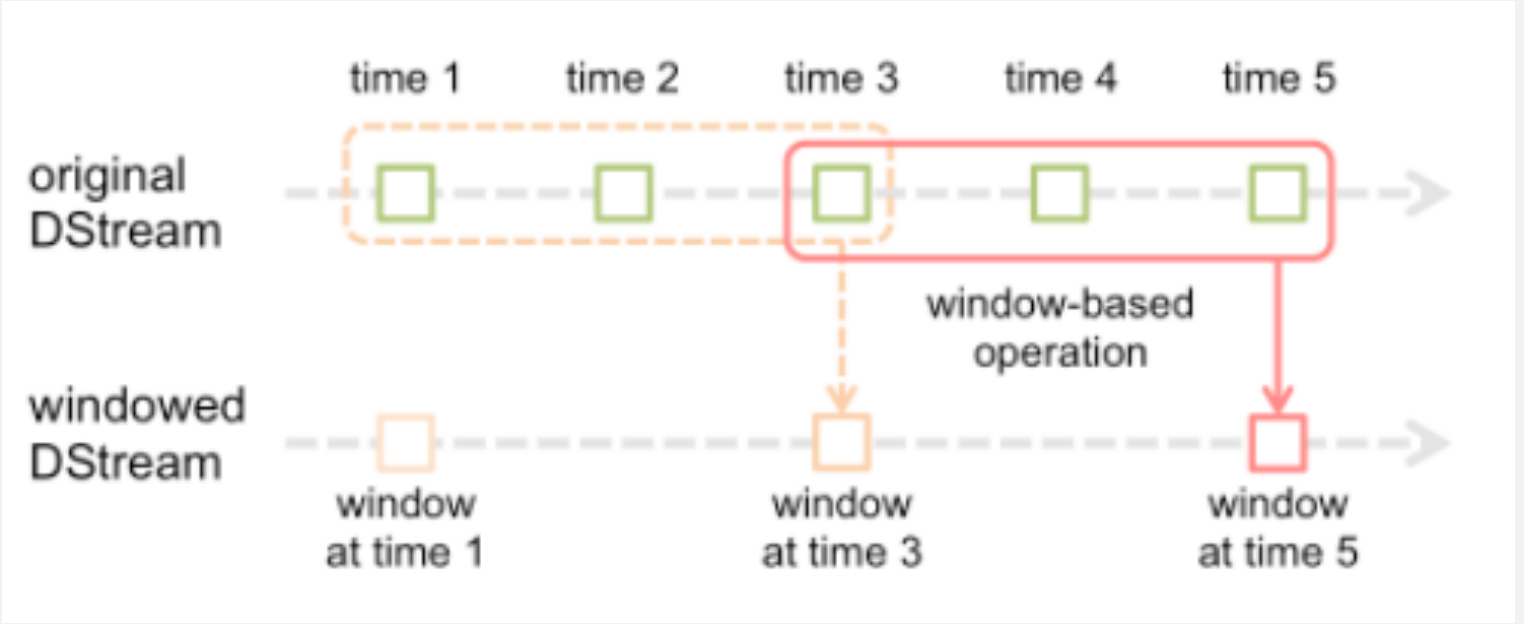
Quick Example

Window Operations

- BI - Batch Interval
- WL - Window Length
- SI - Sliding Interval

WL = n*BI

WL = m*BI



Transformations

- All standard transformations on RDD are available on Streams
- UpdateStateByKey
  - Create a state
  - Update the state for each batch (RDD) in the stream
- Joins on streams
  - stream1.join(stream2);
  - stream1.window(Durations.seconds(20))
    .join(stream2.window(Durations.minutes(1)))

Transformations

```
JavaPairRDD<String, String> dataset = ...
JavaPairDStream<String, String> windowedStream = stream.window(Durations.seconds(20));
JavaPairDStream<String, String> joinedStream = windowedStream.transform(
    new Function<JavaRDD<Tuple2<String, String>>, JavaRDD<Tuple2<String, String>>>() {
        @Override
        public JavaRDD<Tuple2<String, String>> call(JavaRDD<Tuple2<String, String>> rdd) {
            return rdd.join(dataset);
        }
    }
);
```


Output of DStreams

- print()
- saveAsTextFiles(prefix, [suffix])
- saveAsObjectFiles(prefix, [suffix])
- saveAsHadoopFiles(prefix, [suffix])
- foreachRDD(func)
  - The DStream is not printed.
  - func will be applied on each RDD
  - RDD.action is expected

Output of DStreams

- foreachRDD(func) - few limitations
  - connections used for writing data to ext tables/files
    - Mostly not serialisable.
      - Avoid creating connections on driver side.
    - Always better to have the connections created on worker side
      - But not for every Record. Use foreachPartition instead.
  - foreachRDD should apply some sort of action on the RDDs.
    - Else the DStream will be created but will not be of any use.

DataFrame and SQL Operations

words is a DStream

```
words.foreachRDD(
  new Function2<JavaRDD<String>, Time, Void>() {
    @Override
    public Void call(JavaRDD<String> rdd, Time time) {

     SQLContext sqlContext = SQLContext.getOrCreate(rdd.context());
       —
       —
      DataFrame wordsDataFrame = sqlContext.createDataFrame(rowRDD, JavaRow.class);
       —
       —
    }
  }
);
```


Checkpointing

- Store application state to be recover from system failures, JVM crashes
- Deal with Driver Failures
  - Metadata checkpointing
    - Configuration
    - DStream operations
    - Incomplete batches
- Deal with stateful transformations failures
  - Data checkpointing
    - RDDs to avoid recovery time

Checkpointing - How?

