Beam SDK Tutorial

Apache Beam Summit - Beam Introduction

Berlin, June, 2019

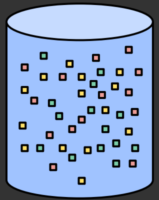
01 Introduction

The Apache Beam programming model

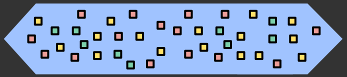
What is part of Apache Beam?

One Model

Multiple Modes



Batch



Streaming

Multiple SDKs

Java

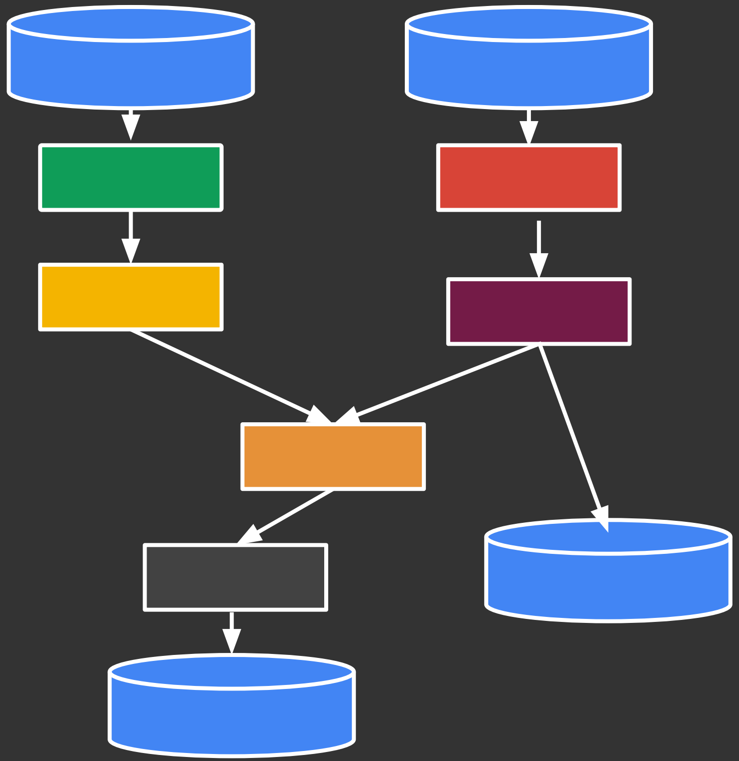
Python

Go

Multiple Runners

- **Direct**: local for testing
- **Cloud Dataflow**: fully managed service on Google Cloud
- **Apache Flink**: local, on-premise, cloud
- **Apache Spark**: local, on-premise, cloud

What is a pipeline?



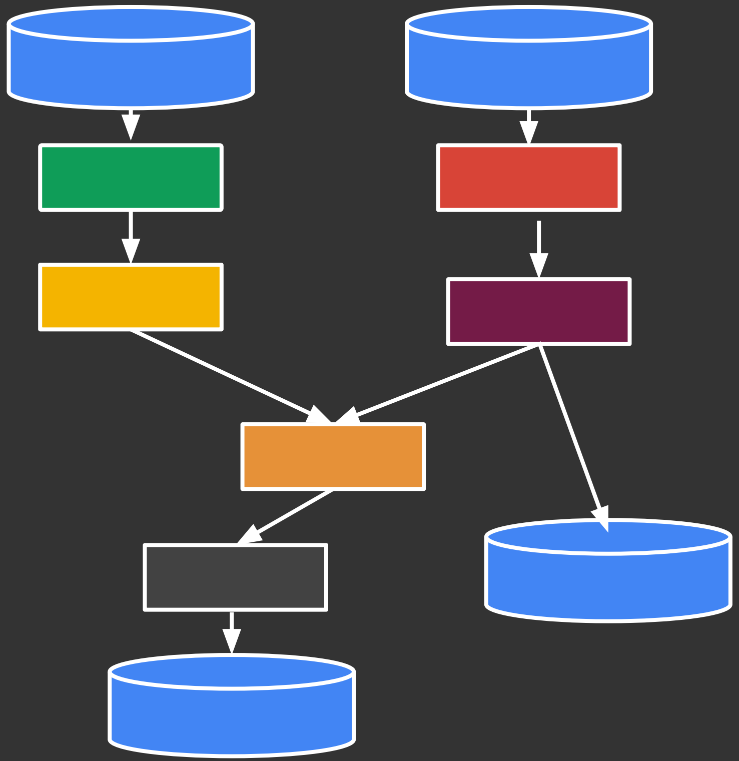
- A Directed Acyclic Graph of data **transformations** applied to one or more **collections** of data
- Possibly **unbounded collections** of data flow on the edges
- May include multiple sources and multiple *sinks*
- Optimized and executed as a unit

Apache Beam Ecosystem

Beam

- Unified programming model
- Portable
  - Multiple Runners
  - Multiple Languages (Java, Python, Go)
- Extensible:
  - IO: GCP + major open source + APIs
  - DSLs: SQL, Scala

What is a pipeline?



