Scalable Stream Processing - Spark Streaming and Flink

Amir H. Payberah
payberah@kth.se
05/10/2018
The Course Web Page

https://id2221kth.github.io
Where Are We?

Data Processing
- Graph Data: Pregel, GraphLab, PowerGraph, GraphX, X-Streem, Chaos
- Structured Data: Spark SQL
- Machine Learning: MLLib, Tensorflow

Batch Data
- MapReduce, Dryad, FlumeJava, Spark

Streaming Data
- Storm, SEEP, Naiad, Spark Streaming, Flink, Millwheel, Google Dataflow

Data Storage
- Distributed File Systems: GFS, Flat FS
- NoSQL Databases: Dynamo, BigTable, Cassandra
- Distributed Messaging Systems: Kafka

Resource Management
- Mesos, YARN
Stream Processing Systems Design Issues

- Continuous vs. micro-batch processing
- Record-at-a-Time vs. declarative APIs
Outline

- Spark streaming
- Flink
Spark Streaming
Contribution

Design issues

- Continuous vs. micro-batch processing
- Record-at-a-Time vs. declarative APIs
Spark Streaming

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

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

- Chops up the live stream into batches of \(X\) seconds.
- Treats each batch as RDDs and processes them using RDD operations.
- Discretized Stream Processing (DStream)
**DStream (1/2)**

- **DStream**: sequence of RDDs representing a stream of data.
DStream (1/2)

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

- **StreamingContext** is the main entry point of all Spark Streaming functionality.

- The second parameter, **Seconds(1)**, represents the time interval at which streaming data will be divided into batches.

```scala
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```
StreamingContext is the main entry point of all Spark Streaming functionality.

The second parameter, `Seconds(1)`, represents the time interval at which streaming data will be divided into batches.

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

It can also be created from an existing `SparkContext` object.

```scala
val sc = ... // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```
DStream operations are broken into the following categories (rather than transformations and actions):

1. Input operations
2. Transformation
3. Output operations
Operations on DStreams

- Input operations
- Transformation
- Output operations
Every input DStream is associated with a Receiver object.

- It receives the data from a source and stores it in Spark’s memory for processing.
Every input DStream is associated with a Receiver object.

- It receives the data from a source and stores it in Spark’s memory for processing.

Three categories of streaming sources:

1. Basic sources directly available in the StreamingContext API, e.g., file systems, socket connections.
2. Advanced sources, e.g., Kafka, Flume, Kinesis, Twitter.
3. Custom sources, e.g., user-provided sources.
Input Operations - Basic Sources

- **Socket connection**
  - Creates a DStream from text data received over a **TCP socket connection**.

```scala
ssc.socketTextStream("localhost", 9999)
```
Input Operations - Basic Sources

- **Socket** connection
  - Creates a DStream from text data received over a TCP socket connection.
  
  ```scala
  ssc.socketTextStream("localhost", 9999)
  ```

- **File** stream
  - Reads data from files.
  
  ```scala
  streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)
  streamingContext.textFileStream(dataDirectory)
  ```
Input Operations - Advanced Sources

- Connectors with external sources
- Twitter, Kafka, Flume, Kinesis, ...

```java
TwitterUtils.createStream(ssc, None)
KafkaUtils.createStream(ssc, [ZK quorum], [consumer group id], [number of partitions])
```
To create a custom source: extend the `Receiver` class.

Implement `onStart()` and `onStop()`.

Call `store(data)` to store received data inside Spark.
class CustomReceiver(host: String, port: Int) extends Receiver[String](StorageLevel.MEMORY_AND_DISK_2) with Logging {
    def onStart() {
        new Thread("Socket Receiver") { override def run() { receive() } }.start()
    }

    def onStop() {}

    private def receive() {
        ... 
        socket = new Socket(host, port)
        val reader = ... // read from the socket connection
        val userInput = reader.readLine()
        while(!isStopped && userInput != null) {
            store(userInput)
            userInput = reader.readLine()
        }
        ...
    }
}
val customReceiverStream = ssc.receiverStream(new CustomReceiver(host, port))

val words = customReceiverStream.flatMap(_.split(" "))
Operations on DStreams

- Input operations
- Transformation
- Output operations
Transformations on DStreams are still lazy!

