Scalable Stream Processing
Spark Streaming and Flink Stream

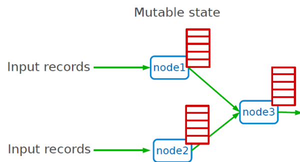
Amir H. Payberah
amir@sics.se

KTH Royal Institute of Technology

Spark Streaming

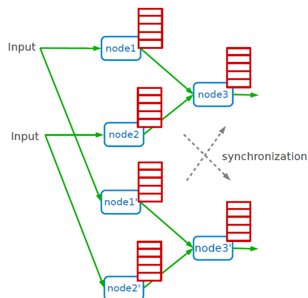
Existing Streaming Systems (1/2)

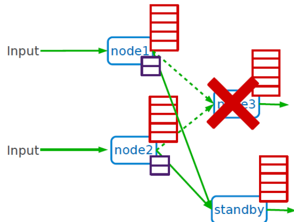
- Record-at-a-time processing model:
  - Each node has mutable state.
  - For each record, updates state and sends new records.
  - State is lost if node dies.



Existing Streaming Systems (2/2)

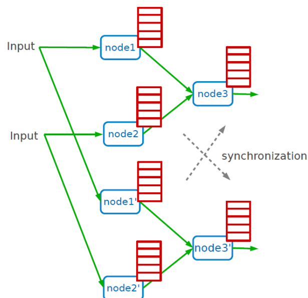
- Fault tolerance via replication or upstream backup.

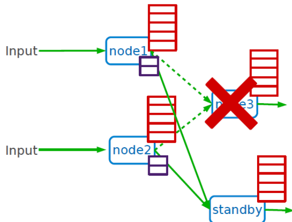




Existing Streaming Systems (2/2)

- Fault tolerance via replication or upstream backup.





Fast recovery, but 2x hardware cost

Only need one standby, but slow to recover

Observation

- Batch processing models for clusters provide fault tolerance efficiently.
- Divide job into deterministic tasks.
- Rerun failed/slow tasks in parallel on other nodes.

Core Idea

- Run a streaming computation as a series of very small and deterministic batch jobs.

Challenges

- Latency (interval granularity)
  - Traditional batch systems replicate state on-disk storage: slow

- Recovering quickly from faults and stragglers

Proposed Solution

- Latency (interval granularity)
  - Resilient Distributed Dataset (RDD)
  - Keep data in memory
  - No replication

- Recovering quickly from faults and stragglers
  - Storing the lineage graph
  - Using the determinism of D-Streams
  - Parallel recovery of a lost node's state

Programming Model

Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.

Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chop up the live stream into batches of X seconds.

Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chop up the live stream into batches of X seconds.
  - Spark treats each batch of data as RDDs and processes them using RDD operations.

Spark Streaming

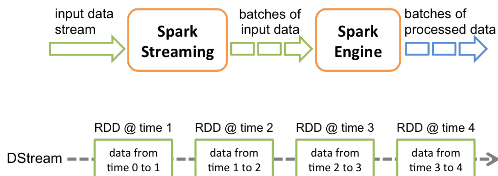
- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chop up the live stream into batches of X seconds.
  - Spark treats each batch of data as RDDs and processes them using RDD operations.
  - Finally, the processed results of the RDD operations are returned in batches.

Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.
    - Chop up the live stream into batches of X seconds.
    - Spark treats each batch of data as RDDs and processes them using RDD operations.
    - Finally, the processed results of the RDD operations are returned in batches.
    - Discretized Stream Processing (DStream)

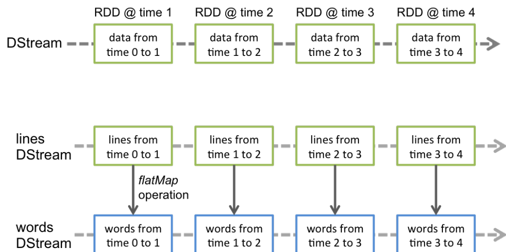
DStream

- DStream: sequence of RDDs representing a stream of data.
- Any operation applied on a DStream translates to operations on the underlying RDDs.



