Spark Streaming and Apache Beam

Objectives

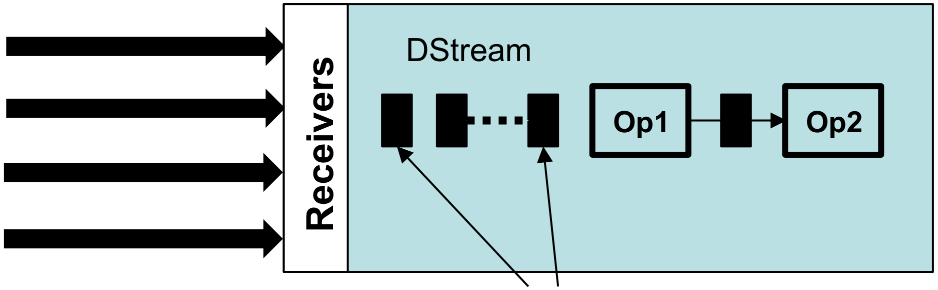
- Apache Spark Streaming
  – Programming

- Apache Beam Hands On
  – Programming

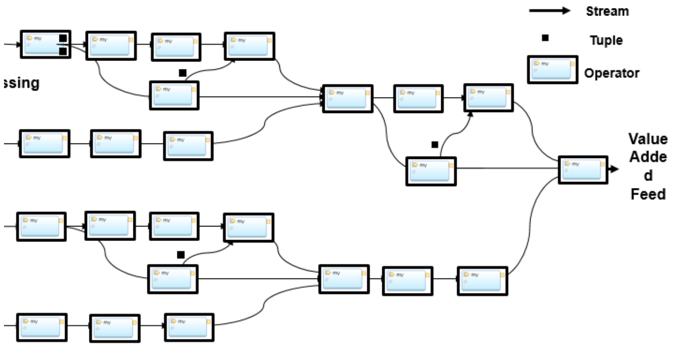
- HW1

Spark Streaming

- Introduce concept `Dstream` (discretized streams)
  - Sequence of RDDs arriving over time
  - Dstreams can be created from different data sources
    - Network interfaces, Flame, Kafka, Flume, File System etc.
  - Support transformations and actions (output operations)



Batches of input streaming
data grouped into tuples



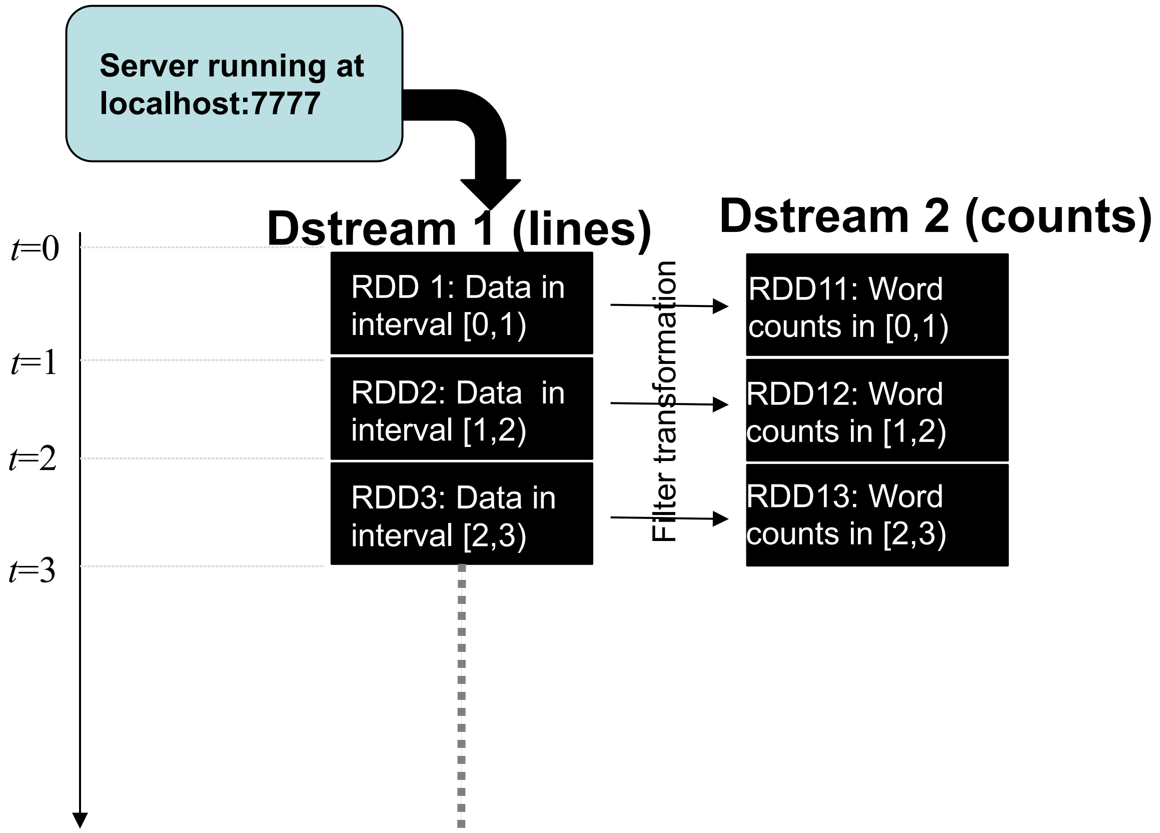
Streaming Example

- Example of constructing Dstream from network socket
  - Setup a Dstream

```
sc = SparkContext(appName="StreamingNetworkWordCount")
ssc = StreamingContext(sc, 1)
lines = ssc.socketTextStream(IP, Port)
counts = lines.flatMap(lambda line: line.split(" "))\
              .map(lambda word: (word, 1))\
              .reduceByKey(lambda a, b: a+b)
counts.pprint()
ssc.start()
ssc.awaitTermination
```

  - Take all data received in one second
  - Break it into words and then do a wordcount
  - Print all results to screen

Streaming Example



The operation of this word count is "stateless" – each window handled independently

Stateless Streaming Transformations

Name	Purpose	Example
`map()`	Apply function defined inline to each tuple (RDD) in the stream	`ds.map(lambda x: x+1)`
`flatMap()`	Apply inline function to each tuple in stream and create tuple with more than one element per input element	`ds.flatMap(lambda x: x.split(" "))`
`filter()`	Filter elements in input tuple to create new tuple	`ds.filter(lambda x: x.contains("error"))`
`reduceByKey()`	Perform reduce by key within tuple to create new tuple	`ds.reduceByKey( lambda (x,y): x+y)`
`groupByKey()`	Group values by key within each tuple	`ds.groupByKey()`
`transform()`	Apply any RDD to RDD function on each tuple	`ds.transform(lambda rdd: myFunc(rdd))`

Stateless Two-Stream Transformations

- Stateless Join on two Streams
  - Perform a standard RDD join on two tuples, one from each stream
  - Fast stream needs to wait for slow stream

```
//Create a streaming context with a 1 second batch
sc = SparkContext(appName="StreamingStatelessJoin")
ssc = StreamingContext(sc, 1)
accessLogs = ssc.socketTextStream(IP, Port)
ipDstream = accessLogs.map(lambda x: (x.getIpAddress(),1))
ipCountDstream = ipDstream.reduceByKey(lambda (x,y): x+y)

ipByteDstream = accessLogs.map(lambda x: (x.getIpAddress(),x.getContentSize())
ipByteCountDstream = ipByteDstream.reduceByKey(lambda (x,y): (x+y))

ipBCDstream = ipByteCountDstream.join(ipCountDstream)
ssc.start()
ssc.awaitTermination()
```

This can cause memory issues if one stream is much slower than the other

Also – limiting in terms of match conditions, given data arrival order etc,

Stateful Transformation on Streams

- Create and maintain state as tuples processed
- Such state, along with internal algorithm, affects the results
- e.g. `DeDuplicate`
  - *tuple is considered a duplicate if it shares the same key with a previously seen tuple within a pre-defined period of time*
- Runtime support is challenging
  - Require synchronization in a multi-threaded context
  - No trivial way to parallelize
  - Require some persistence mechanism for fault-tolerance

Aside: Data Time versus Wall Clock Time

- Both are representations of time
  - With different references
- Data Time
  - Event Time, data timestamp
- Wall Clock Time
  - Arrival time, Processing time

Windows and Stateful Processing

- Sorting, aggregating, or joining data in a relational table
  - All data in the table can be processed
- For streaming data, data flows continuously
  - No beginning, no end
  - Requires a different paradigm for sorting, aggregating and joining
  - Can only work with a subset of consecutive tuples
- This finite set of tuples is called a window
  - Note that in Spark – each tuple is a RDD (collection of elements)

Window Properties

- Three properties that define a window
  - Eviction policy
    - Defines how large a window can get
    - Determines which older tuples are removed from the window
  - Trigger policy
    - When an operation, such as aggregation, takes place as new tuples arrive into the window
  - Partitioning
    - Maintains separate windows for each group of tuples with the same grouping key value
- Window types
  - Tumbling windows:
    - Non-overlapping sets of consecutive tuples
    - Defined only with an eviction policy
  - Sliding windows:
    - Windows formed by adding new tuples to the end of the window and evicting old tuples from the beginning of the window
    - Defined with both an eviction and a trigger policy

