



# Scalable Stream Processing - Spark Streaming and Beam

Amir H. Payberah  
payberah@kth.se  
2021-09-28



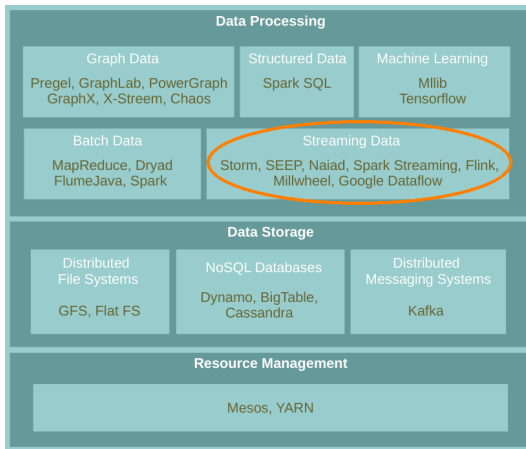


## The Course Web Page

<https://id2221kth.github.io>

<https://tinyurl.com/f6x544h>

# Where Are We?





# Stream Processing Systems Design Issues

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

# Spark Streaming



# Contribution

- ▶ Design issues
  - Continuous vs. **micro-batch processing**
  - Record-at-a-Time vs. **declarative APIs**

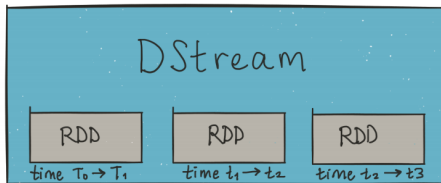
# Spark Streaming

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

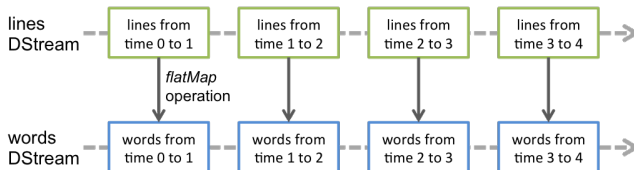
- **DStream**: sequence of **RDDs** representing a stream of data.





## DStream (2/2)

- 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.

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

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



# Input 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.
- ▶ **Basic sources** directly available in the **StreamingContext** API, e.g., **file systems**, **socket connections**.
- ▶ **Advanced sources**, e.g., **Kafka**, **Flume**, **Kinesis**, **Twitter**.



# Input Operations - Basic Sources

## ► Socket connection

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

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

## ► File stream

- Reads data from **files**.

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)
```

```
streamingContext.textFileStream(dataDirectory)
```



## Input Operations - Advanced Sources

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

```
TwitterUtils.createStream(ssc, None)
```

```
KafkaUtils.createStream(ssc, [ZK quorum], [consumer group id], [number of partitions])
```



## Transformations (1/2)

- ▶ Transformations on DStreams are still lazy!
- ▶ DStreams support many of the transformations available on normal Spark RDDs.
- ▶ Computation is kicked off explicitly by a call to the `start()` method.



## Transformations (2/2)

- ▶ **map**: a new **DStream** by passing each **element** of the source DStream through a given function.
- ▶ **reduce**: a new DStream of **single-element RDDs** by **aggregating** the elements in each RDD using a given function.
- ▶ **reduceByKey**: a new DStream of **(K, V) pairs** where the values for each key are **aggregated** using the given reduce function.



## Example - Word Count (1/6)

- First we create a `StreamingContext`

```
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))
```





## Example - Word Count (2/6)

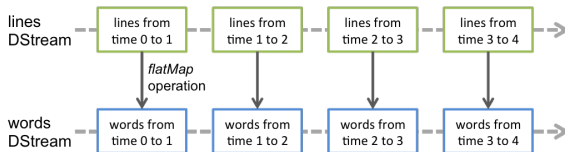
- ▶ Create a **DStream** that represents streaming data from a **TCP source**.
- ▶ Specified as **hostname** (e.g., localhost) and **port** (e.g., 9999).

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

## Example - Word Count (3/6)

- ▶ Use `flatMap` on the stream to split the records text to words.
- ▶ It creates a new DStream.