DStream

- DStream: sequence of RDDs representing a stream of data.
- Any operation applied on a DStream translates to operations on the underlying RDDs.



StreamingContext

- StreamingContext: the main entry point of all Spark Streaming functionality.
- To initialize a Spark Streaming program, a StreamingContext object has to be created.

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

Source of Streaming

- Two categories of streaming sources.
- Basic sources directly available in the StreamingContext API, e.g., file systems, socket connections, ...
- Advanced sources, e.g., Kafka, Flume, Kinesis, Twitter, ...

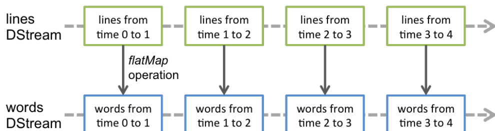
```
ssc.socketTextStream("localhost", 9999)

TwitterUtils.createStream(ssc, None)
```

DStream Transformations

- Transformations: modify data from on DStream to a new DStream.
- Standard RDD operations, e.g., map, join, ...
- DStream operations, e.g., window operations

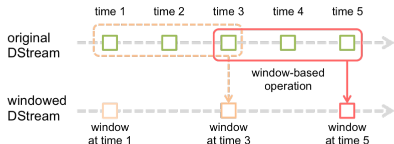
DStream Transformation Example

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

val lines = ssc.socketTextStream("localhost", 9999)

val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

wordCounts.print()
```



Window Operations

- Apply transformations over a sliding window of data: window length and slide interval.




```
val ssc = new StreamingContext(conf, Seconds(1))
val lines = ssc.socketTextStream(IP, Port)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))

val windowedWordCounts = pairs.reduceByKeyAndWindow(_ + _,
  Seconds(30), Seconds(10))
```

MapWithState Operation

- Maintains state while continuously updating it with new information.
- It requires the checkpoint directory.
- A new operation after `updateStateByKey`.

```
val ssc = new StreamingContext(conf, Seconds(1))
ssc.checkpoint(".")

val lines = ssc.socketTextStream(IP, Port)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))

val stateWordCount = pairs.mapWithState(
  StateSpec.function(mappingFunc))

val mappingFunc = (word: String, one: Option[Int], state: State[Int]) => {
  val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
  state.update(sum)
  (word, sum)
}
```

Transform Operation

- Allows arbitrary RDD-to-RDD functions to be applied on a DStream.
- Apply any RDD operation that is not exposed in the DStream API, e.g., joining every RDD in a DStream with another RDD.

```
// RDD containing spam information
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...)

val cleanedDStream = wordCounts.transform(rdd => {
 // join data stream with spam information to do data cleaning
  rdd.join(spamInfoRDD).filter(...)
  ...
})
```

Output Operations

- Push out DStream's data to external systems, e.g., a database or a file system.

- foreachRDD: the most generic output operator
  - Applies a function to each RDD generated from the stream.
  - The function is executed in the driver process.

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    val connection = createNewConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    connection.close()
  }
}
```

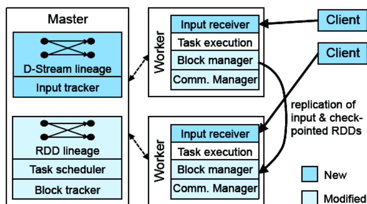
Spark Streaming and DataFrame

```
val words: DStream[String] = ...

words.foreachRDD { rdd =>
  // Get the singleton instance of SQLContext
  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  import sqlContext.implicits._

  // Convert RDD[String] to DataFrame
  val wordsDataFrame = rdd.toDF("word")

  // Register as table
  wordsDataFrame.registerTempTable("words")

  // Do word count on DataFrame using SQL and print it
  val wordCountsDataFrame =
    sqlContext.sql("select word, count(*) as total from words group by word")
  wordCountsDataFrame.show()
}
```

Implementation

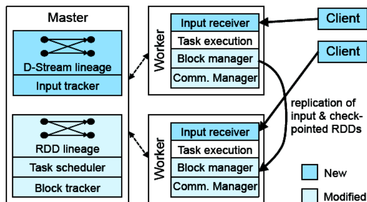
System Architecture

- Spark Streaming components:
  - Master: tracks the DStream lineage graph and schedules tasks to compute new RDD partitions.
  - Workers: receive data, store the partitions of input and computed RDDs, and execute tasks.
  - Client library: used to send data into the system.