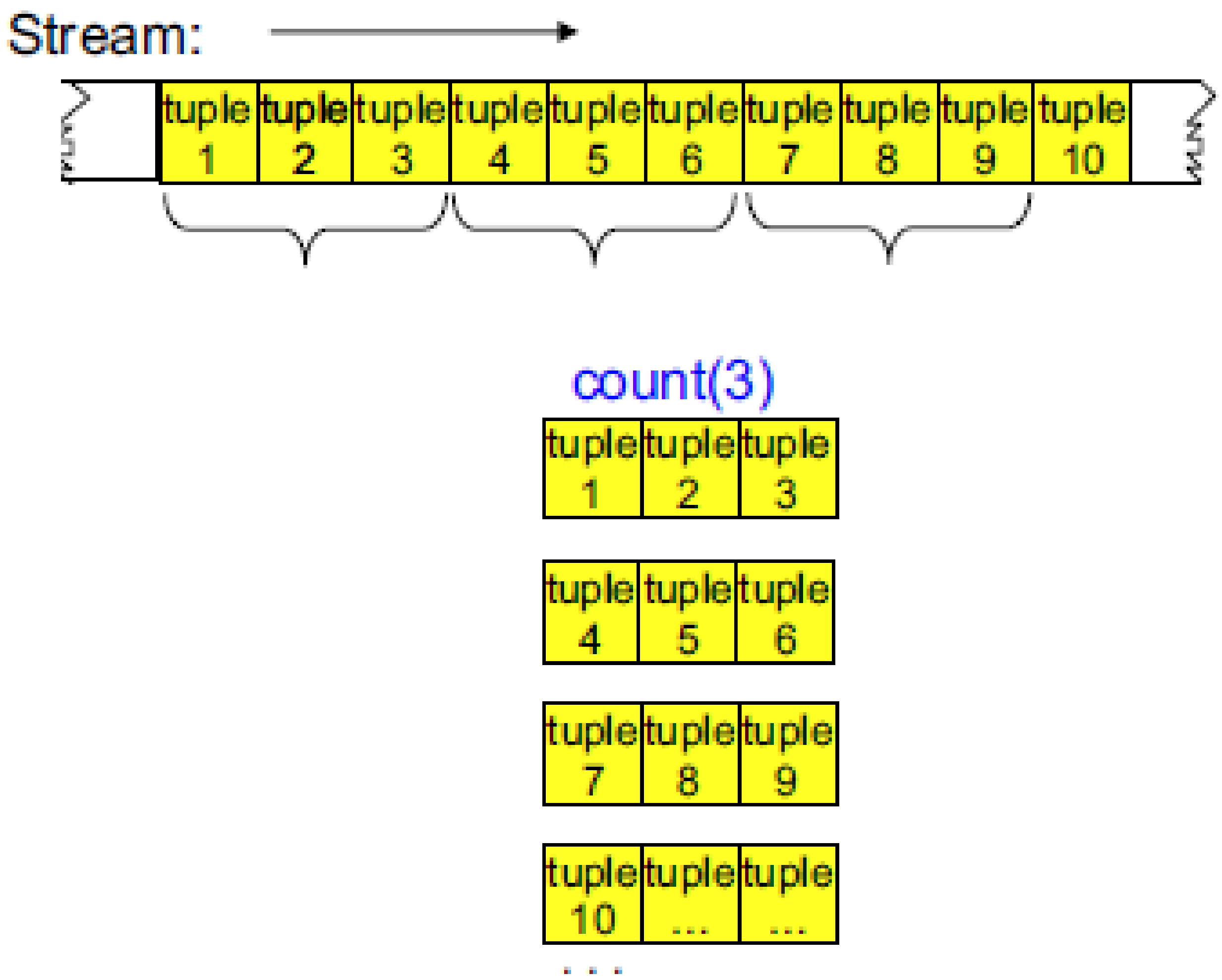
Policy Specifications

- Count (Fixed window size)
  - a fixed number set of tuples
- Time (Fixed processing time interval)
  - set of tuples that arrived in a specified period of time
- Delta (event time based on specified numeric or timestamp attribute)
  - a set of tuples where each of the tuple's specified attribute value is no more than *x* greater than that of the first tuple in the window
  - Often used with timestamps that are part of tuple data
  - Ideally expects monotonic increase in attribute value across tuples
- Punctuation
  - a set of tuples that are between punctuations (control tuples)

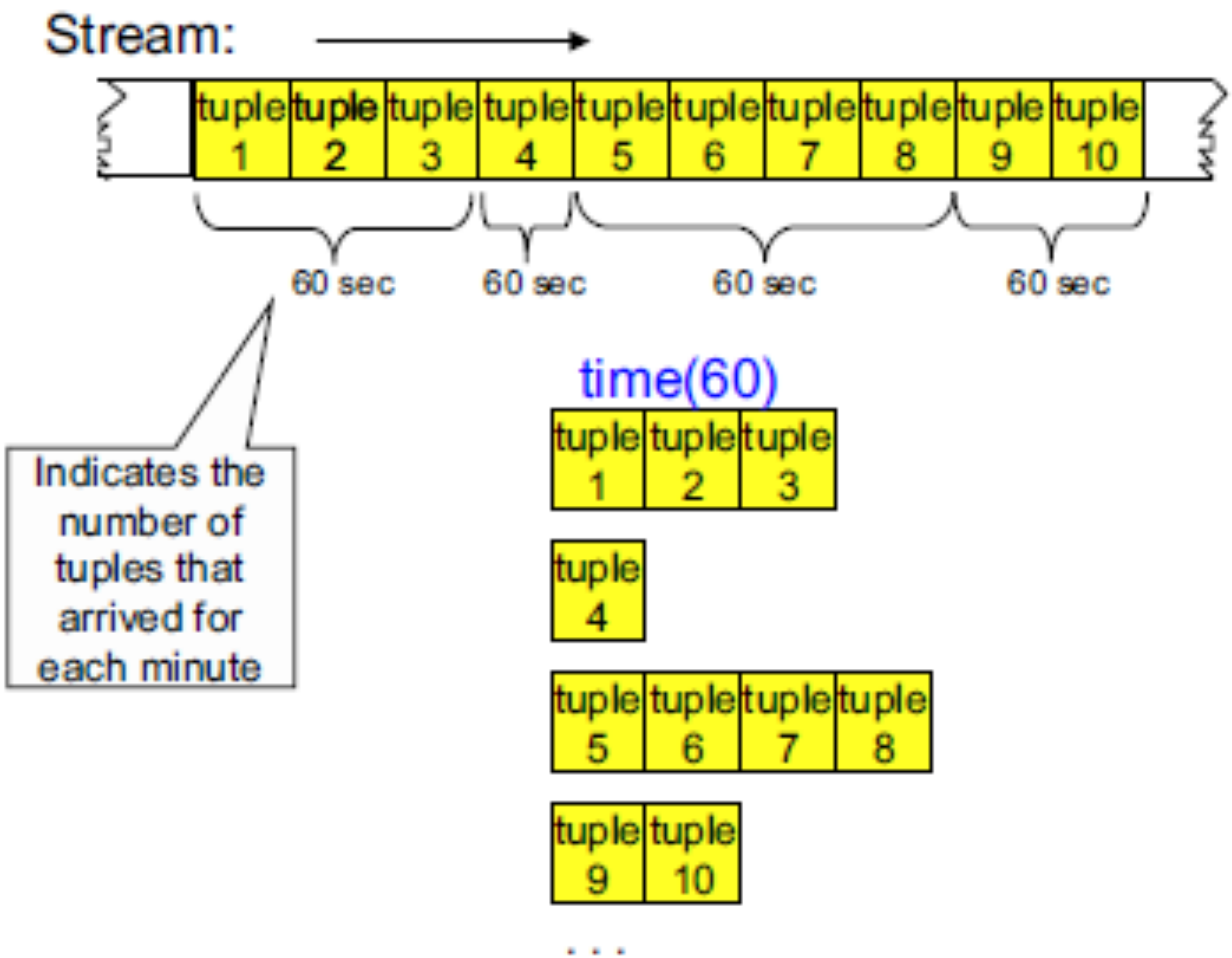
Tumbling Window

- Is specified by providing an eviction policy only
  - Punctuation
  - Count
  - Time
  - Attribute delta
- Stores tuples until the window is full
  - based upon the eviction policy
- When the window is full
  - Executes the operator behavior
- After the behavior has been executed
  - Flushes the window