Now instead, computation is kicked off explicitly by a call to the `start()` method.

DStreams support many of the transformations available on normal Spark RDDs.
Transformations (2/4)

▶ **map**
  - Returns a new DStream by passing each element of the source DStream through a given function.
Transformations (2/4)

- **map**
  - Returns a new DStream by passing each element of the source DStream through a given function.

- **flatMap**
  - Similar to map, but each input item can be mapped to 0 or more output items.
Transformations (2/4)

- **map**
  - Returns a new DStream by passing each element of the source DStream through a given function.

- **flatMap**
  - Similar to **map**, but each input item can be mapped to 0 or more output items.

- **filter**
  - Returns a new DStream by selecting only the records of the source DStream on which func returns true.
Transformations (3/4)

- **count**
  - Returns a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
Transformations (3/4)

- **count**
  - Returns a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.

- **union**
  - Returns a new DStream that contains the union of the elements in two DStreams.
Transformations (4/4)

- **reduce**
  - Returns a new DStream of *single-element RDDs* by *aggregating* the elements in each RDD using a given function.
Transformations (4/4)

▶ **reduce**
  - Returns a new DStream of *single-element RDDs* by *aggregating* the elements in each RDD using a given function.

▶ **reduceByKey**
  - Returns a new DStream of *(K, V) pairs* where the values for each key are aggregated using the given reduce function.
Transformations (4/4)

- **reduce**
  - Returns a new DStream of single-element RDDs by aggregating the elements in each RDD using a given function.

- **reduceByKey**
  - Returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

- **countByValue**
  - Returns a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
Spark provides a set of transformations that apply to a over a **sliding window** of data.

- A window is defined by two parameters: **window length** and **slide interval**.
- A **tumbling window** effect can be achieved by making **slide interval = window length**.
Window Operations (2/3)

- `window(windowLength, slideInterval)`
  - Returns a new DStream which is computed based on windowed batches.
Window Operations (2/3)

- **window(windowLength, slideInterval)**
  - Returns a new DStream which is computed based on windowed batches.

- **countByWindow(windowLength, slideInterval)**
  - Returns a sliding window count of elements in the stream.
Window Operations (2/3)

- **window(windowLength, slideInterval)**
  - Returns a new DStream which is computed based on windowed batches.

- **countByWindow(windowLength, slideInterval)**
  - Returns a sliding window count of elements in the stream.

- **reduceByWindow(func, windowLength, slideInterval)**
  - Returns a new single-element DStream, created by aggregating elements in the stream over a sliding interval using func.
Window Operations (3/3)

- `reduceByKeyAndWindow(func, windowLength, slideInterval)`
  - Called on a DStream of `(K, V)` pairs.
  - Returns a new DStream of `(K, V)` pairs where the values for each key are aggregated using function `func` over **batches in a sliding window**.
reduceByKeyAndWindow(func, windowLength, slideInterval)
• Called on a DStream of (K, V) pairs.
• Returns a new DStream of (K, V) pairs where the values for each key are aggregated using function func over batches in a sliding window.

countByValueAndWindow(windowLength, slideInterval)
• Called on a DStream of (K, V) pairs.
• Returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window.
Join Operation (1/3)

- Stream-stream joins
  - In each batch interval, the RDD generated by \texttt{stream1} will be joined with the RDD generated by \texttt{stream2}.

```scala
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
val joinedStream = stream1.join(stream2)
```
Join Operation (2/3)

- Stream-stream joins
- Joins over windows of the streams.

```scala
val windowedStream1 = stream1.window(Seconds(20))
val windowedStream2 = stream2.window(Minutes(1))

val joinedStream = windowedStream1.join(windowedStream2)
```
Join Operation (3/3)