```
val words = lines.flatMap(_.split(" "))
```





## Example - Word Count (4/6)

- ▶ Map the **words** DStream to a DStream of **(word, 1)**.
- ▶ Get the **frequency of words** in each **batch of data**.
- ▶ Finally, **print** the result.

```
val pairs = words.map(word => (word, 1))  
val wordCounts = pairs.reduceByKey(_ + _)  
wordCounts.print()
```



## Example - Word Count (5/6)

- Start the **computation** and **wait** for it to **terminate**.

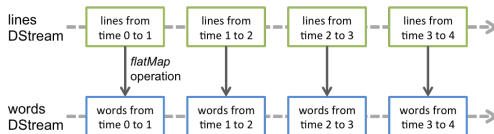
```
// Start the computation  
ssc.start()  
  
// Wait for the computation to terminate  
ssc.awaitTermination()
```

## Example - Word Count (6/6)

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

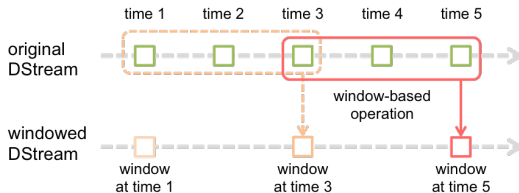
val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()

ssc.start()
ssc.awaitTermination()
```



## Window Operations (1/2)

- ▶ 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 (2/2)

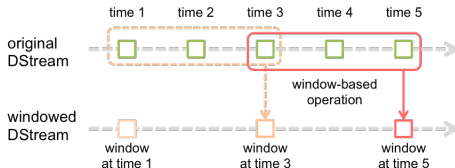
- ▶ `window(windowLength, slideInterval)`
  - Returns a new **DStream** which is computed based on **windowed batches**.
- ▶ `reduceByWindow(func, windowLength, slideInterval)`
  - Returns a new **single-element DStream**, created by aggregating elements in the stream over a **sliding interval** using `func`.
- ▶ `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**.

## Example - Word Count with Window

```
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()
```







## What about States?

- ▶ 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

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

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



# Stateful Stream Operations

## ► `mapWithState`

- It is executed only on set of keys that are available in the `last micro batch`.

```
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])
```

- ## ► Define the update function (`partial updates`) in `StateSpec`.



## Example - Stateful Word Count (1/4)

```
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)
}
```



## Example - Stateful Word Count (2/4)

- ▶ The first micro batch contains a message `a`.
- ▶ `updateFunc = (key: String, value: Option[Int], state: State[Int]) => (key, sum)`
- ▶ Input: `key = a, value = Some(1), state = 0`
- ▶ Output: `key = a, sum = 1`



## Example - Stateful Word Count (3/4)

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

## Example - Stateful Word Count (4/4)

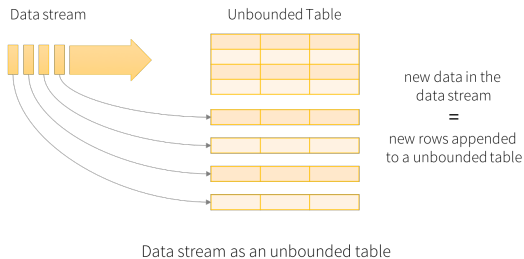
- ▶ The **third micro batch** contains a message **b**.
- ▶ `updateFunc = (key: String, value: Option[Int], state: State[Int]) => (key, sum)`
- ▶ Input: `key = b, value = Some(1), state = 1`
- ▶ Output: `key = b, sum = 2`

# Structured Streaming



# Structured Streaming

- Treating a **live data stream** as a **table** that is being **continuously appended**.