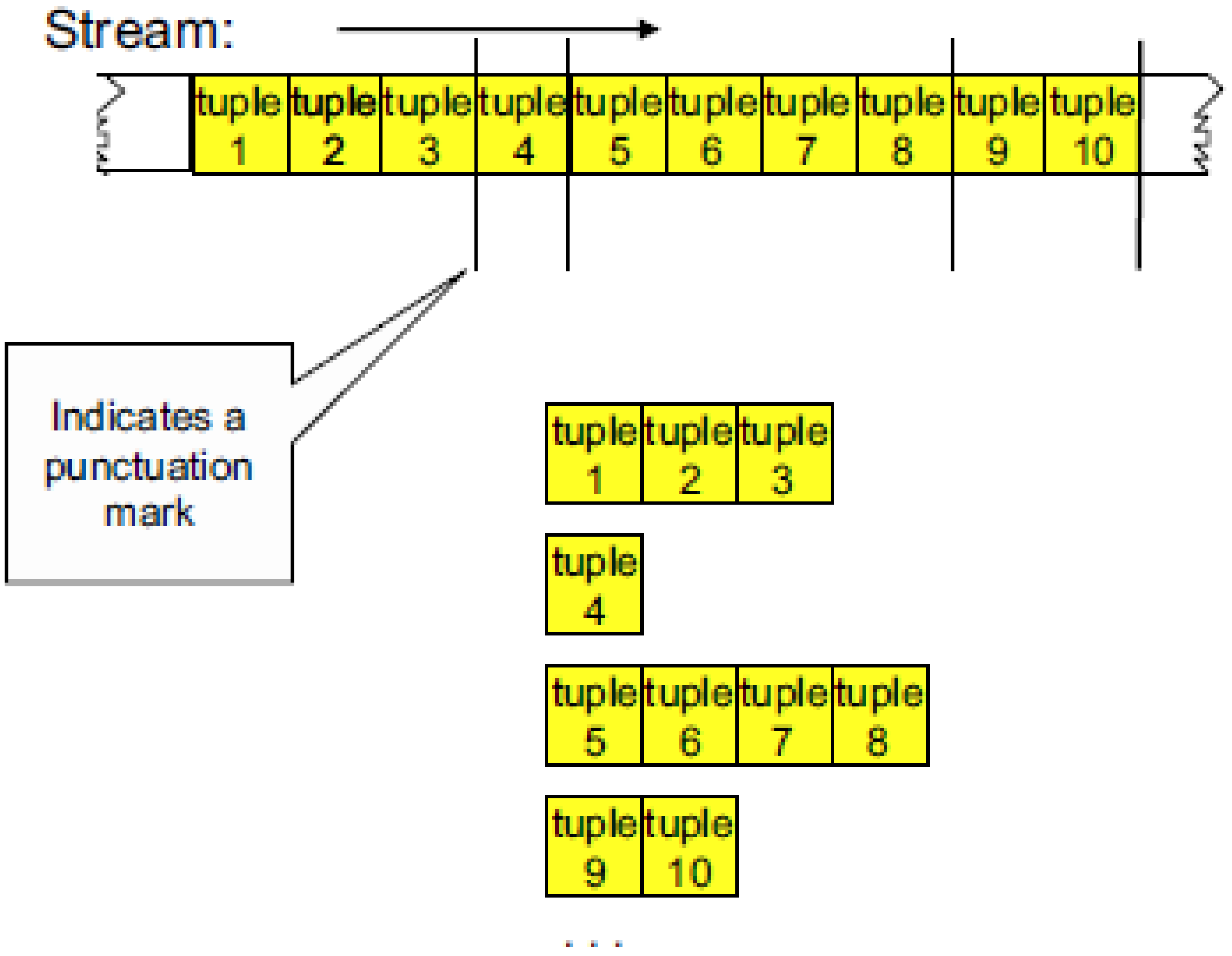
Count Based Tumbling Windows



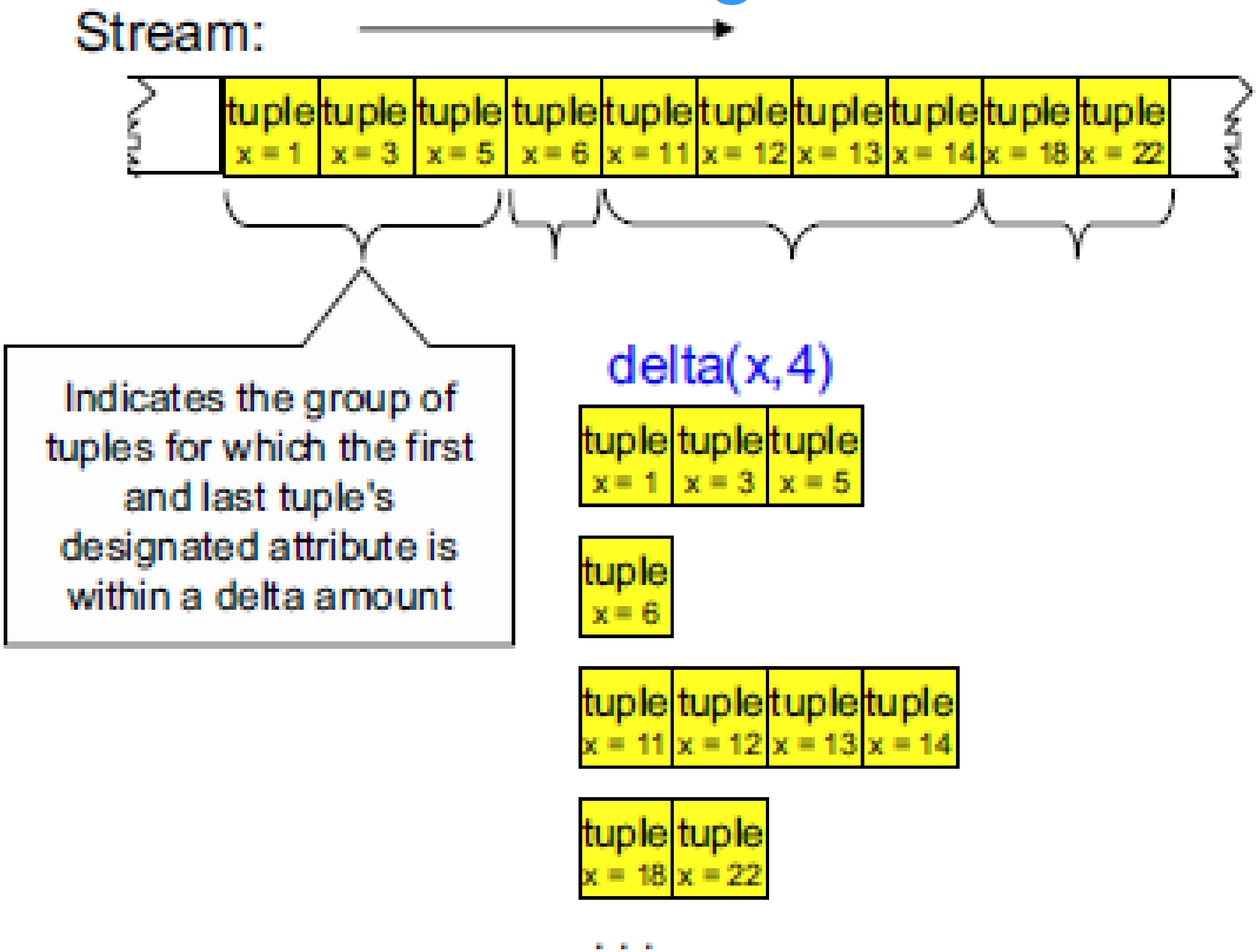
Time Based Tumbling Windows



Punctuation Based Tumbling Windows



Delta Based Tumbling Windows



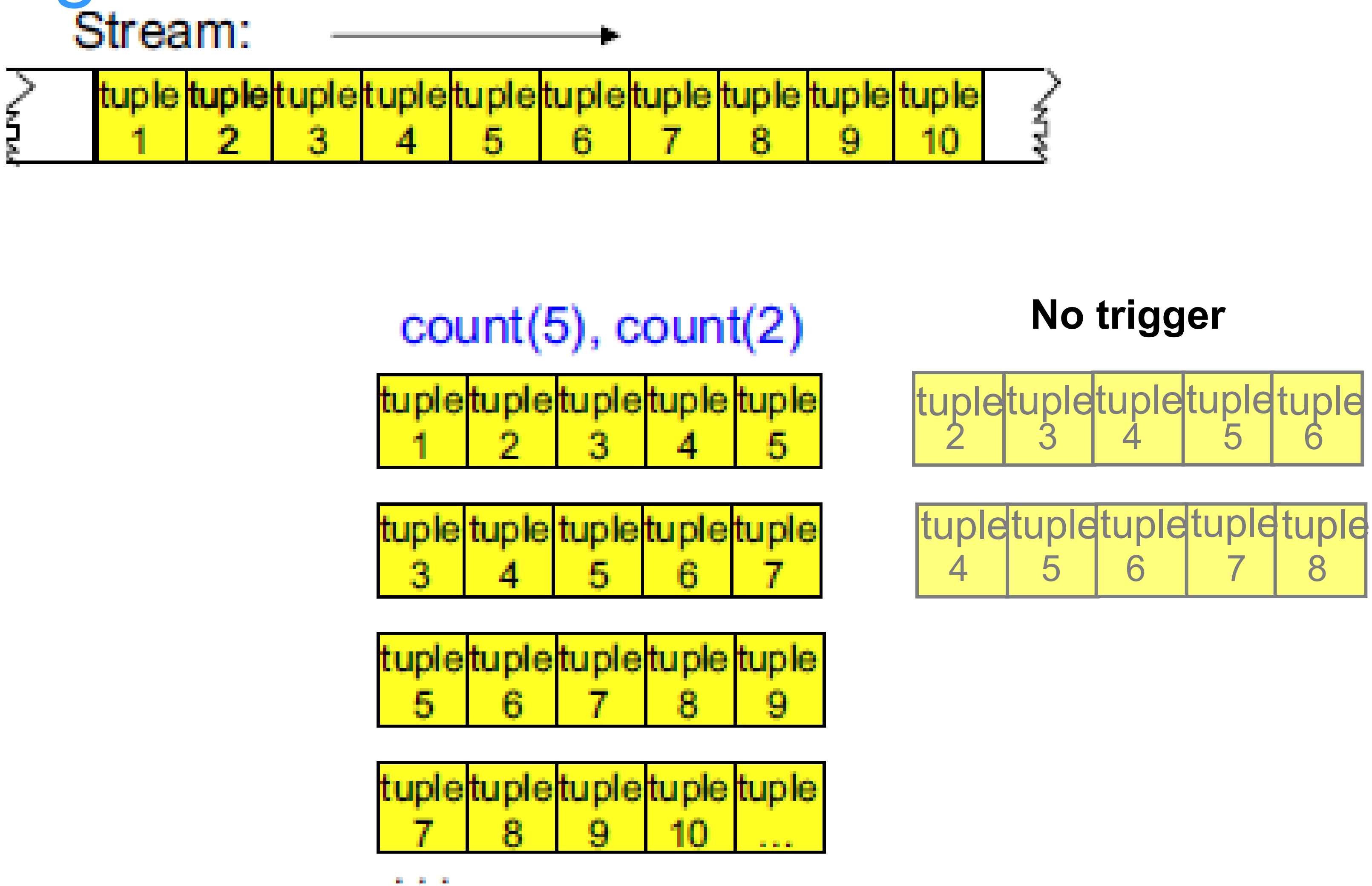
Sliding Windows – Eviction Policies

- When a new tuple arrives
  - It is added to the end of the window
  - Zero or more old tuples are evicted from the start of the window (first-in/first-out)
- Three alternatives for determining how many tuples are evicted:
  - Count
    - Once window fills up, one old tuple evicted for each arriving tuple
  - Time
    - Enough tuples evicted to maintain a maximum age
      - No remaining tuple is more than the given period older than the newly arriving tuple
    - Expired tuples are not removed when the time period is elapsed, but when a new tuple arrives
  - Attribute delta
    - Evict tuples for which a specified attribute’s value is more than *delta* less than that attribute's value in new tuple

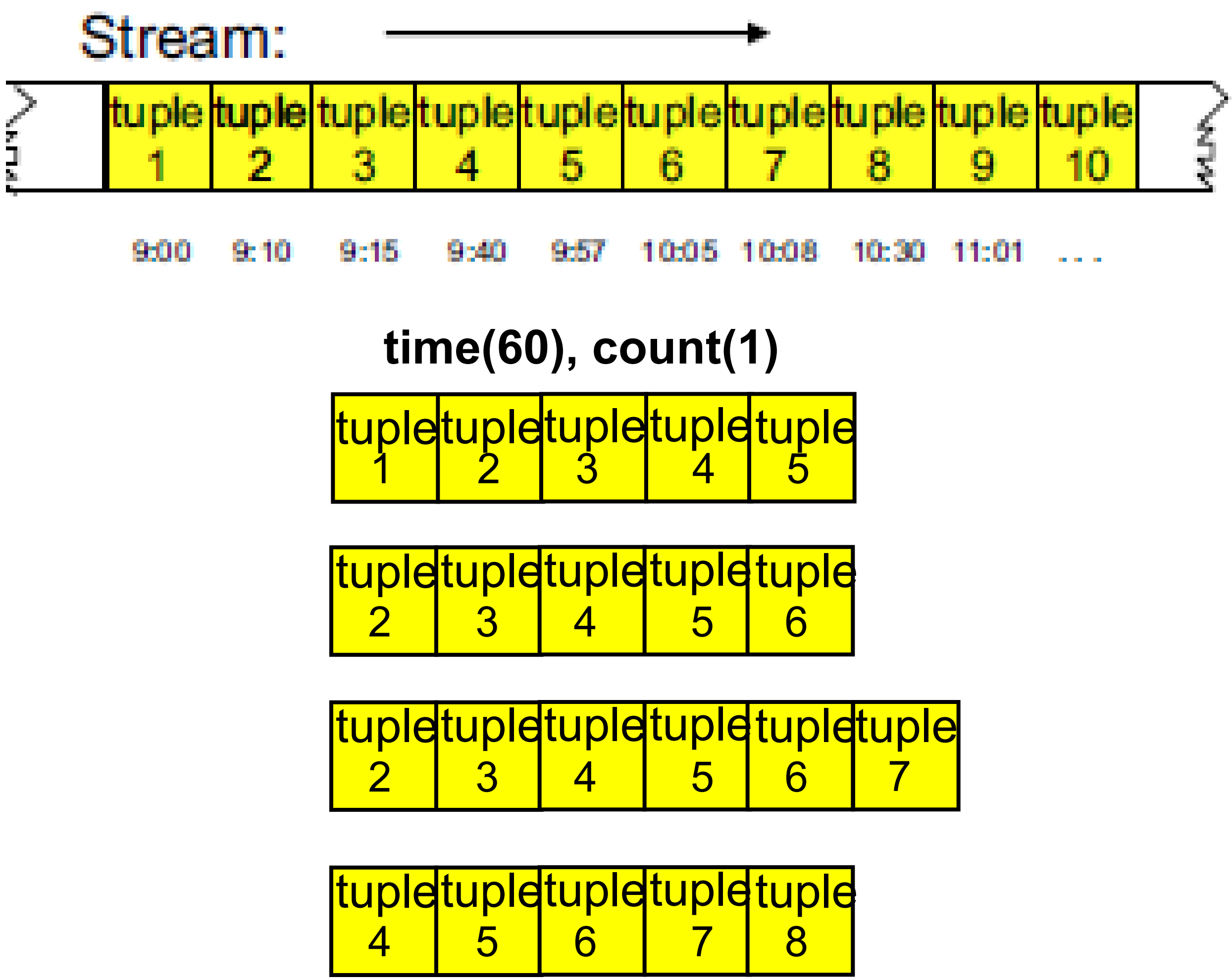
Sliding Windows – Trigger Policies

- Three alternatives for determining when a sliding window is processed:
  - count(n)
    - When a specified number of tuples have arrived since the last time a window was processed
  - time(n)
    - When a specified amount of time has passed
  - delta(attrib, value)
    - When a tuple arrives whose specified non-decreasing numeric attribute value is more than a specified amount greater than that attribute's value for the tuple that triggered the last processing operation
- Eviction policy and trigger policy are independent
  - In general, all nine combinations are possible
  - Spark supports two combinations – with same trigger and eviction policy
  - Beam supports multiple combinations –Java more than Python