- Deal with stateful transformations failures
  - streamingContext.checkpoint(checkpointDirectory)
  - checkpointInterval —> 5*SI to 10*SI
- Deal with Driver Failures
  - Needs a bit of restructuring of the application
    - first time application run
      - create a new StreamingContext
    - failure recovery
      - re-create a StreamingContext from the checkpoint data
- Be careful with upgrades
  - either delete the checkpoint dirs or use new one.

HandsOn Session

Thank you

Backup Slides

Types of Receivers

Receiver Type	Quick Fact	Characteristics
Reliable	Acknowledge the source	• Strong fault-tolerance • Ensures zero data loss • Block generation and rate control to be custom handled • Acknowledgement complexity
UnReliable	Do not (bother to) acknowledge the source	• Simple to implement • Block generation and rate control will be handled by the system

Caching & persistence

- persist() on DStreams
  - persists every RDD in the stream.
  - May become costly at times. Use as needed.
- window operations by default apply persist() on the corresponding RDDs
- Persistence of data from external data sources
  - default with replication 2
  - in memory

Performance Tuning

- For distributed data sources, use multiple receivers.
  - Use multiple DStreams and union them if required.
- Set the batch size correctly so that data is processed in less time
- Increase the parallelism at task level
  - spark.streaming.blockInterval (not less than 50 ms)
  - noof tasks = batch interval / block interval

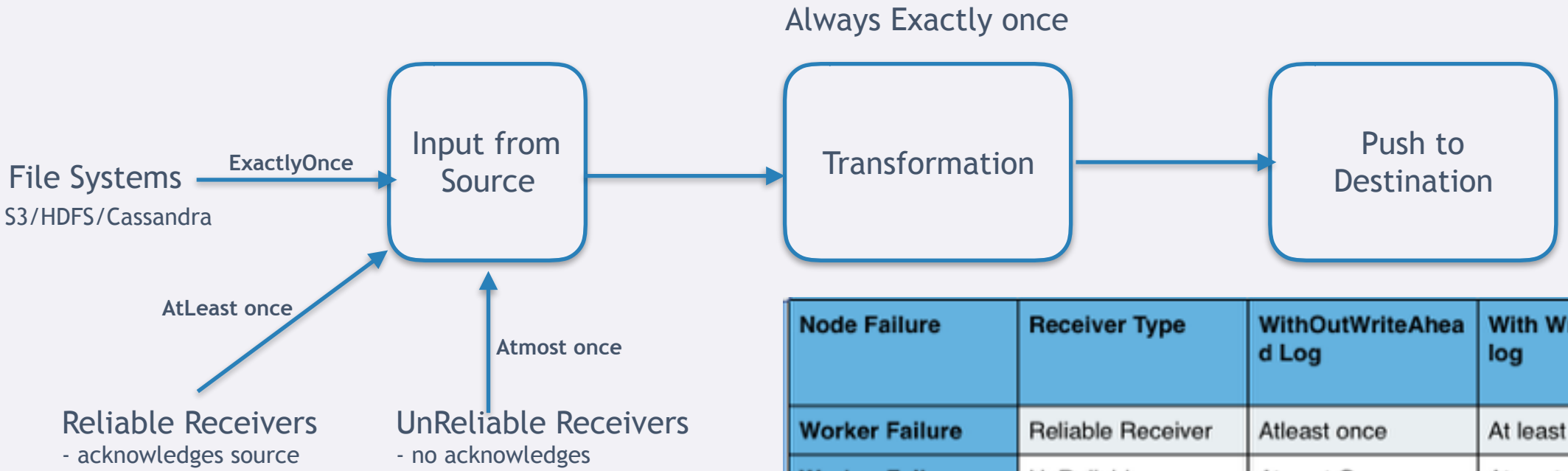
Fault-tolerance Semantics

- Batch Data processing jobs
  - Typically the sources (like HDFS, S3, Cassandra) of batch jobs are already Fault Tolerant
  - The RDDs are build for fault tolerance
- For Streaming Jobs
  - Data comes via Network (Source may or may not be fault tolerant)
  - Though data is replicated (by default factor 2) on executors, data can be lost
    - If worker nodes fail
      - executors failure causes loss of Data received and replicated
      - receiver failure causes Data Received and pending replication
    - If driver node fails
      - SparkContext is killed and all the data is lost

Guarantees provided by Streaming Systems

- At most once - Data Loss
- At least once - No Data Loss
- Exactly once - Ideal case

In Spark Streaming



Node Failure	Receiver Type	WithOutWriteAhead Log	With Writeahead log
Worker Failure	Reliable Receiver	Atleast once	At least once
Worker Failure	UnReliable Receiver	Atmost Once	Atmost Once
Driver Failure	Reliable Receiver	Atmost Once OR UnGuaranteed	At least once
Driver Failure	UnReliable Receiver	Atmost Once OR UnGuaranteed	At least once