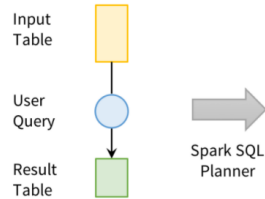




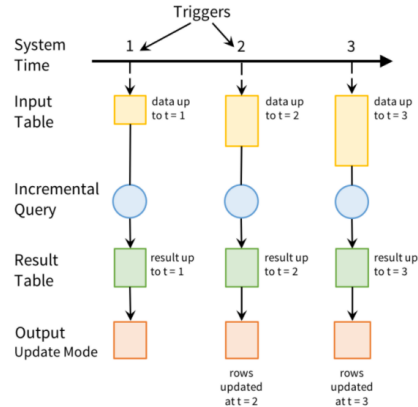
## Programming Model (1/2)

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

► 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.



## Five Steps to Define a Streaming Query (1/5)

- ▶ Define **input sources**.
- ▶ Use `spark.readStream` to create a `DataStreamReader`.

```
val spark = SparkSession...  
  
val lines = spark.readStream.format("socket")  
    .option("host", "localhost")  
    .option("port", 9999)  
    .load()
```



## Five Steps to Define a Streaming Query (2/5)

- ▶ Transform data.
- ▶ E.g., below `counts` is a `streaming DataFrame` that represents the running word counts.

```
import org.apache.spark.sql.functions._  
  
val words = lines.select(split(col("value"), "\\s").as("word"))  
  
val counts = words.groupBy("word").count()
```



## Five Steps to Define a Streaming Query (3/5)

- ▶ Define **output sink** and **output mode**.
- ▶ Use `DataFrame.writeStream` to define how to write the processed output data.

```
val writer = counts.writeStream.format("console").outputMode("complete")
```

## Five Steps to Define a Streaming Query (4/5)

- Specify processing details.

```
\\ word count details
import org.apache.spark.sql.streaming._

val checkpointDir = "...

val writer2 = writer
  .trigger(Trigger.ProcessingTime("1 second"))
  .option("checkpointLocation", checkpointDir)
```





## Five Steps to Define a Streaming Query (5/5)

- ▶ **Start** the query.
- ▶ `streamingQuery` represents an **active query** and can be used to **manage the query**.

```
val streamingQuery = writer2.start()
```

## Streaming Data Sources and Sinks - Files (1/2)

### ► Reading from files.

```
import org.apache.spark.sql.types._

val inputDirectoryOfJsonFiles = ...

val fileSchema = new StructType()
  .add("key", IntegerType)
  .add("value", IntegerType)

val inputDF = spark.readStream
  .format("json")
  .schema(fileSchema)
  .load(inputDirectoryOfJsonFiles)
```

## Streaming Data Sources and Sinks - Files (2/2)

### ► Writing to files.

```
val outputDir = ...  
val checkpointDir = ...  
val resultDF = ...  
  
val streamingQuery = resultDF  
  .writeStream  
  .format("parquet")  
  .option("path", outputDir)  
  .option("checkpointLocation", checkpointDir)  
  .start()
```



## Streaming Data Sources and Sinks - Kafka (1/2)

### ► Reading from Kafka.

```
val inputDF = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribe", "events")
  .load()
```

## Streaming Data Sources and Sinks - Kafka (2/2)

### ► Writing to Kafka.

```
val counts = ... // DataFrame[word: string, count: long]
val streamingQuery = counts
  .selectExpr("cast(word as string) as key", "cast(count as string) as value")
  .writeStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("topic", "wordCounts")
  .outputMode("update")
  .option("checkpointLocation", checkpointDir)
  .start()
```



## Basic Operations (1/2)

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

```
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
```



## Basic Operations (2/2)

### ► Aggregation

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

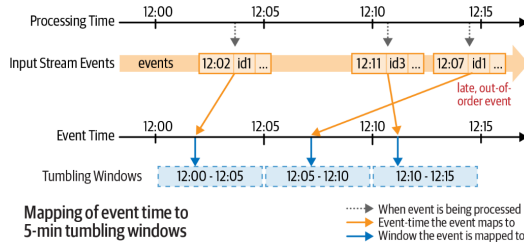
### ► SQL commands

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

# Window Operation (1/3)

- Aggregations over a sliding event-time window.
- E.g., below is expressing a five-minute count.

```
// The sensorReadings DataFrame has the generation timestamp as a column named eventTime
sensorReadings.groupBy("sensorId", window("eventTime", "5 minute")).count()
```