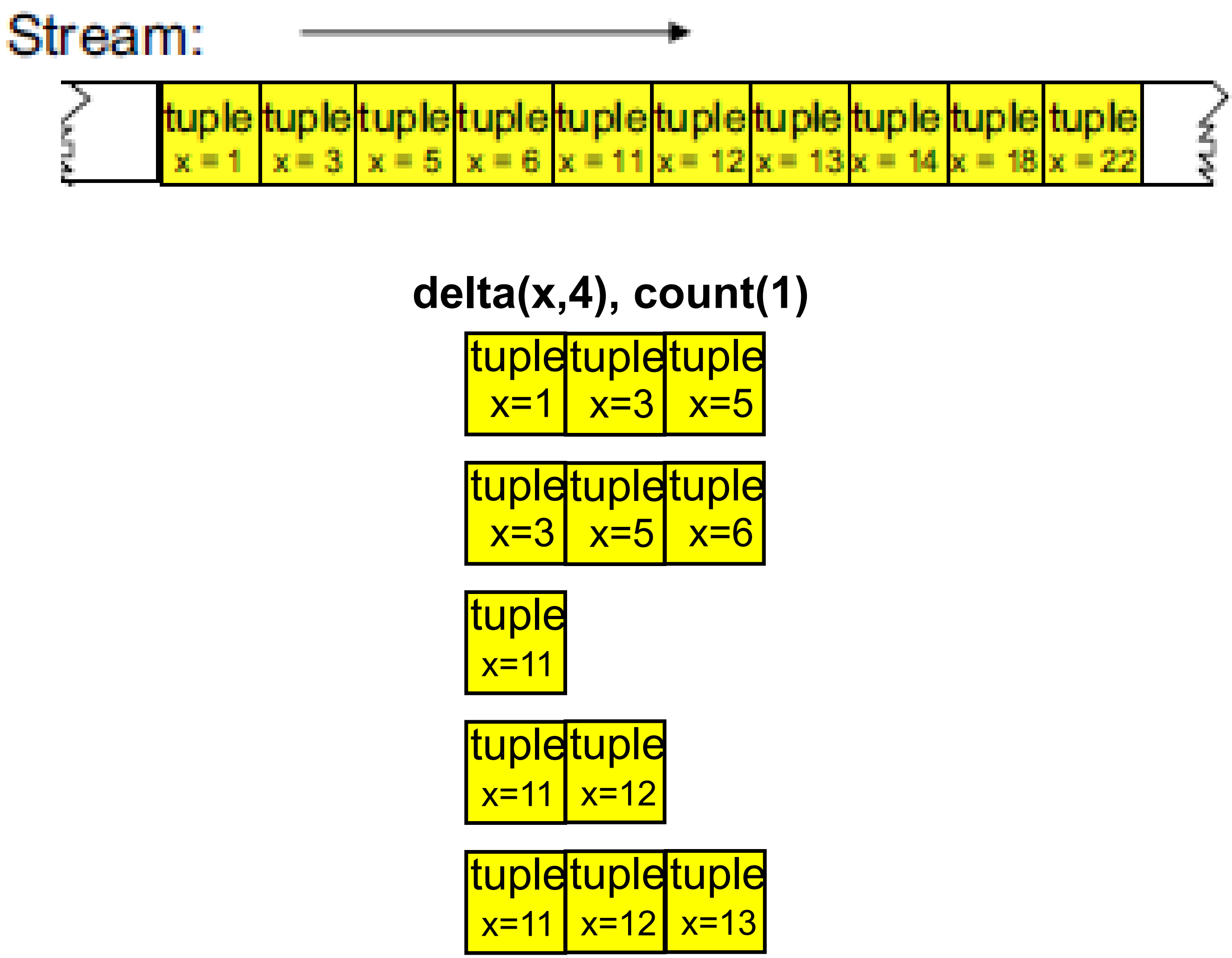
Sliding Window



Sliding Window



Sliding Window



Windowing

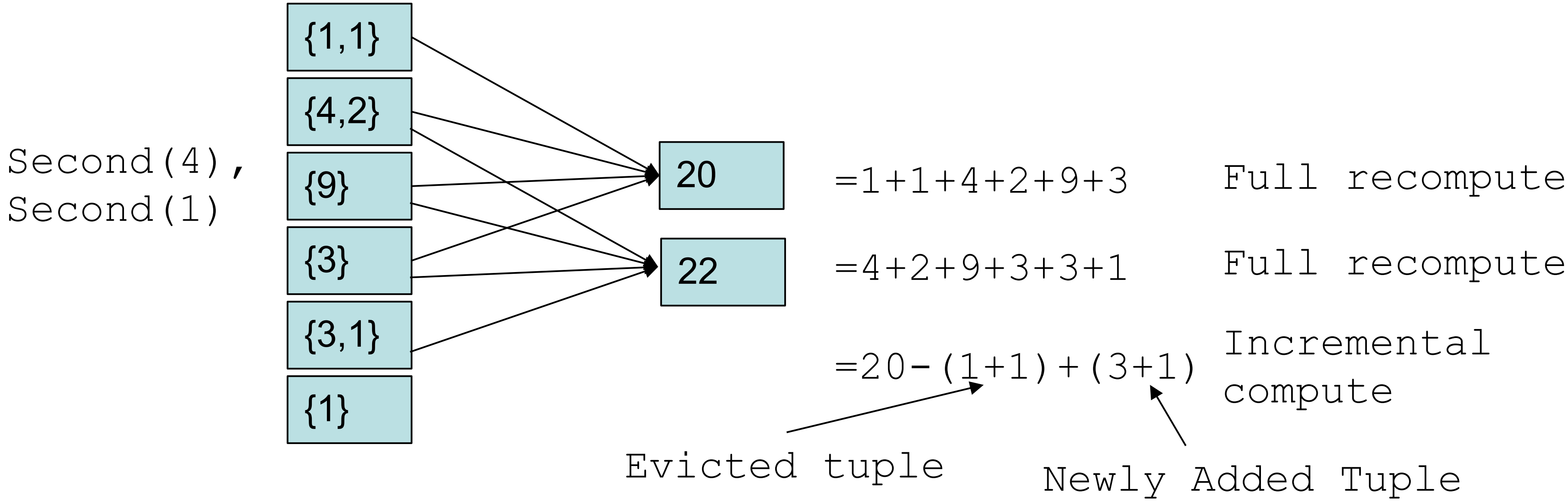
- Window creates a collection of tuples/RDDs within window
- Has size, slide – corresponds to eviction, trigger policies

```
//Create a streaming window
winLogs = accessLogs.window(30, 10)
winCounts = winLogs.countByKey()
```

- Slide cannot be smaller than batch size of Dstream
- Performs a Union across RDDs in the window
- Can then apply a reduce across this to compute additional results
  - More efficient aggregations available

Windowed Aggregations

- Incremental operations for aggregations
  - `reduceByWindow`, `reduceByKeyAndWindow`
  - Instead of re-computing fully, remove effect of evicted tuples and add in effect of newly added tuples
  - Require easily reversible functions, e.g. sum



Windowed Aggregations

```
//Create a streaming reduce with a 1 second batch, 4 second window
ipDstream = accessLogs.map(lambda x: (x.getIpAddress(),1))
ipCountDstream = ipDstream.reduceByKeyAndWindow(
                  lambda (x,y): x+y,
                  4,
                  1)
ipCountEDstream = ipDstream.reduceByKeyAndWindow(
                  lambda (x,y): x+y,
                  lambda (x,y): x-y,
                  4,
                  1)
```

Reversible function specification followed by window specification allows incremental computation, as opposed to full recompute

Other functions include `reduceByWindow()`, `countByWindow()` and `countByKeyAndWindow()`

Spark Streaming: Inputs and Outputs

- `print()` – grabs first 10 elements from each tuple and prints them to screen
- `saveAsTextFiles("output","txt")` – writes multiple files into a directory, one per tuple in txt format

- Stream of files
  - `val logData = ssc.textFileStream("logDirectory")`
  - Each file read as a tuple in the Dstream
- Other sources
  - Twitter
  - Kafka, Kinesis, Flume, and others

Limitations of Spark Streaming Model

- All windows defined in wall-clock time
  - Sizes and slides limited to multiples of batch size
- No data-time windows
  - Skewed by randomness in data arrival
  - Cannot handle out of order data
- Limited join and aggregation conditions

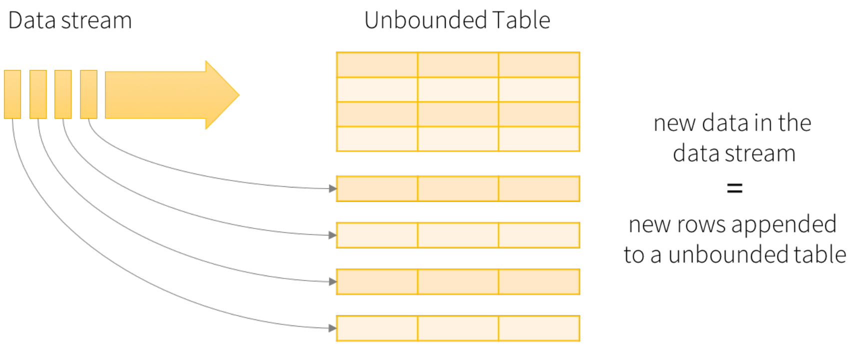
Spark Structured Stream Programming

- Uses SQL-like commands on Dataframes
  - Built on top of the Spark SQL engine
- Structured streaming queries processed as micro-batches (like Dstreams)
- Word Count Example (from before)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
spark = SparkSession.builder.appName \
       ("StructuredNetworkWordCount”).getOrCreate()
lines = spark.readStream.format("socket").option("host”,"localhost").\
        option("port", 9999).load()
words = lines.select(explode( split(lines.value, " ")).alias("word") )
wordCounts = words.groupBy("word").count()
```

Structured Stream Programming

- Different from the concept of "true" streaming
  - Each received line viewed as a row in continuously growing table (with column called "value")



Data stream as an unbounded table

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

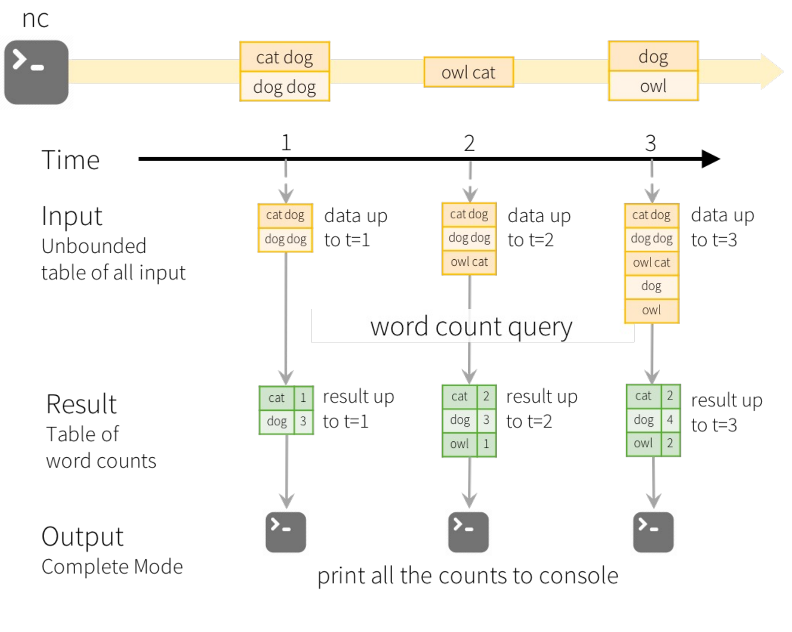
- Can write results out periodically (1 sec by default)

