



Scalable Stream Processing - Spark Streaming and Beam

Amir H. Payberah
payberah@kth.se
2020-09-22



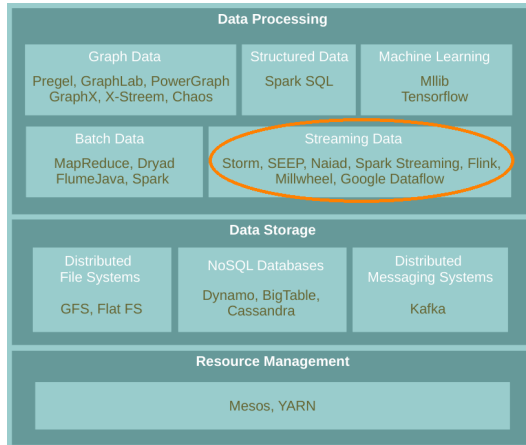


The Course Web Page

`https://id2221kth.github.io`

`https://tinyurl.com/y4qph82u`

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**

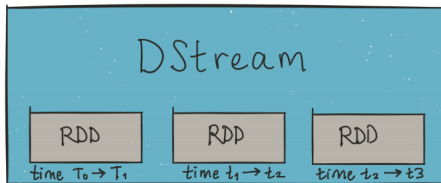
Spark Streaming

- ▶ 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**)



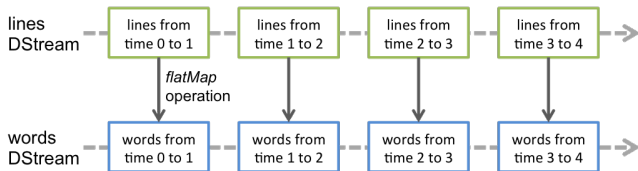
DStream (1/2)

- **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)

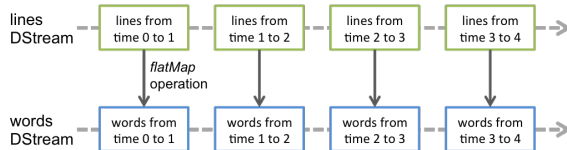
- ▶ Create a `DStream` that represents streaming data from a `TCP` source.
- ▶ Specified as `hostname` (e.g., `localhost`) and `port` (e.g., `9999`).

```
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)

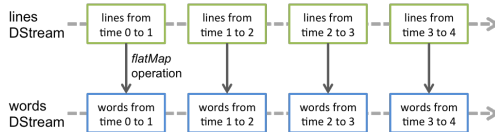
```

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()

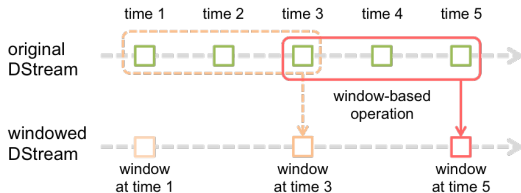
ssc.start()
ssc.awaitTermination()

```



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**.

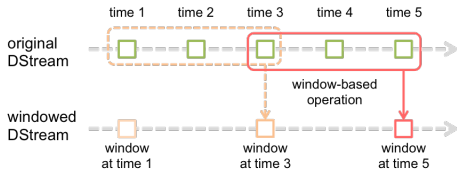


Example - Word Count with Window

```
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`



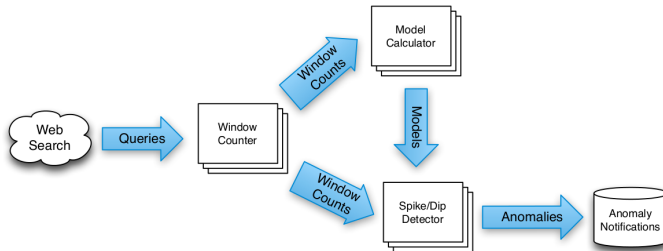
Google Dataflow and Beam

- ▶ 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**.