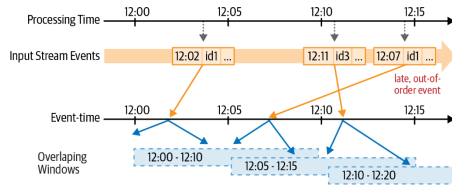


## Window Operation (2/3)

- ▶ Computing counts corresponding to 10-minute windows sliding every five minutes.

```
import org.apache.spark.sql.functions.*

sensorReadings.groupBy("sensorId", window("eventTime", "10 minute", "5 minute")).count()
```

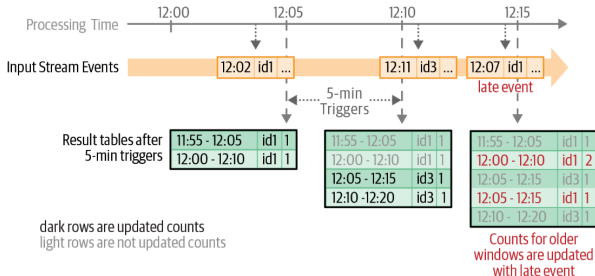


Mapping of event time to overlapping windows of length 10 mins and sliding interval 5 mins

.....▶ When event is being processed  
 —▶ Event-time the event maps to  
 —▶ Window the event is mapped to

## Window Operation (3/3)

- ▶ Assume that the input records are processed with a **trigger interval** of **five minutes**.
- ▶ The **state** of the **result table** at **each of the micro-batches** is shown in this figure.



# Handling Late Data

- ▶ A **watermark** is defined as a **moving threshold** in **event time** that trails behind the **maximum event time** seen by the query in the processed data.
- ▶ The **trailing gap** (**watermark delay**) defines **how long** the engine will **wait** for **late data** to arrive.

```
sensorReadings
  .withWatermark("eventTime", "10 minutes")
  .groupBy("sensorId", window("eventTime", "10 minutes", "5 minute"))
  .mean("value")
```

- Stateful processing using `groupByKey()` and `mapGroupsWithState()`.

```
def arbitraryStateUpdateFunction(  
  key: K,  
  newDataForKey: Iterator[V],  
  previousStateForKey: GroupState[S]  
): U  
  
val inputDataset: Dataset[V] = ...// input streaming Dataset  
  
inputDataset.groupByKey(keyFunction) // keyFunction() generates key from input  
  .mapGroupsWithState(arbitraryStateUpdateFunction)
```



## Stateful Operations - Example (1/3)

- ▶ Define the data types of **K**, **V**, **S**, and **U**.
- ▶ **Input data (V)**: `case class UserAction(userId: String, action: String)`
- ▶ **Keys (K)**: `String` (that is, the `userId`)
- ▶ **State (S)**: `case class UserStatus(userId: String, active: Boolean)`
- ▶ **Output (U)**: `UserStatus`, as we want to output the latest user status.

## Stateful Operations - Example (2/3)

- ▶ Define a **function** that is **called with new user actions**.
- ▶ **Two situations** we need to handle: whether a **previous state exists** for that key (i.e., `userId`) or **not**.
- ▶ Accordingly, we will **initialize the user's status**, or **update the existing status** with the new actions.

```
def updateUserStatus(  
  userId: String,  
  newActions: Iterator[UserAction],  
  state: GroupState[UserStatus]): UserStatus = {  
    val userStatus = state.getOption.getOrElse { new UserStatus(userId, false) }  
    newActions.foreach { action => userStatus.updateWith(action) }  
    state.update(userStatus)  
    return userStatus  
  }
```

## Stateful Operations - Example (3/3)

- ▶ Apply the function on the actions.
- ▶ We group the input actions `Dataset` using `groupByKey()` and then apply the `updateUserStatus` function using `mapGroupsWithState()`.

```
val userActions: Dataset[UserAction] = ...  
val latestStatuses = userActions  
  .groupByKey(userAction => userAction.userId)  
  .mapGroupsWithState(updateUserStatus _)
```