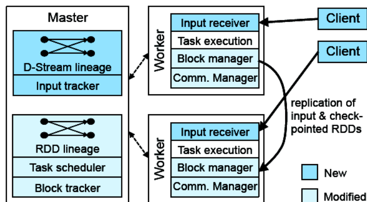
Application Execution (1/2)

- The system loads streams:
  - By receiving records directly from clients,
  - or by loading data periodically from an external storage, e.g., HDFS



Application Execution (1/2)

- The system loads streams:
  - By receiving records directly from clients,
  - or by loading data periodically from an external storage, e.g., HDFS
- All data is managed by a block store on each worker, with a tracker on the master to let nodes find the locations of blocks.
  - Each block is given a unique ID, and any node that has that ID can serve it.
  - The block store keeps new blocks in memory but drops them in an LRU fashion.



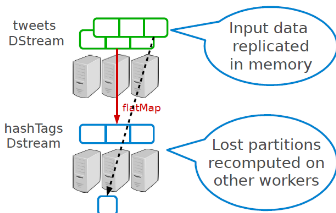
Application Execution (2/2)

- To decide when to start processing a new interval:
  - The nodes have their clocks synchronized via NTP.
  - Each node sends the master a list of block IDs it received in each interval when it ends.

- The master starts each task whenever its parents are finished.

Fault Tolerance

- Spark remembers the sequence of operations that creates each RDD from the original fault-tolerant input data (lineage graph).
- Batches of input data are replicated in memory of multiple worker nodes.
- Data lost due to worker failure, can be recomputed from input data.



Parallel Recovery

- When a node fails, the RDD partitions on the node and its running tasks are recomputed in parallel on other nodes.
- The system periodically checkpoints some of the RDDs, by asynchronously replicating them to other worker nodes.
- When a node fails, the system detects all missing RDD partitions and launches tasks to recompute them from the last checkpoint.
- Many tasks can be launched at the same time to compute different RDD partitions.

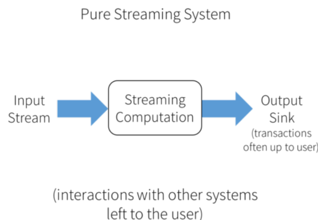
Master Recovery

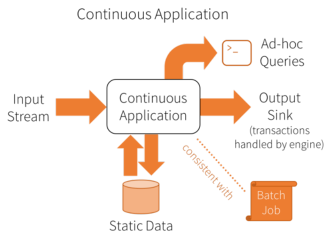
- To tolerate failures of Spark master:
  - Writing the state of the computation reliably when starting each timestep.
  - Having workers connect to a new master and report their RDD partitions to it when the old master fails.
- Operations are deterministic, therefore there is no problem if a given RDD is computed twice.

Structured Streaming

Motivation

- Continuous applications: end-to-end applications that react to data in real-time.
  - Updating data that will be served in real-time
  - Extract, transform and load (ETL)
  - Creating a real-time version of an existing batch job
  - Online machine learning



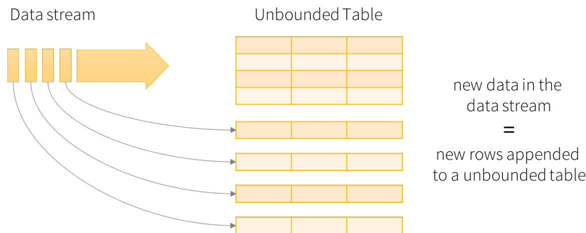


Structured Streaming

- Structured streaming is a new high-level API to support continuous applications.
- A higher-level API than Spark streaming.
- Built on the Spark SQL engine.
- Perform database-like query optimizations.