- **Stream-dataset** joins

```
val dataset: RDD[String, String] = ...
val windowedStream = stream.window(Seconds(20))...
val joinedStream = windowedStream.transform { rdd => rdd.join(dataset) }
```
Operations on DStreams

- Input operations
- Transformation
- 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**.
What’s wrong with this code?

dstream.foreachRDD { rdd =>
    val connection = createNewConnection() // executed at the driver
    rdd.foreach { record =>
        connection.send(record) // executed at the worker
    }
}
What’s wrong with this code?

This requires the **connection object** to be serialized and sent from the **driver to the worker**.

dstream.foreachRDD { rdd =>
  val connection = createNewConnection() // executed at the driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
}
What’s wrong with this code?

Creating a connection object has time and resource overheads.

Creating and destroying a connection object for each record can incur unnecessarily high overheads.

dstream.foreachRDD { rdd =>
  rdd.foreach { record =>
    val connection = createNewConnection()
    connection.send(record)
    connection.close()
  }
}
A better solution is to use `rdd.foreachPartition`

Create a single connection object and send all the records in a RDD partition using that connection.

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

Word Count in Spark Streaming
First we create a StreamingContext

```scala
import org.apache.spark._
import org.apache.spark.streaming._

// Create a local StreamingContext with two working threads and batch interval of 1 second.
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```
Create a DStream that represents streaming data from a TCP source.

Specified as hostname (e.g., localhost) and port (e.g., 9999).

```scala
val lines = ssc.socketTextStream("localhost", 9999)
```
Use `flatMap` on the stream to split the records text to words.

It creates a new DStream.

```scala
val words = lines.flatMap(_.split(" "))
```
Map the **words** DStream to a DStream of \((\text{word}, 1)\).

- Get the **frequency of words** in each **batch of data**.

- Finally, **print** the result.

```scala
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()
```
Start the computation and wait for it to terminate.

```java
// Start the computation
ssc.start()

// Wait for the computation to terminate
ssc.awaitTermination()
```
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()

ssc.start()
ssc.awaitTermination()
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 windowedWordCounts = pairs.reduceByKeyAndWindow(_ + _, Seconds(30), Seconds(10))
windowedWordCounts.print()

ssc.start()
ssc.awaitTermination()
State and DStream
What is State?

- Accumulate and aggregate the results from the start of the streaming job.

- Need to check the previous state of the RDD in order to do something with the current RDD.
What is State?

- Accumulate and aggregate the results from the start of the streaming job.

- Need to check the previous state of the RDD in order to do something with the current RDD.

- Spark supports stateful streams.
Checkpointing is a feature for any non-stateful transformation.

It is mandatory that you provide a checkpointing directory for stateful streams.

```scala
val ssc = new StreamingContext(conf, Seconds(1))
ssc.checkpoint("path/to/persistent/storage")
```
Stateful Stream Operations

Spark API proposes two functions for statful processing:

- `updateStateByKey`:
  - Executed on the whole range of keys in DStream.
  - Performance is proportional to the size of the state.

- `mapWithState`:
  - Executed only on a set of keys that are available in the last micro batch.
  - Performance is proportional to the size of the batch.
Stateful Stream Operations

Spark API proposes two functions for statful processing:

- **updateStateByKey**
  - It is executed on the whole range of keys in DStream.
  - The performance of these operation is proportional to the size of the state.

- **mapWithState**
  - It is executed only on set of keys that are available in the last micro batch.
  - The performance is proportional to the size of the batch.
Stateful Stream Operations

Spark API proposes two functions for statful processing:

- **updateStateByKey**
  - It is executed on the whole range of keys in DStream.
  - The performance of these operation is proportional to the size of the state.

- **mapWithState**
  - It is executed only on set of keys that are available in the last micro batch.
  - The performance is proportional to the size of the batch.
updateStateByKey Operation

- It manages the **state per key** (assuming we have **(key, value) pairs**).