# Google Dataflow and Beam

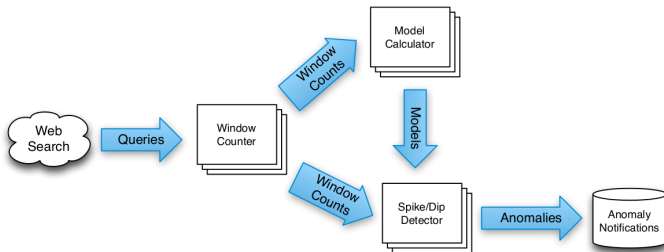


- ▶ Google's **Zeitgeist**: tracking trends in web queries.
- ▶ Builds a **historical model** of each query.
- ▶ Google discontinued Zeitgeist, but most of its features can be found in **Google Trends**.



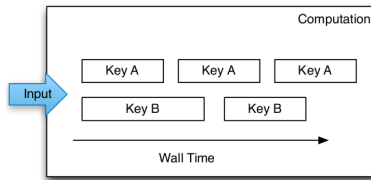
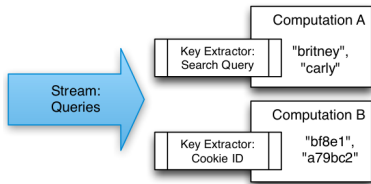
# MillWheel Dataflow

- ▶ **MillWheel** is a framework for building **low-latency** data-processing applications.
- ▶ A **dataflow graph** of **transformations** (**computations**).
- ▶ **Stream**: **unbounded data** of (**key, value, timestamp**) records.
  - Timestamp: **event-time**



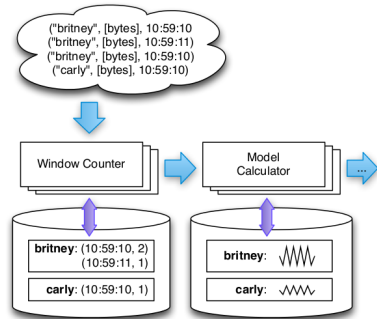
# Key Extraction Function and Computations

- ▶ Stream of (key, value, timestamp) records.
- ▶ **Key extraction function**: specified by the stream consumer to **assign keys** to records.
- ▶ **Computation** can only access state for the **specific key**.
- ▶ **Multiple** computations can extract **different keys** from the **same stream**.



# Persistent State

- ▶ Keep the **states** of the computations
- ▶ Managed on **per-key** basis
- ▶ Stored in **Bigtable** or **Spanner**
- ▶ Common use: **aggregation**, **joins**, ...





## Delivery Guarantees

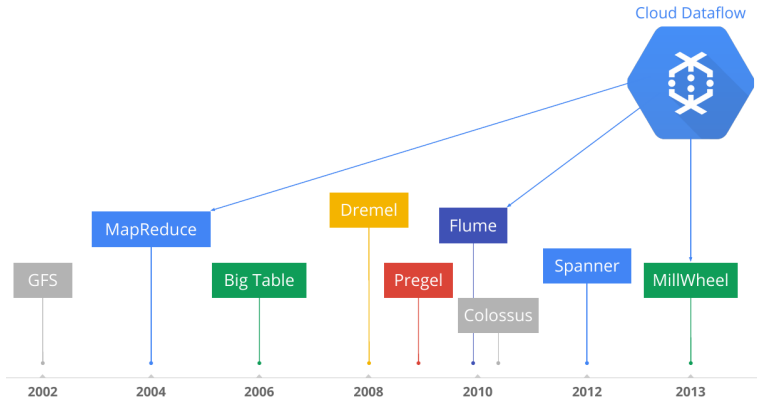
- ▶ Emitted records are **checkpointed** before **delivery**.
  - The **checkpoints** allow **fault-tolerance**.
- ▶ When a delivery is **ACKed** the checkpoints can be **garbage collected**.
- ▶ If an ACK is **not received**, the record can be **re-sent**.
- ▶ **Exactly-one** delivery: **duplicates are discarded** by MillWheel at the recipient.