Programming Model (1/2)

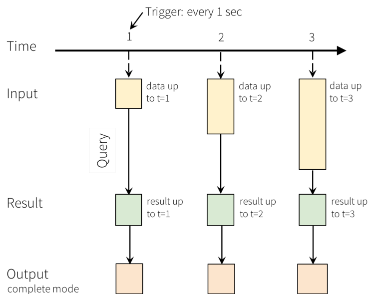
- Treating a live data stream as a table that is being continuously appended.
- Users can express their streaming computation as standard batch-like query as on a static table.
- Spark runs it as an incremental query on the unbounded input table.



Data stream as an unbounded table

Programming Model (2/2)

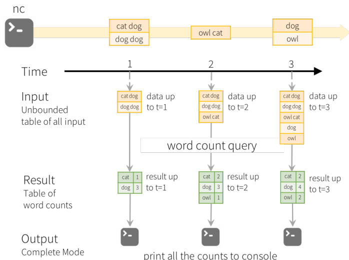
- A query on the input will generate the Result Table.
- Every trigger interval (e.g., every 1 second), new rows get appended to the Input Table, which eventually updates the Result Table.
- Whenever the result table gets updated, we can write the changed result rows to an external sink.



Programming Model for Structured Streaming

Example

```
val spark: SparkSession = ...

val lines = spark.readStream.format("socket").option("host", "localhost")
  .option("port", 9999).load()
val words = lines.as[String].flatMap(_.split(" "))
val wordCounts = words.groupBy("value").count()
val query = wordCounts.writeStream.outputMode("complete")
  .format("console").start()

query.awaitTermination()
```



Model of the Quick Example

Creating Streaming DataFrames and Datasets

- Creating through the `DataStreamReader` returned by `SparkSession.readStream()`.

```
val spark: SparkSession = ...

// Read text from socket
val socketDF = spark.readStream.format("socket")
    .option("host", "localhost").option("port", 9999).load()

// Read all the csv files written atomically in a directory
val userSchema = new StructType().add("name", "string").add("age", "integer")
val csvDF = spark.readStream.option("sep", ";")
    .schema(userSchema).csv("/path/to/directory")
```

Basic Operations

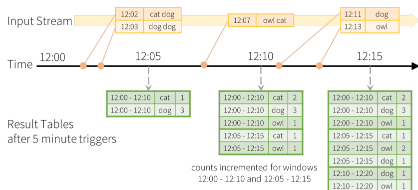
- Most of the common operations on DataFrame/Dataset are supported for streaming.

```
case class DeviceData(device: String, type: String, signal: Double,
  time: DateTime)

// streaming DataFrame with schema
// { device: string, type: string, signal: double, time: string }
val df: DataFrame = ...
val ds: Dataset[DeviceData] = df.as[DeviceData]

// Selection and projection
df.select("device").where("signal > 10") // using untyped APIs
ds.filter(_.signal > 10).map(_.device) // using typed APIs

// Aggregation
df.groupBy("type") // using untyped API
ds.groupByKey(_.type) // using typed API
```

Window Operation (1/2)

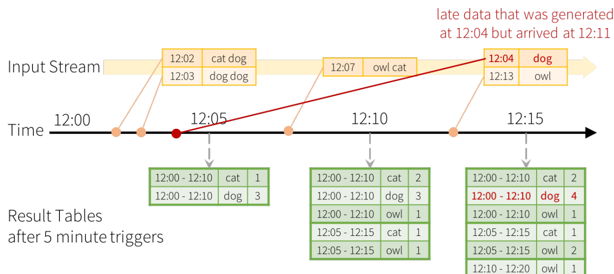
- Aggregations over a sliding event-time window.
- Event-time is the time embedded in the data itself, not the time Spark receives them.

```
// count words within 10 minute windows, updating every 5 minutes.
// streaming DataFrame of schema {timestamp: Timestamp, word: String}
val words = ...

val windowedCounts = words.groupBy(
  window($"timestamp", "10 minutes", "5 minutes"),
  $"word"
).count()
```



Window Operation (2/2)

- Late data



Flink Stream

Batch Processing vs. Stream Processing (1/2)

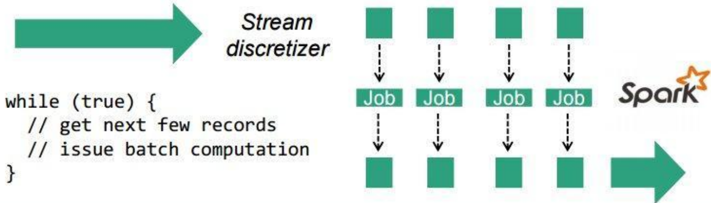
- Batch processing is just a special case of stream processing.