```
query = wordCounts.writeStream.outputMode("complete") \
          .format("console").start()

query.awaitTermination()
```

- Keeps query running in background

Writing Results Periodically

- *Complete Mode* - Entire updated Result Table written.
- *Append Mode* - Only results of new rows written.
- *Update Mode* - Only results for all updated rows mode



Model of the Quick Example

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

- Does not materialize entire table – instead computes results from previous results, incrementally

Structured Streaming Operations

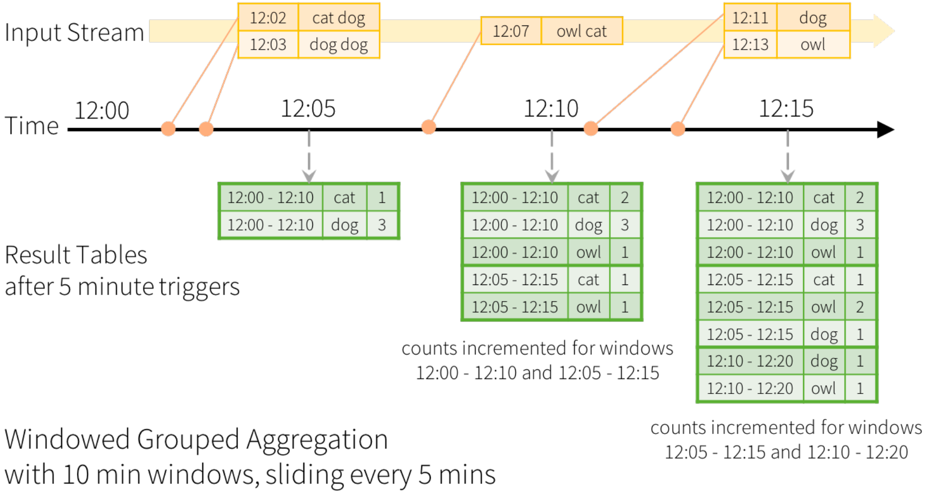
- All operations are SQL queries
  - Selection, projection, joins, aggregates etc.
  - Can create streaming dataframe from multiple sources

```
userSchema = StructType().add(”device", "string").add(”deviceType",
”string").add(“signal”,”double”).add(“time”,”DateType”)

csvDF = spark.readStream.option("sep",
";").schema(userSchema).csv("/path/to/directory")

#streaming DataFrame with schema { device: string, deviceType: string, signal:
double, time: DateType }

gt10df = csvDF.select("device").where("signal > 10")

counts = csvDF.groupBy("deviceType").count()
```

Structured Streaming: Event Windows

- Support both time based windows as well as **event time** (data time) based windows

```
words = ... # streaming DataFrame of
schema { timestamp: Timestamp, word:
String }
# Group the data by window and word and
compute the count of each group

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

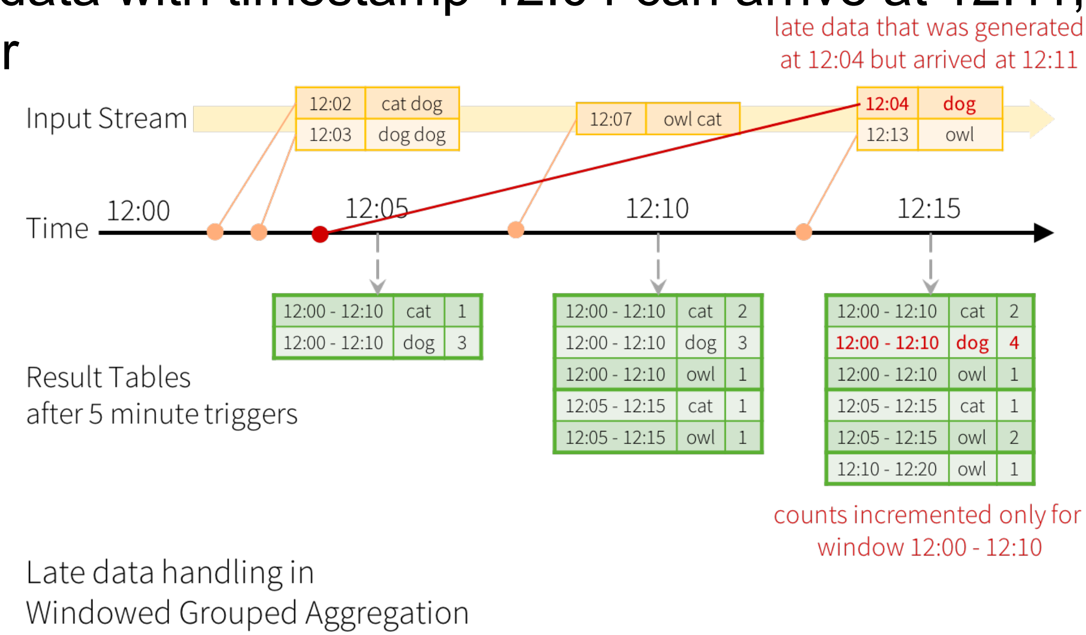


https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

- This is useful when data time and arrival time are different due to real-world issues
- Recall : Triggers are at micro-batch intervals

Structured Streaming: Watermarking

- Watermark: specifies maximum delay to wait for late arriving data
  - e.g. data with timestamp 12:04 can arrive at 12:11, or out of order



https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

- Without watermark, need to maintain unbounded state

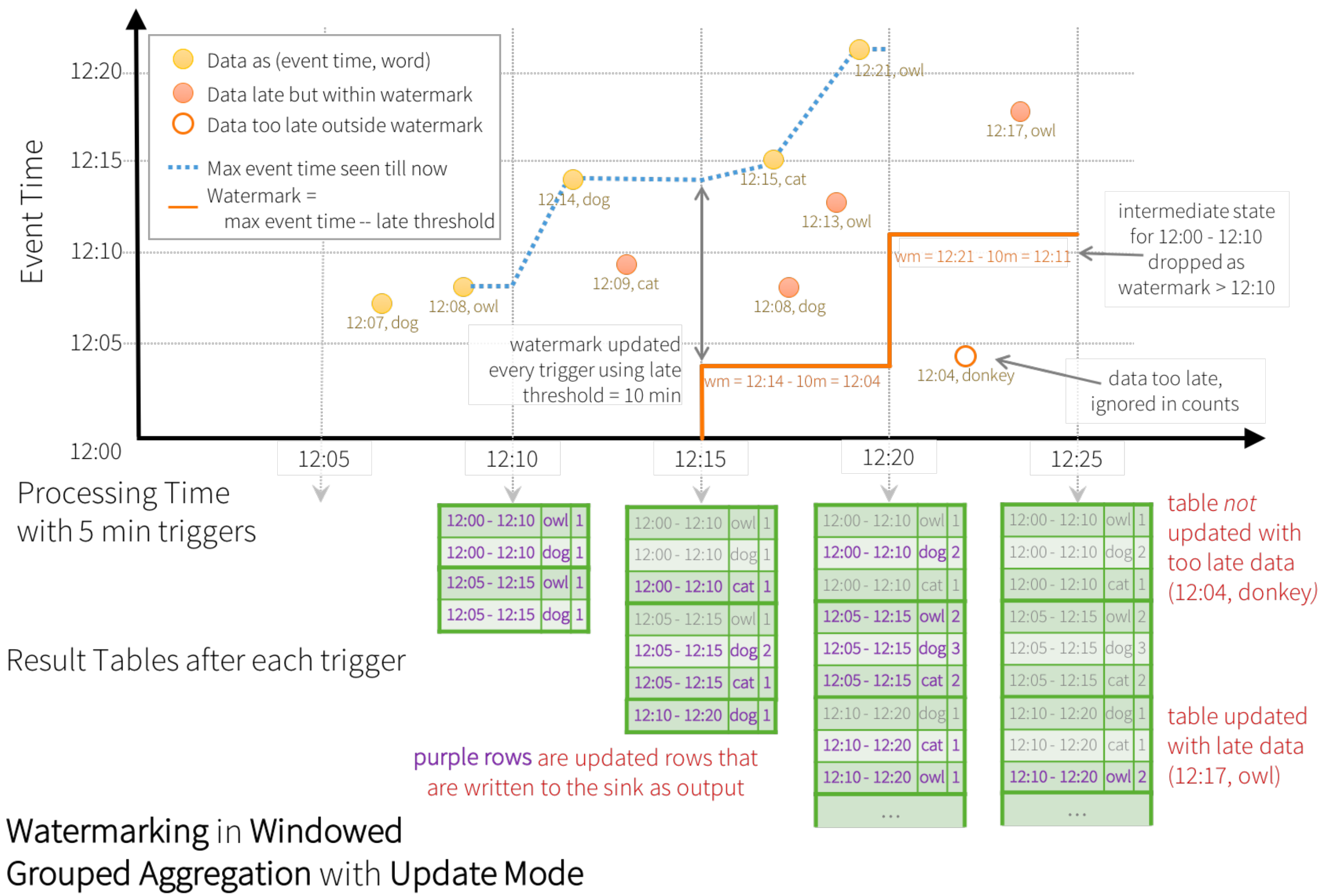
Structured Streaming: Watermarking

- Allows system to know when old data can be dropped
  - Do not need to update aggregates anymore
- Watermark specified in terms of event time
  - Defines maximum out of order in the data (not really lateness)

```
words = ... # streaming DataFrame schema { timestamp: Timestamp, word:
String }
# Group the data by window and word and compute the count of each group
windowedCounts = words.withWatermark("timestamp", "10 minutes") \
.groupBy( window(words.timestamp, "10 minutes", "5 minutes"), words.word)\
.count()
```

- Data that is older than the watermark (10 min) is discarded
- Output mode must be Append or Update

Structured Streaming: Watermarking



https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Watermarking and Other Processing

Join

```
impressions = spark.readStream....
clicks = spark.readStream....
# Apply watermarks on event-time columns impressionsWithWatermark =
impressions.withWatermark("impressionTime", "2 hours")
clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")
# Join with event-time constraints
impressionsWithWatermark.join( clicksWithWatermark, expr(""" clickAdId =
impressionAdId AND clickTime >= impressionTime AND clickTime <=
impressionTime + interval 1 hour """) )
```

Deduplicate

```
streamingDf = spark.readStream. ...
# With watermark using guid and eventTime columns
streamingDf.withWatermark("eventTime", "10 seconds") \
.dropDuplicates("guid", "eventTime")
```

- When joining streams with multiple watermarks, a global watermark is computed as the watermark of the slowest stream – so that no data is accidentally deleted

Structured Streaming: Input and Output

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

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

# Write to file/s
writeStream.format("parquet") // can be "orc", "json", "csv", etc.
.option("path", "path/to/destination/dir") .start()

# Write to Kafka
writeStream.format("kafka").option("kafka.bootstrap.servers",
"host1:port1,host2:port2").option("topic", "updates") .start()