```scala
def updateStateByKey[S](updateFunc: (Seq[V], Option[S]) => Option[S])
```

// Seq[V]: the list of new values received for the given key in the current batch
// Option[S]: the state we are updating on every iteration.
updateStateByKey Operation

- It manages the state per key (assuming we have (key, value) pairs).

```scala
def updateStateByKey[S](updateFunc: (Seq[V], Option[S]) => Option[S])
```

// Seq[V]: the list of new values received for the given key in the current batch
// Option[S]: the state we are updating on every iteration.

- To define `updateFunc` we have to figure out two things:
  1. Define the state
  2. Specify how to update the state using the previous state and the new values
Problems with updateStateByKey Operation

- **Performance**
  - For each new incoming batch, the transformation iterates the entire state store, regardless of whether a new value for a given key has been consumed or not.

- **No built-in timeouts**
  - Think what would happen in our example, if the event signaling the end of the user session was lost, or had not arrived for some reason.
mapWithState Operation

- **mapWithState** is an alternative to **updateStateByKey**:
  - Update function (**partial updates**)
  - Built in **timeout** mechanism
  - Choose the **return type**
  - Initial state

```scala
def mapWithState[StateType, MappedType](spec: StateSpec[K, V, StateType, MappedType]): DStream[MappedType]  
  StateSpec.function(updateFunc)  
  val updateFunc = (batch: Time, key: String, value: Option[Int], state: State[Int])
```
mapWithState Operation

- **mapWithState** is an alternative to **updateStateByKeys**:
  - Update function *(partial updates)*
  - Built in **timeout** mechanism
  - Choose the **return type**
  - Initial state

```scala
def mapWithState[StateType, MappedType](spec: StateSpec[K, V, StateType, MappedType]): DStream[MappedType]

StateSpec.function(updateFunc)
val updateFunc = (batch: Time, key: String, value: Option[Int], state: State[Int])
```

- You put all of the things into **StateSpec**.
```scala
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.updateStateByKey(updateFunc)