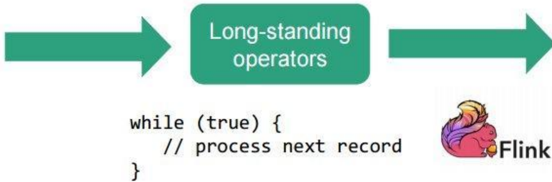
Batch Processing vs. Stream Processing (2/2)

- Batched/Stateless: scheduled in batches
  - Short-lived tasks (hadoop, spark)
  - Distributed streaming over batches (spark stream)

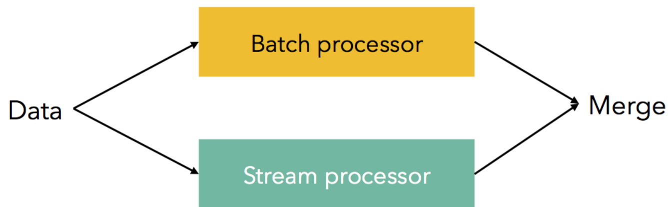
- DataFlow/Stateful: continuous/scheduled once (Storm, Samza, Naiad, Flink)
  - Long-lived task execution
  - State is kept inside tasks

Native vs. Non-Native Streaming



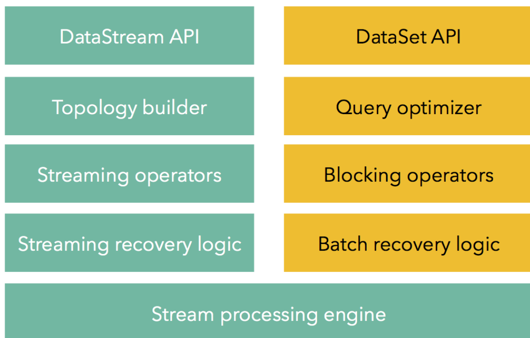


Lambda Architecture



Flink

- Distributed data flow processing system
- Unified real-time stream and batch processing



Programming Model

Programming Model

- Data stream
  - An unbounded, partitioned immutable sequence of events.

- Stream operators
  - Stream transformations that generate new output data streams from input ones.

Flink Stream API (1/2)

- Transformations:
  - Basic transformations: Map, Reduce, Filter, Aggregations
  - Binary stream transformations: CoMap, CoReduce
  - Windowing semantics: policy based flexible windowing (Time, Count, Delta ...)
  - Temporal binary stream operators: Joins, Crosses
  - Native support for iterations

Flink Stream API (2/2)

- Data stream sources
  - File system
  - Message queue connectors
  - Arbitrary source functionality

- Data stream outputs

Word Count in Flink - Batch and Stream

- Batch (DataSet API)

```
case class Word (word: String, frequency: Int)
val lines: DataSet[String] = env.readTextFile(...)

lines.flatMap {line => line.split(" ").map(word => Word(word, 1))}
  .groupBy("word").sum("frequency").print()
```

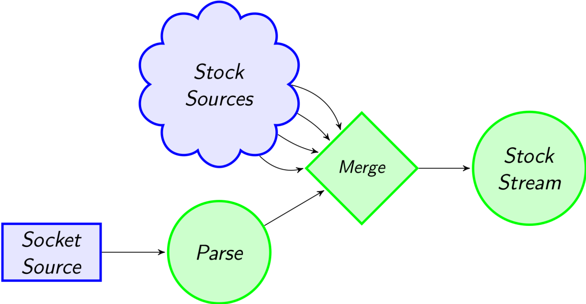
- Streaming (DataStream API)

```
case class Word (word: String, frequency: Int)
val lines: DataStream[String] = env.fromSocketStream(...)

lines.flatMap {line => line.split(" ").map(word => Word(word, 1))}
  .keyBy("word").window(Time.of(5, SECONDS))
  .every(Time.of(1, SECONDS)).sum("frequency").print()
```