# Write to Console
writeStream .format("console") .start()
```

Other Issues

- Recommended minimum latency is 500msec
  - Receivers can have too much overhead with less than that
- Scaling
  - Multiple receivers run in parallel, and union can be used to merge streams
  - Need to be careful about ordering
  - Can repartition the data
- Fault tolerance
  - Checkpointing based fault tolerance

References

- https://spark.apache.org/docs/latest/streaming-programming-guide.html
- https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
- H. Karau, A. Konwinski, P. Wendell and M. Zaharia, “Learning Spark”, O’Reilly Press
- http://spark.apache.org/
- Chapter 4 in Stream Processing Book

Apache Beam

Apache Beam

- Open source unified model
  - Composition language for data pipelines
  - For batch and streaming jobs
- Multiple language SDK
  - Java, Python, Go, Scala
- Multiple execution frameworks (runners)
  - Apache Apex
  - Apache Flink
  - Apache Gearpump
  - Apache Samza
  - Apache Spark
  - Google Dataflow
  - IBM Streams

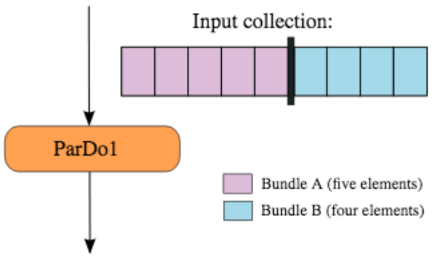
Apache Beam Runners Support

- Access transparency
  - Local and remote components accessible via same operations
- Location transparency
  - Components locatable via name, independent of location
- Concurrency transparency
  - Components/objects can be run in parallel
- Migration transparency
  - Allows movement of components without affecting other components
- Replication transparency
  - Allows multiple instances of objects for improved reliability (mirrored web pages)
- Failure transparency
  - Components designed while accounting for failure of other services

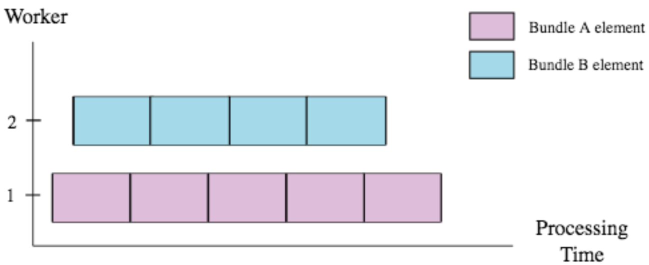
- Different runners implement these differently, however some common requirements

Runners: Access, Location, Concurrency

- Serialization and Communication
  - Shipping data items from one pipeline stage to another (core function of distributed system)
  - Done for grouping (as reduce), for parallelization, or for persistence
  - Uses either disk, network or in-memory (for transforms on the same node)
- Bundling and Parallelization
  - Focus on data parallel tasks (see later)
  - Each data stream (Pcollection) can be decomposed into ***bundles*** and distributed to multiple workers



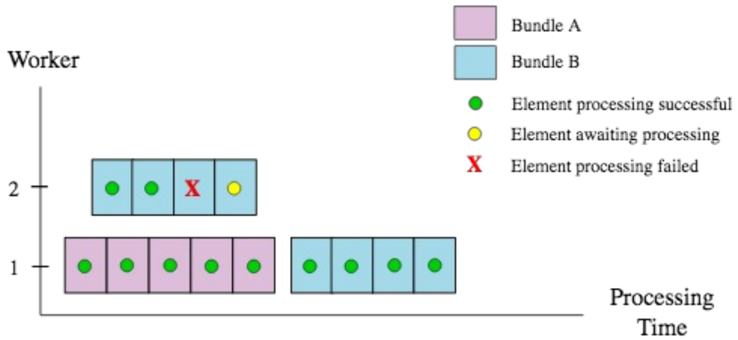
https://beam.apache.org/documentation/execution-model/



https://beam.apache.org/documentation/execution-model/

Runners: Failure Tolerance

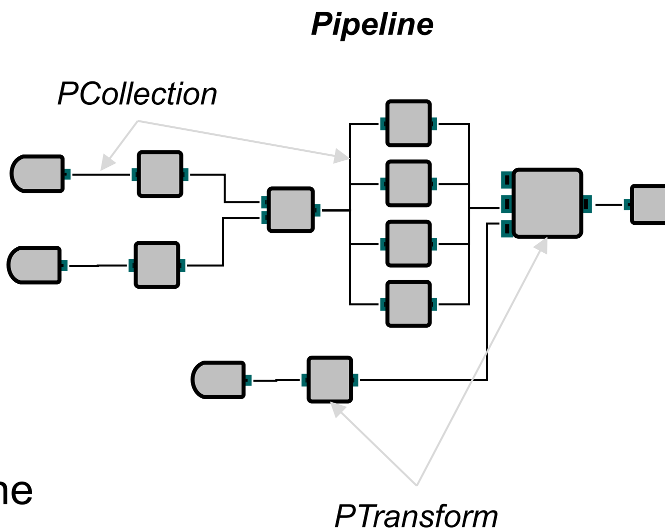
- Provided at the granularity of ***bundles***
  - If any element in the bundle fails, processing recomputed



  - Note that bundle may be processed on a different node than the original processing
- Guarantees at least once processing
- What is the difference between batch processing and stream processing?

Apache Beam Programming

- Define driver program using one of Beam SDK languages
  - Java, Python, Go
- Core abstractions
  - *Pipeline* (application flowgraph)
    - Encapsulates entire data processing task
    - Needs to specify where and how to run
  - *PCollection* (Dataset or Stream)
    - Bounded datasets for batch processing
    - Unbounded for stream processing
  - *PTransform* (Operator)
    - Data processing operation or step in the pipeline
    - Multiple special I/O transforms for PCollections



Apache Beam Programming: Pipeline

Creating a pipeline in Python

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

def mypipelinerun():
        p = beam.Pipeline(options=PipelineOptions())
```

Creating a pipeline in Java

```
import org.apache.beam.sdk.Pipeline;
import org.apache.beam.sdk.options.PipelineOptions;
import org.apache.beam.sdk.options.PipelineOptionsFactory;

public static void main(String[] args) throws IOException {
    PipelineOptions options = PipelineOptionsFactory.create();
    // Then create the pipeline.
    Pipeline p = Pipeline.create(options);
}
```

Pipeline Options: Information about runner to use, project info (e.g. cloud account) etc.
Allows reading these options from command line as `--<option>=<value>` at submission time

Pipelines and Pipeline Options

- Can define custom options to be passed at submission time

```
class MyOptions(PipelineOptions):
  @classmethod def _add_argparse_args(cls, parser):
    parser.add_argument('--input', help='Input for the pipeline’,
      default='gs://my-bucket/input’)
    parser.add_argument('--output’, help='Output for the pipeline’,
      default='gs://my-bucket/output’)
```

```
public interface MyOptions extends PipelineOptions {
  @Description("My custom command line argument.")
  @Default.String("DEFAULT") String getMyCustomOption();
  void setMyCustomOption(String myCustomOption);
}
```

Allows reading these options from command line as `--<option>=<value>` at submission time

PCollections

- Potentially distributed, multi-element dataset
  - Equivalent to Stream in the streaming case
  - Consumed and produced by Transforms (operators)
  - Can be created by reading from external interfaces or in memory

```
lines = p | 'ReadMyFile' >> beam.io.ReadFromText('gs://some/inputData.txt’)

lines = (p | beam.Create([ 'To be, or not to be: that is the question: ',
'Whether tis nobler in the mind to suffer']))
```

```
PCollection<String> lines = p.apply( "ReadMyFile",
TextIO.read().from("protocol://path/to/some/inputData.txt"));

static final List<String> LINES = Arrays.asList( "To be, or not to be: that
is the question: ", "Whether tis nobler in the mind to suffer ");
p.apply(Create.of(LINES)).setCoder(StringUtf8Coder.of())
```

`lines`: PCollection whose elements are each a single string corresponding to a line of text in the file. In the batch case, this is a finite, bounded set of elements. In the streaming case, we could read from a continuously updating file (hot file) or from network

PCollections

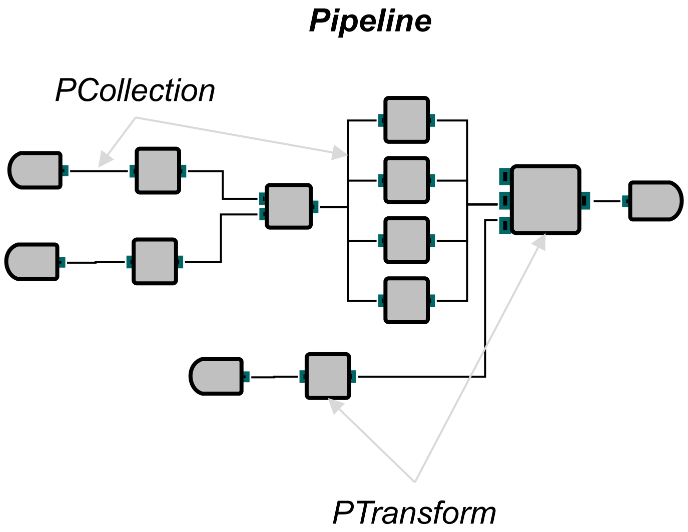
- PCollection elements can have any type
  - All elements need to have the same type
  - Need to be encoded as byte strings for distributed processing (hence need to specify encoder)
- PCollections are Immutable
  - To transform them create new Pcollections (recall RDDs)
- Random Access
  - Elements cannot be accessed in random order – collected in order of arrival (can have ordering transforms)
- Size
  - No limit on size of PCollection – can fit in memory of one machine, or distributed across multiple
  - Can be bounded (batch) or unbounded (stream)

PCollections

- Windowing
  - Used to partition unbounded Pcollections into finite sized logical sets
- Element timestamps for PCollections
  - Each element in unbounded PCollection is expected to have timestamp – assigned by source (creation or arrival time)
  - Timestamps often used for windowing
  - Batch PCollection elements are assigned the same timestamp
  - Transforms can be used to assign timestamps to elements

Transforms (Operators)

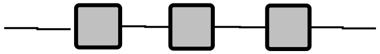
- Different operations that can be applied to 0 or more PCollections to create 0 or more PCollections
- Processing logic provided as function objects (user code)
  - Code may be run distributedly in parallel depending on the runner
- Two types of core general purpose transforms
  - ParDo and Combine – that get specialized with user code
  - Other composite transforms as well



Applying Transforms

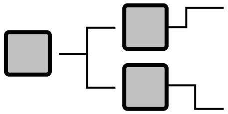
- Transforms applied to PCollection to create new PCollection
- Can be chained to apply multiple transformations

```
[Final Output PCollection] = ([Initial Input PCollection] |
[First Transform] | [Second Transform] | [Third Transform])

[Final Output PCollection] = [Initial Input
PCollection].apply([First Transform]) .apply([Second
Transform]) .apply([Third Transform])
```



- Can build DAGs of transformations

```
[Output PCollection 1] = [Input PCollection] | [Transform 1]
[Output PCollection 2] = [Input PCollection] | [Transform 2]

[Output PCollection 1] = [Input PCollection].apply([Transform 1])
[Output PCollection 2] = [Input PCollection].apply([Transform 2])
```



Core Beam Transformations

- `ParDo`
- `GroupByKey`
- `CoGroupByKey`
- `Combine`
- `Flatten`
- `Partition`
- Each of these can be extended with user code
- Represent different ways of processing data
- Analogous to different functions in Spark/Spark Streaming

ParDo Transform

- Analogous to the `map` functionality in Spark/Spark Streaming and `Functor` in Streams
- Takes each element in the PCollection, applies a function (user code) to it to create output elements that are assigned to new PCollection
- Typical Applications:
  - Filter data: (select) some elements based on a condition
  - Format or type converting data (e.g. parse strings)
  - Extract parts of each element (project)
  - Apply some computation to the elements
- To apply user code within `ParDo`, we need to define a `DoFn` class
  - Needs to conform with certain guidelines

DoFn for ParDo

Need to define `process` method with custom logic. This function is passed data, element by element, from the PCollection

```
#Compute length for each word (element in PCollection)
class ComputeWordLengthFn(beam.DoFn):
  def process(self, element):
    return [len(element)]

words = ...
word_lengths = words | beam.ParDo(ComputeWordLengthFn())
```

Why this?