# What is Google Cloud Dataflow?



# Google Cloud Dataflow (1/2)

- Google managed service for unified **batch** and **stream** data processing.





## Google Cloud Dataflow (2/2)

- ▶ Open source **Cloud Dataflow SDK**
- ▶ Express your data processing **pipeline** using **FlumeJava**.
- ▶ If you run it in **batch** mode, it executed on the **MapReduce** framework.
- ▶ If you run it in **streaming** mode, it is executed on the **MillWheel** framework.



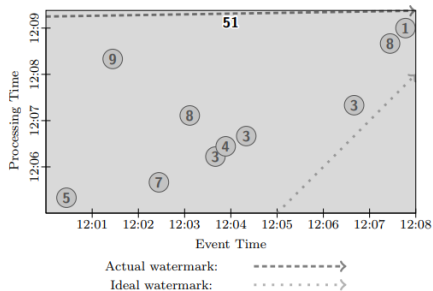
- 
- Complete pipeline online concurrently
- source      tokenizer      window count

# Windowing and Triggering

- ▶ **Windowing** determines **where** in **event time** data are grouped together for processing.
  - **Fixed time** windows (tumbling windows)
  - **Sliding time** windows
  - **Session** windows
- ▶ **Triggering** determines **when** in **processing time** the results of groupings are emitted as panes.
  - **Time-based** triggers
  - **Data-driven** triggers
  - **Composit** triggers

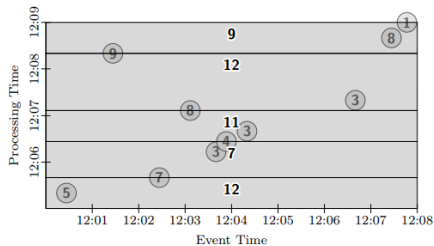
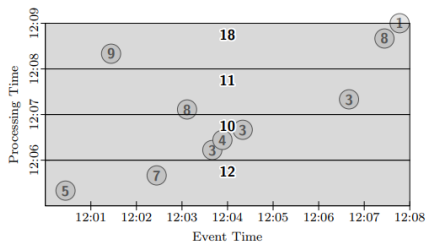
## Example (1/3)

### ► Batch processing



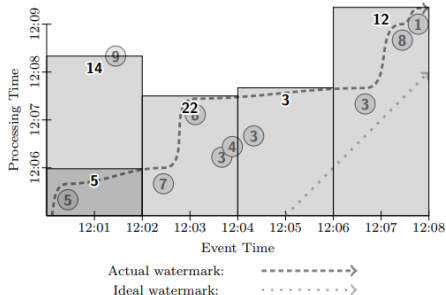
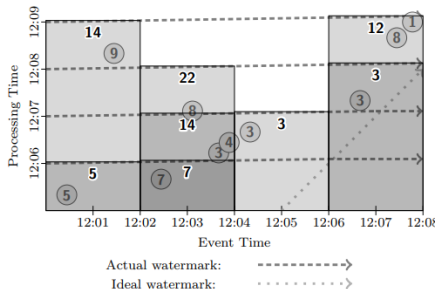
## Example (2/3)

- ▶ Trigger at **period** (time-based triggers)
- ▶ Trigger at **count** (data-driven triggers)



## Example (3/3)

- ▶ Fixed window, trigger at **period** (**micro-batch**)
- ▶ Fixed window, trigger at **watermark** (**streaming**)



# Where is Apache Beam?



# From Google Cloud Dataflow to Apache Beam

- ▶ In 2016, [Google Cloud Dataflow](#) team announced its intention to donate the [programming model](#) and [SDKs](#) to the Apache Software Foundation.
- ▶ That resulted in the incubating project [Apache Beam](#).