Windowning Semantics

- Trigger and eviction policies
  - window(eviction, trigger)
  - window(eviction).every(trigger)
- Built-in policies:
  - Time: Time.of(length, TimeUnit/Custom timestamp)
  - Count: Count.of(windowSize)
  - Delta: Delta.of(treshold, Distance function, Start value)
- Window transformations:
  - Reduce, mapWindow

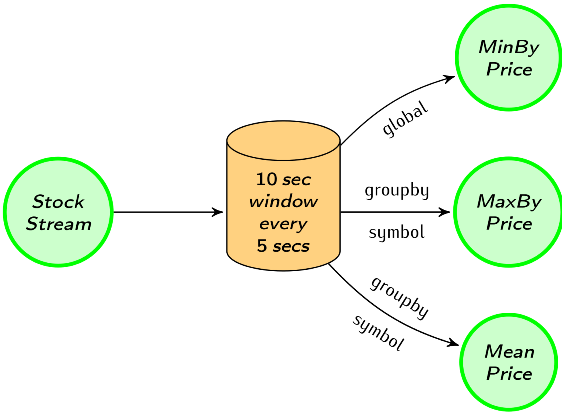
Example 1 - Reading From Multiple Inputs (1/2)



Example 1 - Reading From Multiple Inputs (2/2)

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

//Read from a socket stream at map it to StockPrice objects
val socketStockStream = env.socketTextStream("localhost", 9999).map(x => {
  val split = x.split(",")
  StockPrice(split(0), split(1).toDouble)
})

//Generate other stock streams
val SPX_Stream = env.addSource(generateStock("SPX")(10) _)
val FTSE_Stream = env.addSource(generateStock("FTSE")(20) _)
val DJI_Stream = env.addSource(generateStock("DJI")(30) _)
val BUX_Stream = env.addSource(generateStock("BUX")(40) _)

//Merge all stock streams together
val stockStream = socketStockStream.merge(SPX_Stream, FTSE_Stream,
  DJI_Stream, BUX_Stream)
```

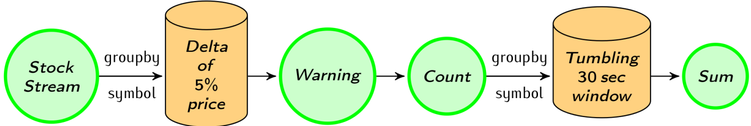
Example 2 - Window Aggregations (1/2)



Example 2 - Window Aggregations (2/2)

```
//Define the desired time window
val windowedStream = stockStream
  .window(Time.of(10, SECONDS)).every(Time.of(5, SECONDS))

//Compute some simple statistics on a rolling window
val lowest = windowedStream.minBy("price")
val maxByStock = windowedStream.groupBy("symbol").maxBy("price")
val rollingMean = windowedStream.groupBy("symbol").mapWindow(mean _)

//Compute the mean of a window
def mean(ts: Iterable[StockPrice], out: Collector[StockPrice]) = {
  if (ts.nonEmpty) {
    out.collect(StockPrice(ts.head.symbol,
      ts.foldLeft(0: Double)(_ + _.price) / ts.size))
  }
}
```

Example 3 - Data-Driven Windows (1/2)



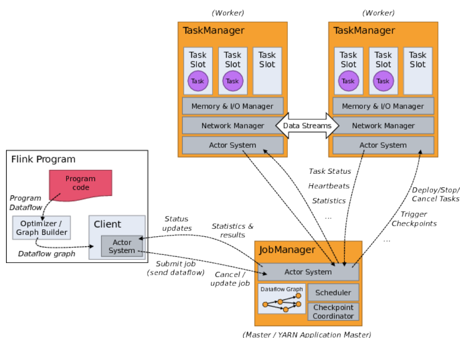
Example 3 - Data-Driven Windows (2/2)

```scala
case class Count(symbol: String, count: Int)
val defaultPrice = StockPrice("", 1000)

//Use delta policy to create price change warnings
val priceWarnings = stockStream.groupBy("symbol")
  .window(Delta.of(0.05, priceChange, defaultPrice))
  .mapWindow(sendWarning _)