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

PCollection<String> words = ...;
PCollection<Integer> wordLengths = words.apply(ParDo.of(new WordLengthFn()));
```

Input PCollection has elements that are strings, output PCollection has elements that are ints

DoFn for ParDo

- DoFns can be invoked in parallel (by different workers) or multiple times based on fault tolerance requirements
  - Unsafe to make the DoFn stateful in number of invocations
- Simple DoFns can be define inline using `lambda` functions

```
word_lengths = words | beam.ParDo(lambda word: [len(word)])

PCollection<Integer> wordLengths = words.apply( "ComputeWordLengths",
  ParDo.of(
    new DoFn<String, Integer>(){
      @ProcessElement public void processElement(@Element String word,
OutputReceiver<Integer> out) {
        out.output(word.length()); } } ) );
```

One-to-One ParDos can be implemented as `Map` transforms (another special builtin)

```
word_lengths = words | beam.Map(len)
```

GroupByKey

- Applied to PCollections with key/value pairs as elements
- Like a Shuffle operation in typical Map/Shuffle/Reduce workflows
- Groups all elements in the PCollection which have the same key

```
cat, 1
dog, 5
and, 1
jump, 3
tree, 2
cat, 5
dog, 2
and, 2
```

```
cat, [1, 5]
dog, [5, 2]
and, [1, 2]
jump, [3]
tree, [2]
```

- Applying to unbounded PCollections (streams) requires windowing to be performed first
- Note similarity to the Spark `groupByKey` applied to Pair RDDs

CoGroupByKey

- Like Join functionality with equality conditions on keys
- Applied to two or more PCollections with key/value pairs
- Groups all elements across two different PCollections that share the same key
  - Also performs a GroupByKey within each PCollection

```
emails = p | 'CreateEmails' >> beam.Create(emails_list)
phones = p | 'CreatePhones' >> beam.Create(phones_list)

results = ({'emails': emails, 'phones': phones} | beam.CoGroupByKey())
```

```
PCollection<KV<String, String>> emails = p.apply("CreateEmails",
Create.of(emailsList));
PCollection<KV<String, String>> phones = p.apply("CreatePhones",
Create.of(phonesList));

PCollection<KV<String, CoGbkResult>> results =
KeyedPCollectionTuple.of(emailsTag, emails) .and(phonesTag, phones)
.apply(CoGroupByKey.create());
```

CoGroupByKey

```
emails_list = [
('amy', 'amy@example.com’),
('carl', 'carl@example.com'),
('julia', 'julia@example.com'),
('carl', 'carl@email.com’),
]
phones_list = [
('amy', '111-222-3333’),
('james', '222-333-4444’),
('amy', '333-444-5555’),
('carl', '444-555-6666’),
]
```

CoGroup ByKey →

```
results = [
('amy’, {
   'emails': ['amy@example.com’],
   'phones': ['111-222-3333', '333-444-
5555’]
}),
('carl', {
  'emails': ['carl@email.com’,
     'carl@example.com’],
  'phones': ['444-555-6666’]
}),
('james', {
  'emails': [],
  'phones': ['222-333-4444’]
}),
('julia', {
  'emails': ['julia@example.com’],
  'phones': []
}),
]
```

Combine

- Like Aggregate functionality
- Applied to PCollection to combine elements using an appropriate aggregation function
  - Aggregation function should be commutative and associative
  - Prebuilt aggregation functions (sum, max, min) available

```
pc = [1, 10, 100, 1000]
small_sum = pc | beam.CombineGlobally(SumFn())

def bounded_sum(values, bound=500):
  return min(sum(values), bound)

small_sum = pc | beam.CombineGlobally(bounded_sum)
```

- Can define both simple and complex aggregation functions

Combine: Custom Aggregation

- Example Aggregation Function

```
class AverageFn(beam.CombineFn):
  def create_accumulator(self):
    return (0.0, 0)
  def add_input(self, sum_count, input):
    (sum, count) = sum_count
    return sum + input, count + 1
  def merge_accumulators(self, accumulators):
    sums, counts = zip(*accumulators)
    return sum(sums), sum(counts)
  def extract_output(self, sum_count):
    (sum, count) = sum_count
    return sum / count if count else float('NaN')

average = pc | beam.CombineGlobally(AverageFn())
```

- Recall similarity to aggregate in Pyspark
  - In the streaming setting these are applied across a window
  - Need to specify default behavior for empty windows
- Can also similarly have `CombinePerKey`
  - for PCollections with key,value pairs

Flatten and Partition

- Flatten
  - merges PCollections of the same type into one PCollection
  - Like a stream combiner

```
merged = ( (pcoll1, pcoll2, pcoll3) | beam.Flatten())
```

- Partition
  - splits a single PCollection into multiple PCollections based on partition condition

```
#Define partitioning function that return integer
students = ...
def partition_fn(student, num_partitions):
  return int(get_percentile(student) * num_partitions / 100)

by_decile = students | beam.Partition(partition_fn, 10)
fortieth_percentile = by_decile[4]
```

  - Individual partitions can be accessed by index

Beam Writing Custom Functions

- Recall: function code gets executed distributedly
  - Can have multiple copies running in parallel
  - Can also be retried based on failure/s
  - Code must be serializable and thread safe
- Serializable
  - Should not include large amounts of data and only include serializable classes – need to ship to workers
- Thread Safe
  - Care when creating threads
- ‘Idempotent’
  - May be executed on the same data multiple times

Side Information for ParDo Transforms

- Useful to be able to have some side information
  - Not directly from the streaming data, but determined at runtime
  - E.g. Machine learning model, or parameters
- Passed as extra parameters to the process function of the ParDo

```
class FilterUsingLength(beam.DoFn):
  def process(word, lower_bound, upper_bound=float('inf')):
    if lower_bound <= len(word) <= upper_bound:
      yield word

small_words = words | beam.ParDo(FilterUsingLength(), 0, 3)
```

- Can similarly pass side input to other Beam transforms
  - E.g. FlatMap

Multiple Output Streams from ParDo

- Can create multiple output streams from transform

```
#Three output streams, main, above_cutoff_lengths, and marked strings
class GetWords(beam.DoFn):
  def process(self, element, cutoff_length, marker):
    if len(element) <= cutoff_length:
      yield element
    else:
      yield pvalue.TaggedOutput( 'above_cutoff_lengths’, len(element))
    if element.startswith(marker):
      yield pvalue.TaggedOutput('marked strings', element)


results = (words | beam.ParDo(GetWords(), cutoff_length=2, marker='x’)
             .with_outputs('above_cutoff_lengths', 'marked strings’,
             main='below_cutoff_strings’))

below = results.below_cutoff_strings
above = results.above_cutoff_lengths
marked = results['marked strings']
```

- Access streams by tags or by keys

Pipeline I/O – Reading and Writing

- Built in connectors
  - Memory, File, Cloud Storage, Databases, Pub-sub…

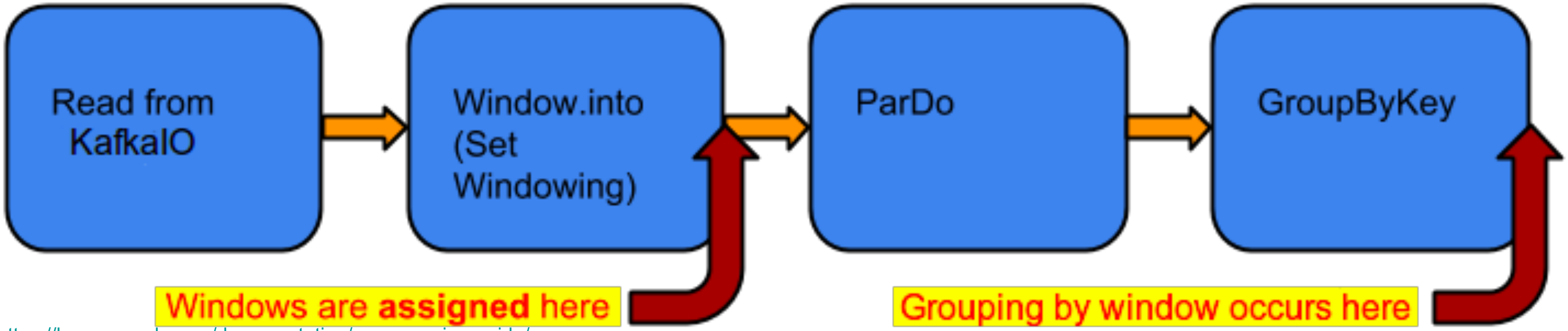
```
output | beam.io.WriteToText('gs://some/outputData’)
lines = p | 'ReadFromText' >> beam.io.ReadFromText('input-*.csv’)
```