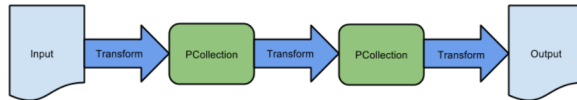
# Programming Components

- ▶ Pipelines
- ▶ PCollections
- ▶ Transforms
- ▶ I/O sources and sinks



## Pipelines (1/2)

- ▶ A **pipeline** represents a **data processing job**.
- ▶ **Directed graph** of operating on data.
- ▶ A pipeline consists of **two** parts:
  - **Data** (**PCollection**)
  - **Transforms** applied to that data

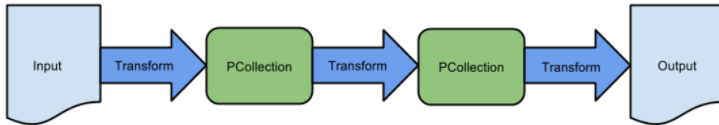


## Pipelines (2/2)

```
public static void main(String[] args) {  
  
    // Create a pipeline  
    PipelineOptions options = PipelineOptionsFactory.create();  
    Pipeline p = Pipeline.create(options);  
  
    p.apply(TextIO.Read.from("gs://..."))    // Read input.  
      .apply(new CountWords())              // Do some processing.  
      .apply(TextIO.Write.to("gs://..."));  // Write output.  
  
    // Run the pipeline.  
    p.run();  
}
```

## PCollections (1/2)

- ▶ A **parallel collection** of records
- ▶ **Immutable**
- ▶ Must specify **bounded** or **unbounded**





## PCollections (2/2)

```
// Create a Java Collection, in this case a List of Strings.
static final List<String> LINES = Arrays.asList("line 1", "line 2", "line 3");

PipelineOptions options = PipelineOptionsFactory.create();
Pipeline p = Pipeline.create(options);

// Create the PCollection
p.apply(Create.of(LINES)).setCoder(StringUtf8Coder.of())
```

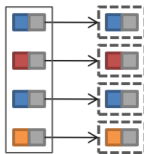


# Transformations

- ▶ A **processing operation** that transforms data
- ▶ Each transform accepts **one (or multiple)** **PCollections** as input, performs an operation, and produces **one (or multiple)** new **PCollections** as output.
- ▶ Core transforms: **ParDo**, **GroupByKey**, **Combine**, **Flatten**

## Transformations - ParDo

- Processes each element of a **PCollection** independently using a **user-provided DoFn**.



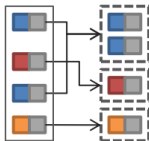
```
// The input PCollection of Strings.
PCollection<String> words = ...;

// The DoFn to perform on each element in the input PCollection.
static class ComputeWordLengthFn extends DoFn<String, Integer> { ... }

// Apply a ParDo to the PCollection "words" to compute lengths for each word.
PCollection<Integer> wordLengths = words.apply(ParDo.of(new ComputeWordLengthFn()));
```

## Transformations - GroupByKey

- Takes a `PCollection` of key-value pairs and **gathers up all values with the same key**.



```
// A PCollection of key/value pairs: words and line numbers.
PCollection<KV<String, Integer>> wordsAndLines = ...;

// Apply a GroupByKey transform to the PCollection "wordsAndLines".
PCollection<KV<String, Iterable<Integer>>> groupedWords = wordsAndLines.apply(
    GroupByKey.<String, Integer>create());
```

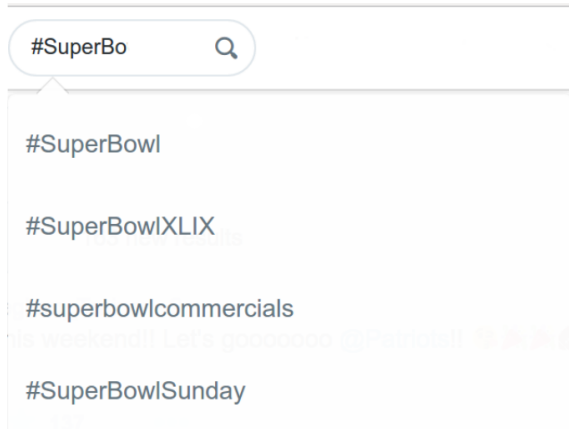
## Transformations - Join and CoGroupByKey

- Groups together the **values** from **multiple PCollections** of key-value pairs.