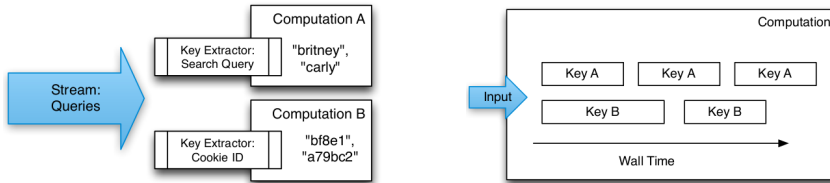
MillWheel Dataflow

- ▶ **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**



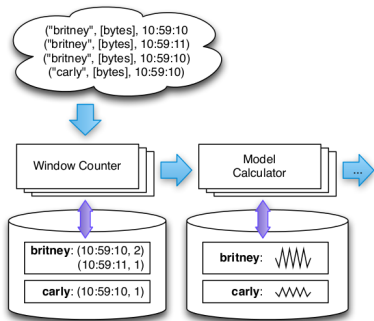
Key Extraction Function and Computations

- ▶ 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

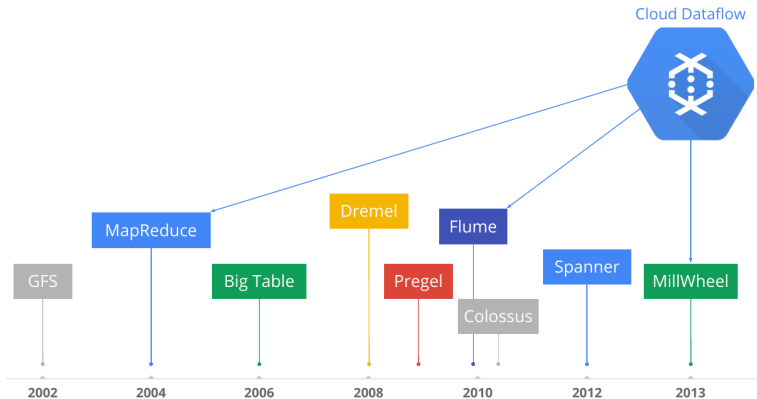
- ▶ Emitted records are **checkpointed** before **delivery**.
 - The **checkpoints** allow **fault-tolerance**.
- ▶ When a delivery is **ACKed** the checkpoints can be **garbage collected**.
- ▶ If an ACK is **not received**, the record can be **re-sent**.
- ▶ **Exactly-one** delivery: **duplicates are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



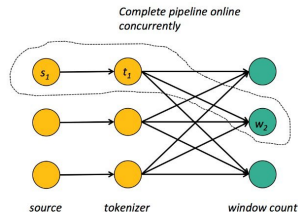


Google Cloud Dataflow (2/2)

- ▶ 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.

Programming Model

- ▶ Pipeline, a **directed graph** of data processing transformations
- ▶ **Optimized** and executed as a unit
- ▶ May include multiple **inputs** and multiple **outputs**
- ▶ May encompass many logical **MapReduce** or **Millwheel** operations



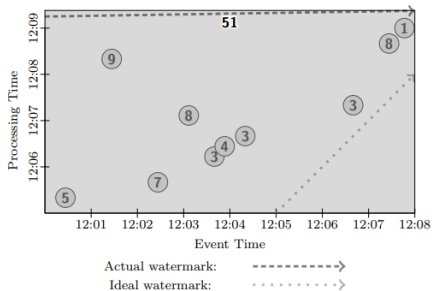


Windowing and Triggering

- ▶ **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

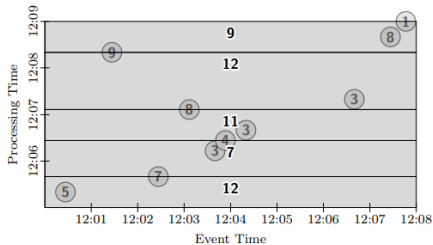
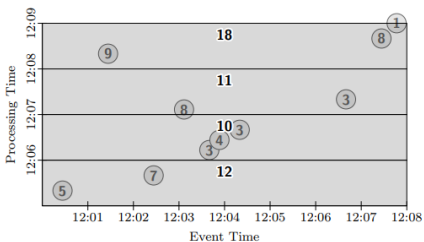
Example (1/3)

- ▶ Batch processing



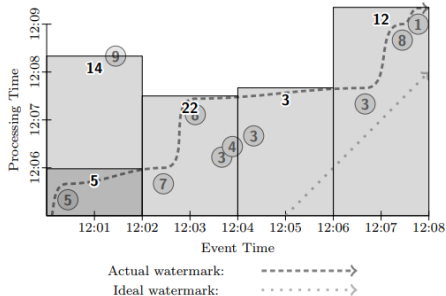
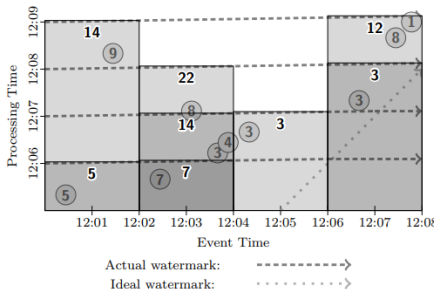
Example (2/3)

- ▶ 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](#).