```
output.apply(TextIO.write().to("gs://some/outputData"));
p.apply(“ReadFromText”,
TextIO.read().from("protocol://my_bucket/path/to/input-*.csv");
```

	File	Messaging	Database
Java	FileIO, AvroIO, TextIO, TFRecordIO, XmlIO, TikaIO, ParquetIO	Kinesis, Kafka, Pub/Sub, MQTT, JMS	Cassandra, Hadoop InputFormat, Hbase, Hive, Apache Kudu, Apache Solr, Elasticsearch, Google BigQuery, Google Cloud Bigtable, Google Cloud Datastore, Google Cloud Spanner, JDBC, MongoDB, Redis
Python	avroio, textio, tfrecordio, vcfio	GC Pub/Sub	Google BigQuery, Cloud Datastore

Beam Windowing

- Windowing used to partition PCollections based on individual tuple timestamps
  – Support attribute delta (specifically based on timestamp/arrival time)
- Each element in a PCollection is assigned to one or more windows
- Each individual window contains a finite number of elements
- All transforms and aggregations are applied on a window by window basis
  – E.g. GroupByKey groups elements by key and window
- Windows actually used when aggregation needs to be performed



From https://beam.apache.org/documentation/programming-guide/

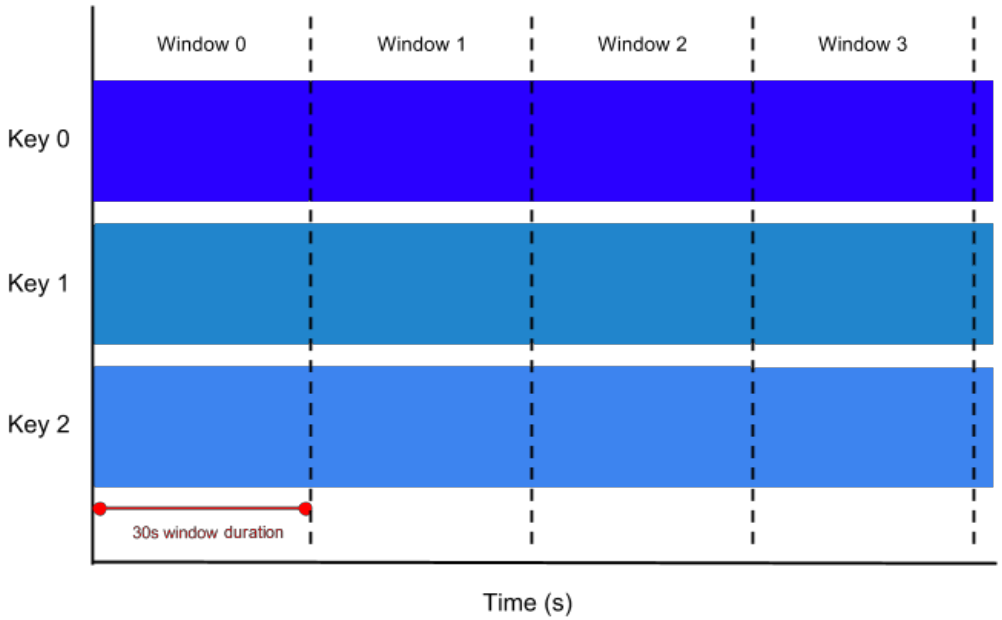
Built-in Windowing Support

- Delta-based Windows
  - Fixed Time Windows (Tumbling Windows)
  - Sliding Time Windows
- Per-Session Windows
- Single Global Window
- Calendar-based Windows

- All windowing performed per Key
- Custom Windows can also be defined with a `WindowFn`

From https://beam.apache.org/documentation/programming-guide/

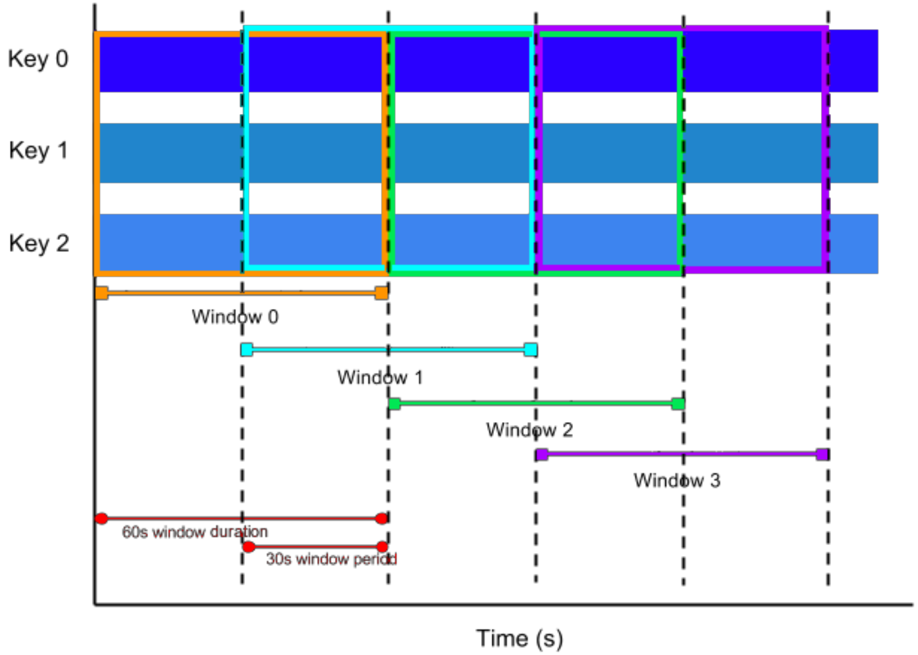
Beam Fixed Time Windows




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

```
from apache_beam import window
fixed_windowed_items=(items | 'window' >> beam.WindowInto(window.FixedWindows(30)))
```

Beam Sliding Time Windows




```
from apache_beam import window
sliding_windowed_items=( items | 'window' >>
                          beam.WindowInto(window.SlidingWindows(60, 30)))
```

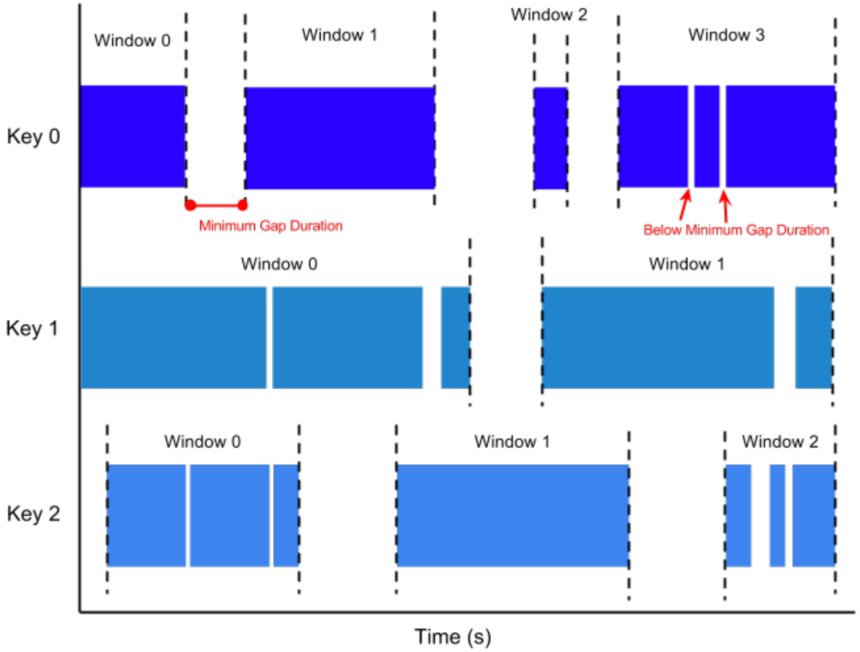
```
PCollection<String> items = ...;
PCollection<String> slidingWindowedItems = items.apply(
      Window.<String>into(SlidingWindows.of(Duration.standardSeconds(60)).
          every(Duration.standardSeconds(30))));
```

Beam Session Windows

- Defined for irregularly spaced data
  - where gap in data arrival specifies window boundaries

```
from apache_beam import window
session_windowed_items = ( items | 'window’
     >> beam.WindowInto(window.
        Sessions(10 * 60)))
```

```
PCollection<String> items = ...;
PCollection<String> sessionWindowedItems =
        items.apply(Window.<String>
           into(Sessions.withGapDuration
           (Duration.standardMinutes(10))));
```



  - New window started when the gap in arrival is greater than the specified minimum value

Watermarks and Late Data

- Used to account for difference between event time and arrival time
  - Can specify max lateness in data arrival to allow it to be added to window
    - E.g. if max lateness is 30 seconds and we have a 5:00 minute window, then Beam will wait 30 seconds for such data
    - If at 5:29 in real time, we have data with event time 4:45 then it will be added to window 1
    - If at 5:34 in real time we have data with event time 4:43 – this data will be discarded – since it arrived later than the max lateness
- Only supported in Java, not in Python
- Watermark estimated by Beam, based on data arrival patterns, and user specification

```
PCollection<String> items = ...;
PCollection<String> fixedWindowedItems = items.apply(
   Window.<String>into(FixedWindows.of(Duration.standardMinutes(1)))
     .withAllowedLateness(Duration.standardDays(2)));
```

Timestamps and PCollections

- Streaming data comes with timestamps naturally
  - Batch data does not
- Need to tell Beam how to extract/assign timestamps from the data
- Apply ParDo transforms to do this

```
class AddTimestampDoFn(beam.DoFn):
  def process(self, element):
    unix_timestamp = extract_timestamp_from_log_entry(element)
    yield beam.window.TimestampedValue(element, unix_timestamp)

timestamped_items = items | 'timestamp' >> beam.ParDo(AddTimestampDoFn())
```

```
PCollection<LogEntry> unstampedLogs = ...;
PCollection<LogEntry> stampedLogs = unstampedLogs.apply(ParDo.of(new DoFn<LogEntry,
LogEntry>(){
  public void processElement(@Element LogEntry el, OutputReceiver<LogEntry> out){
     Instant logTimeStamp = extractTimeStampFromLogEntry(element);
     out.outputWithTimestamp(element, logTimeStamp);
  }}));
```

Window Triggers

- Determine when results from the window are emitted
- Event time triggers
- Processing time triggers
- Data-driven triggers
- Composite triggers

Window Triggers and Composition

- Event Triggers: operate on the event time (data time), e.g. `AfterWaterMark` trigger
  - Processing performed when watermark passes end of window
  - Recall watermarks and Structured Spark Streaming
- Processing Time Triggers: operate on wall clock time, e.g. `AfterProcessingTime` trigger
  - Processing performed interval after the first element in window is received
  - Supports count and time based interval specification
- Data-driven Triggers: element count, e.g. `AfterCount` trigger
  - Processing performed after receiving a certain number of tuples
- Different types of triggers can be composed

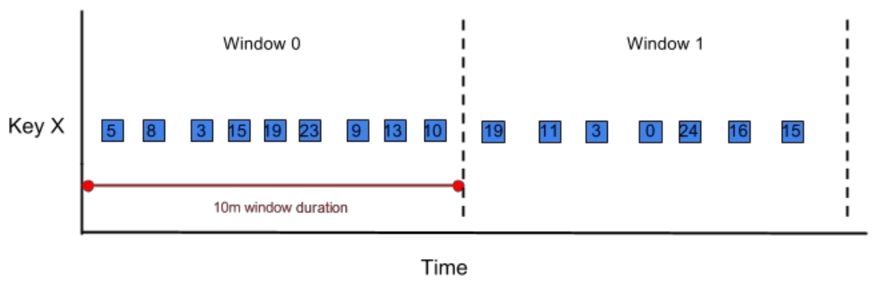
```
AfterWatermark(early=AfterProcessingTime(delay=1*60), late=AfterCount(1))

AfterWatermark.pastEndOfWindow()
  .withEarlyFirings( AfterProcessingTime .pastFirstElementInPane()
    .plusDuration(Duration.standardMinutes(1))
  .withLateFirings(AfterPane.elementCountAtLeast(1))
```

Setting Window Triggers

- Associated with the `WindowInto` transform

```
pcollection | WindowInto( FixedWindows(1 * 60),
              trigger=AfterProcessingTime(10 * 60),
              accumulation_mode=AccumulationMode.DISCARDING)

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

- Accumulation Mode determines what happens to windows on each firing (since repeated firings possible)



`accumulatingFiredPanes()` **Mode**
First trigger firing: [5, 8, 3]
Second trigger firing: [5, 8, 3, 15, 19, 23]
Third trigger firing: [5, 8, 3, 15, 19, 23, 9, 13, 10]

`discardingFiredPanes()` **Mode**
First trigger firing: [5, 8, 3]
Second trigger firing: [15, 19, 23]
Third trigger firing: [9, 13, 10]

Handling Late Data

- In Python – use parameter allowed_lateness

```
pc = [Initial PCollection] pc |
beam.WindowInto( FixedWindows(60),
trigger=trigger_fn,
accumulation_mode=accumulation_mode,
allowed_lateness=Duration(seconds=2*24*60*60))
# 2 days
```

- In Java – use `withAllowedLateness()` to specify

References

- Apache Beam Documentation: https://beam.apache.org/documentation/programming-guide/
- Apache Beam Community: https://beam.apache.org/community/contact-us/