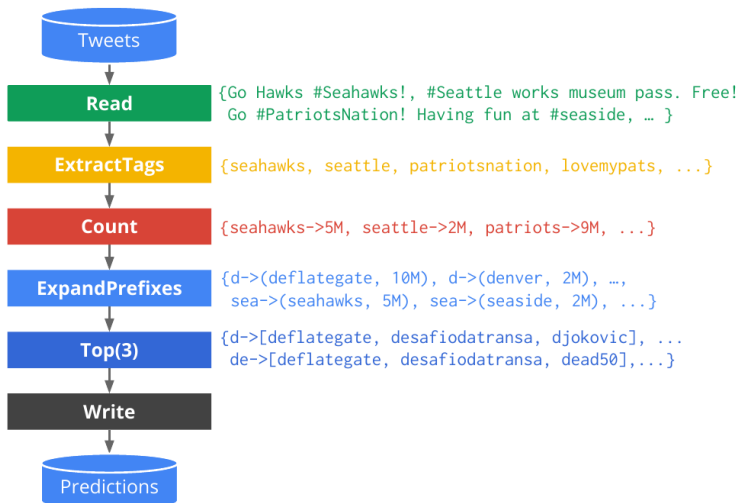
```
// Each data set is represented by key-value pairs in separate PCollections.  
// Both data sets share a common key type ("K").  
PCollection<KV<K, V1>> pc1 = ...;  
PCollection<KV<K, V2>> pc2 = ...;  
  
// Create tuple tags for the value types in each collection.  
final TupleTag<V1> tag1 = new TupleTag<V1>();  
final TupleTag<V2> tag2 = new TupleTag<V2>();  
  
// Merge collection values into a CoGbkResult collection.  
PCollection<KV<K, CoGbkResult>> coGbkResultCollection =  
    KeyedPCollectionTuple.of(tag1, pc1)  
        .and(tag2, pc2)  
        .apply(CoGroupByKey.<K>create());
```



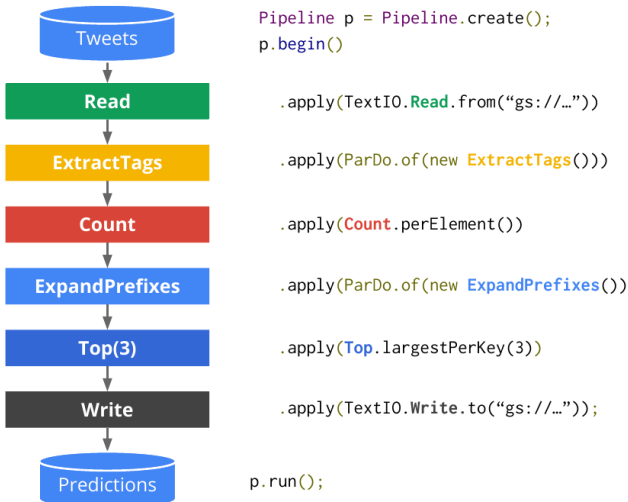
## Example: HashTag Autocompletion (1/3)



## Example: HashTag Autocompletion (2/3)



## Example: HashTag Autocompletion (3/3)



# Summary



# Summary

## ▶ Spark

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

## ▶ Google cloud dataflow

- Pipeline
- PCollection: windows and triggers
- Transforms

- ▶ M. Zaharia et al., “Spark: The Definitive Guide”, O’Reilly Media, 2018 - Chapters 20-23.
- ▶ M. Zaharia et al., “Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters”, HotCloud’12.
- ▶ T. Akidau et al., “MillWheel: fault-tolerant stream processing at internet scale”, VLDB 2013.
- ▶ T. Akidau et al., “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing”, VLDB 2015.
- ▶ The world beyond batch: Streaming 102  
<https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102>

Questions?