val updateFunc = (values: Seq[Int], state: Option[Int]) => {
  val newCount = values.foldLeft(0)(_ + _)
  val oldCount = state.getOrElse(0)
  val sum = newCount + oldCount
  Some(sum)
}
```
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(updateFunc))

val updateFunc = (key: String, value: Option[Int], state: State[Int]) => {
  val newCount = value.getOrElse(0)
  val oldCount = state.getOption.getOrElse(0)
  val sum = newCount + oldCount
  state.update(sum)
  (key, sum)
}
The first micro batch contains a message a.
The first micro batch contains a message a.

**updateStateByKey**
- `updateFunc = (values: Seq[Int], state: Option[Int]) => Some(sum)`
- Input: `values = [1]`, `state = None` (for key a)
- Output: `sum = 1` (for key a)
updateStateByKey vs. mapWithState Example (1/3)

- The first micro batch contains a message a.

- **updateStateByKey**
  - updateFunc = (values: Seq[Int], state: Option[Int]) => Some(sum)
  - Input: values = [1], state = None (for key a)
  - Output: sum = 1 (for key a)

- **mapWithState**
  - updateFunc = (key: String, value: Option[Int], state: State[Int]) => (key, sum)
  - Input: key = a, value = Some(1), state = 0
  - Output: key = a, sum = 1
updateStateByKey vs. mapWithState Example (2/3)

- The second micro batch contains messages \(a\) and \(b\).
updateStateByKey vs. mapWithState Example (2/3)

- The **second micro batch** contains messages **a** and **b**.

**updateStateByKey**
- `updateFunc = (values: Seq[Int], state: Option[Int]) => Some(sum)`
- **Input:** `values = [1]`, `state = Some(1)` (for key **a**)
- **Input:** `values = [1]`, `state = None` (for key **b**)
- **Output:** `sum = 2` (for key **a**)
- **Output:** `sum = 1` (for key **b**)

**mapWithState**
- `updateFunc = (key: String, value: Option[Int], state: State[Int]) => (key, sum)`
- **Input:** `key = a`, `value = Some(1)`, `state = 1`
- **Input:** `key = b`, `value = Some(1)`, `state = 0`
- **Output:** `key = a`, `sum = 2` (for key **a**)
- **Output:** `key = b`, `sum = 1` (for key **b**)

53 / 79
The second micro batch contains messages a and b.

**updateStateByKey**
- **updateFunc = (values: Seq[Int], state: Option[Int]) => Some(sum)**
- **Input:** values = [1], state = Some(1) (for key a)
- **Input:** values = [1], state = None (for key b)
- **Output:** sum = 2 (for key a)
- **Output:** sum = 1 (for key b)

**mapWithState**
- **updateFunc = (key: String, value: Option[Int], state: State[Int]) => (key, sum)**
- **Input:** key = a, value = Some(1), state = 1
- **Input:** key = b, value = Some(1), state = 0
- **Output:** key = a, sum = 2
- **Output:** key = b, sum = 1
updateStateByKey vs. mapWithState Example (3/3)

▶ The third micro batch contains a message \( b \).
▶ The third micro batch contains a message b.

▶ `updateStateByKey`

- `updateFunc = (values: Seq[Int], state: Option[Int]) => Some(sum)`
- Input: `values = []`, `state = Some(2)` (for key a)
- Input: `values = [1]`, `state = Some(1)` (for key b)
- Output: `sum = 2` (for key a)
- Output: `sum = 2` (for key b)
updateStateByKey vs. mapWithState Example (3/3)

- The third micro batch contains a message b.

- **updateStateByKey**
  - updateFunc = (values: Seq[Int], state: Option[Int]) => Some(sum)
  - Input: values = [], state = Some(2) (for key a)
  - Input: values = [1], state = Some(1) (for key b)
  - Output: sum = 2 (for key a)
  - Output: sum = 2 (for key b)

- **mapWithState**
  - updateFunc = (key: String, value: Option[Int], state: State[Int]) => (key, sum)
  - Input: key = b, value = Some(1), state = 1
  - Output: key = b, sum = 2
Structured Streaming
Structured Streaming

- Treating a live data stream as a table that is being continuously appended.
- Built on the Spark SQL engine.
- Perform database-like query optimizations.
Two main steps to develop a Spark structured streaming:
Two main steps to develop a Spark structured streaming:

1. Defines a query on the input table, as a static table.
   - Spark automatically converts this batch-like query to a streaming execution plan.
Two main steps to develop a Spark structured streaming:

1. Defines a query on the input table, as a static table.
   - Spark automatically converts this batch-like query to a streaming execution plan.

2. Specify triggers to control when to update the results.
   - Each time a trigger fires, Spark checks for new data (new row in the input table), and incrementally updates the result.
Programming Model (2/2)

User's batch-like query on input table

Incremental execution on streaming data

Input Table

User Query

Result Table

Spark SQL Planner

System Time

1

2

3

Input Table

Incremental Query

Result Table

Output Update Mode

Triggers

data up to t = 1

data up to t = 2

data up to t = 3

result up to t = 1

result up to t = 2

result up to t = 3

rows updated at t = 2

rows updated at t = 3
Output Modes

- Three output modes:

1. **Append**: only the new rows appended to the result table since the last trigger will be written to the external storage.
Output Modes

- **Three** output modes:

1. **Append**: only the new rows appended to the result table since the last trigger will be written to the external storage.

2. **Complete**: the entire updated result table will be written to external storage.
Output Modes

- **Three** output modes:

1. **Append**: only the new rows appended to the result table since the last trigger will be written to the external storage.

2. **Complete**: the entire updated result table will be written to external storage.

3. **Update**: only the rows that were updated in the result table since the last trigger will be changed in the external storage.
   - This mode works for output sinks that can be updated in place, such as a MySQL table.
Assume we receive \((id, time, action)\) events from a mobile app. We want to count how many actions of each type happened each hour. Store the result in MySQL.

We could express it as the following SQL query.