//Count the number of warnings every half a minute
val warningsPerStock = priceWarnings.map(Count(_, 1))
  .groupBy("symbol")
  .window(Time.of(30, SECONDS))
  .sum("count")

def priceChange(p1: StockPrice, p2: StockPrice): Double = {
  Math.abs(p1.price / p2.price - 1)
}

def sendWarning(ts: Iterable[StockPrice], out: Collector[String]) = {
  if (ts.nonEmpty) out.collect(ts.head.symbol)
}
```

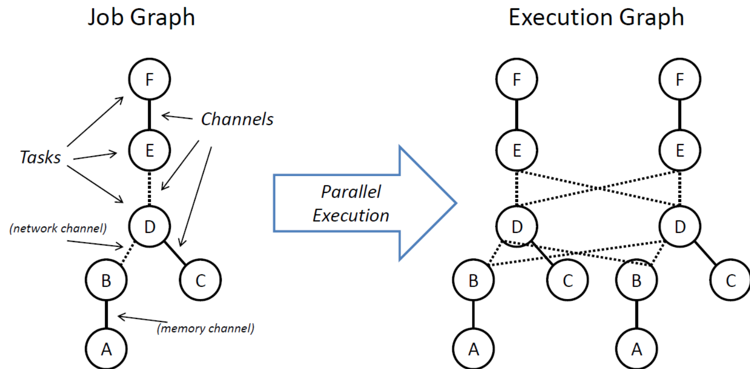
Implementation

Flink Architecture

- Master (JobManager): schedules tasks, coordinates checkpoints, coordinates recovery on failures, etc.
- Workers (TaskManagers): JVM processes that execute tasks of a dataflow, and buffer and exchange the data streams.
    - Workers use task slots to control the number of tasks it accepts.
    - Each task slot represents a fixed subset of resources of the worker.



Application Execution



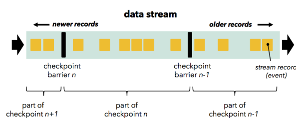
- Jobs are expressed as data flows.
- Job graphs are transformed into the execution graph.
- Execution graphs consist information to schedule and execute a job.

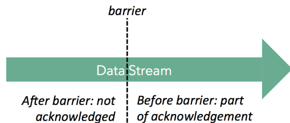
Fault Tolerance (1/3)

- Fault tolerance in Spark
  - RDD re-computation
- Fault tolerance in Storm
  - Tracks records with unique IDs.
  - Operators send acks when a record has been processed.
  - Records are dropped from the backup when the have been fully acknowledged.
- Fault tolerance in Flink
  - More coarse-grained approach than Storm.
  - Based on consistent global snapshots (inspired by Chandy-Lamport).
  - Low runtime overhead, stateful exactly-once semantics.

Fault Tolerance (2/3)

- Acks sequences of records instead of individual records.
- Periodically, the data sources inject checkpoint barriers into the data stream.
- The barriers flow through the data stream, and trigger operators to emit all records that depend only on records before the barrier.
- Once all sinks have received the barriers, Flink knows that all records before the barriers will never be needed again.

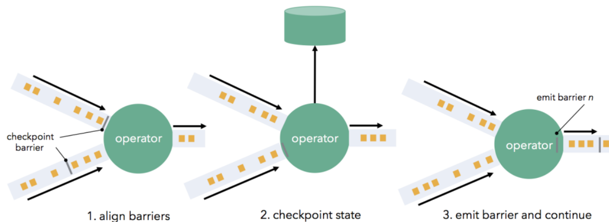




Fault Tolerance (3/3)

- Asynchronous barrier snapshotting for globally consistent checkpoints.

- Checkpointing and recovery.



Summary

Summary

- Spark
  - Mini-batch processing
  - DStream: sequence of RDDs
  - RDD and window operations
  - Structured streaming

- Flink
  - Unified batch and stream
  - Native streaming: data flow and pipelining
  - Different windowing semantics
  - Job graphs and execution graph
  - Asynchronous barriers

Questions?

Acknowledgements

Some slides and pictures were derived from Gyula Fora slides.