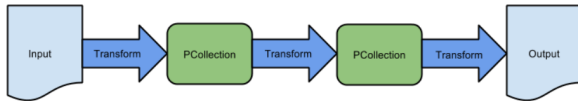


Programming Components

- ▶ 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



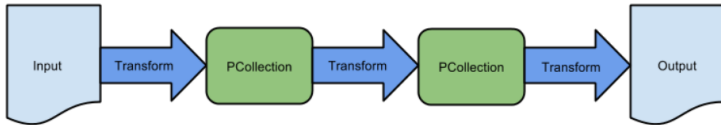


Pipelines (2/2)

```
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())
```

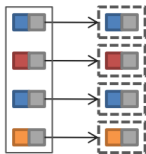


Transformations

- ▶ 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

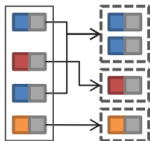
- Processes each element of a `PCollection` independently using a **user-provided DoFn**.



```
// 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.

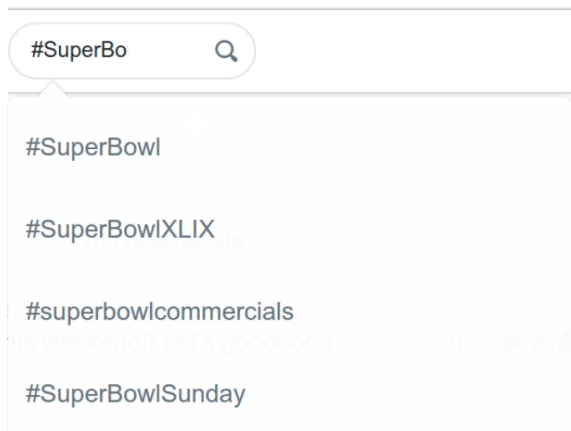
```
// Each data set is represented by key-value pairs in separate PCollections.
// Both data sets share a common key type ("K").
PCollection<KV<K, V1>> pc1 = ...;
PCollection<KV<K, V2>> pc2 = ...;

// 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)



```
Pipeline p = Pipeline.create();  
p.begin();
```

```
.apply(TextIO.Read.from("gs://..."))
```

```
.apply(ParDo.of(new ExtractTags()))
```

```
.apply(Count.perElement())
```

```
.apply(ParDo.of(new ExpandPrefixes()))
```

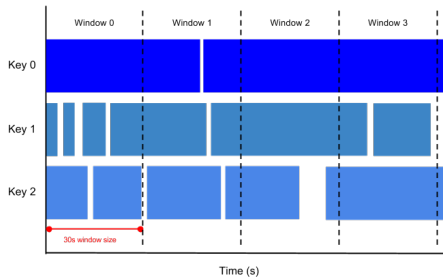
```
.apply(Top.largestPerKey(3))
```

```
.apply(TextIO.Write.to("gs://..."));
```

```
p.run();
```

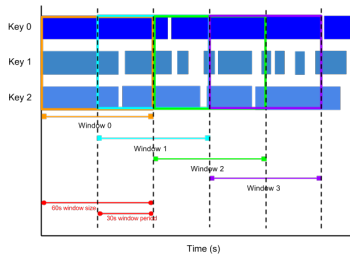
Windowing (1/2)

► Fixed time windows



```
PCollection<String> items = ...;  
  
PCollection<String> fixedWindowedItems = items.apply(  
    Window.<String>into(FixedWindows.of(Duration.standardSeconds(30))));
```

► Sliding time windows



```
PCollection<String> items = ...;
```

```
PCollection<String> slidingWindowedItems = items.apply(
    Window.<String>into(SlidingWindows.of(Duration.standardSeconds(60))
        .every(Duration.standardSeconds(30))));
```



Triggering

- ▶ E.g., emits results one minute after the first element in that window has been processed.

```
PCollection<String> items = ...;

items.apply(
  Window.<String>into(FixedWindows
    .of(1, TimeUnit.MINUTES))
    .triggering(AfterProcessingTime.pastFirstElementInPane()
      .plusDelayOf(Duration.standardMinutes(1)));
```

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



References

- ▶ 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?