```sql
SELECT action, WINDOW(time, "1 hour"), COUNT * 
FROM events 
GROUP BY action, WINDOW(time, "1 hour")
```
val inputDF = spark.readStream.json("s3://logs")

inputDF.groupBy(col("action"), window(col("time"), "1 hour")).count().
  .writeStream.format("jdbc").start("jdbc:mysql://...")
Basic Operations

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

```scala
case class Call(action: String, time: Timestamp, id: Int)

val df: DataFrame = spark.readStream.json("s3://logs")
val ds: Dataset[Call] = df.as[Call]

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

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

// SQL commands
df.createOrReplaceTempView("dfView")
spark.sql("select count(*) from dfView") // returns another streaming DF
```
Window Operation

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

- Use `groupBy()` and `window()` to express **windowed aggregations**.

```scala
// count words within 10 minute windows, updating every 5 minutes.
// streaming DataFrame of schema {time: Timestamp, word: String}
val calls = ...
val actionHours = calls.groupBy(col("action"), window(col("time"), "1 hour", "5 minutes"))
```
Spark streaming uses **watermarks** to measure progress in **event time**.

Watermarks **flow as part of the data stream** and carry a **timestamp** $t$.

A $W(t)$ declares that **event time** has reached time $t$ in that stream
- There should be no more elements from the stream with a timestamp $t' \leq t$. 
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()
// count words within 10 minute windows, updating every 5 minutes.
// streaming DataFrame of schema {timestamp: Timestamp, word: String}
val words = ...
val windowedCounts = words.withWatermark("timestamp", "10 minutes")
  .groupBy(window(col("timestamp"), "10 minutes", "5 minutes"), col("word"))
  .count()
Flink
Flink

- **Distributed data flow** processing system
- **Unified real-time** stream and **batch** processing
- Process **unbounded** and **bounded** Data
- Design issues
  - Continuous vs. micro-batch processing
  - Record-at-a-Time vs. declarative APIs
Programs and Dataflows

```java
DataStream<String> lines = env.addSource(
    new FlinkKafkaConsumer<>(...));
DataStream<Event> events = lines.map((line) -> parse(line));
DataStream<Statistics> stats = events
    .keyBy("id")
    .timeWindow(Time.seconds(10))
    .apply(new MyWindowAggregationFunction());
stats.addSink(new RollingSink(path));
```
Window Operations

- A window defines a finite set of elements on an unbounded stream.

- Windows can be
  - Time window (e.g., every 30 seconds)
  - Count window (e.g., every 100 elements)

- One typically distinguishes different types of windows:
  - Tumbling windows (no overlap)
  - Sliding windows (with overlap)
  - Session windows (punctuated by a gap of inactivity)
Watermark and Late Elements

- It is possible that certain elements will violate the watermark condition.
  - After the $W(t)$ has occurred, more elements with timestamp $t' \leq t$ will occur.

- Streaming programs may explicitly expect some late elements.

```scala
val input: DataStream[T] = ...

input.keyBy(<key selector>)
  .window(<window assigner>)
  .allowedLateness(<time>)
  .<windowed transformation>(<window function>)
```
Fault Tolerance (1/2)

- Fault tolerance in **Spark**
  - RDD *re-computation*
Fault Tolerance (1/2)

▶ 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 they have been fully acknowledged.
Fault Tolerance (1/2)

- 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 they 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 (2/2)

- 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 (2/2)

- 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.
- Asynchronous barrier snapshotting for globally consistent checkpoints.
Summary

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

Flink
- Unified batch and stream
- Different windowing semantics
- Asynchronous barriers
Summary
References


▶ Some slides were derived from Heather Miller’s slides:
  http://heather.miller.am/teaching/cs4240/spring2018

▶ Structured Streaming In Apache Spark:
Questions?