- Beam represents datasets using an abstraction called `PCollection`
- Data transformations are represented by an abstraction called `PTransform`

The pipeline describes...

What are you computing?

Where in event time?

When in processing time?

How do refinements relate?

The pipeline describes...

What = Transformations

Where = Windowing

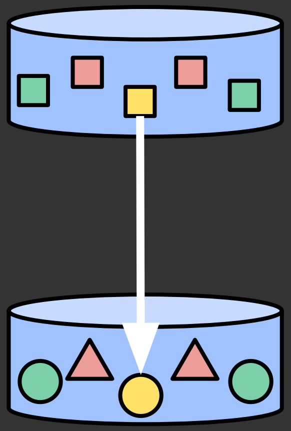
When = Watermarks + Triggers

How = Accumulation

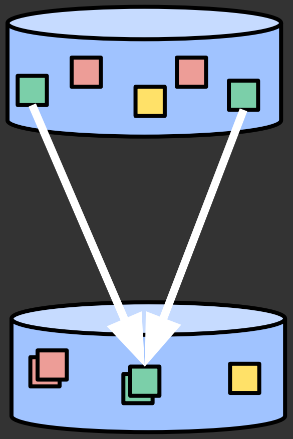
02 Writing a pipeline

What results are calculated?

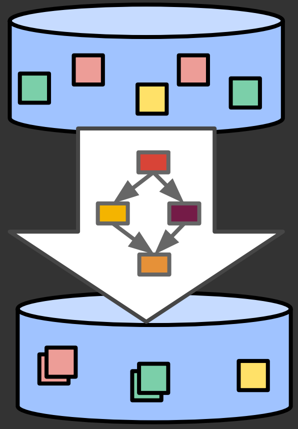
What are you computing?



Element-Wise
(map)

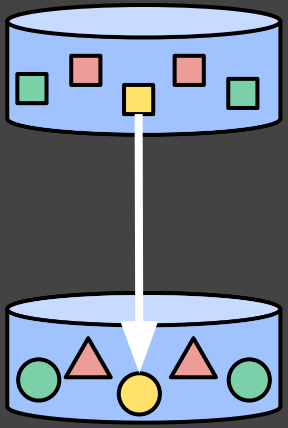


Aggregating
(reduce)

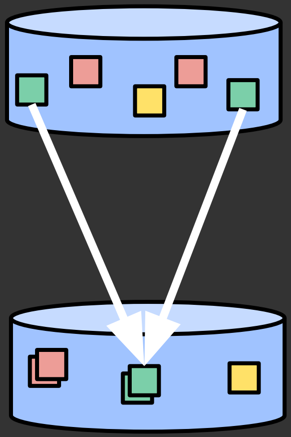


Composite
(reusable combinations)

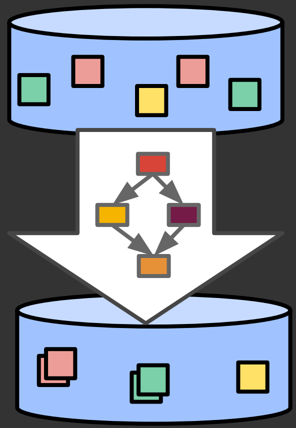
What are you computing?



Element-Wise
(map)



Aggregating
(reduce)



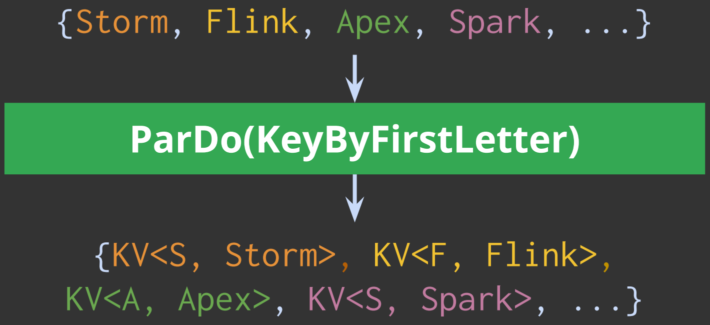
Composite
(reusable combinations)

Element-wise transforms: ParDo

(ParDo = “Parallel Do”)

Performs a user-provided transformation on each element of a PCollection independently

ParDo can be used for many different operations...

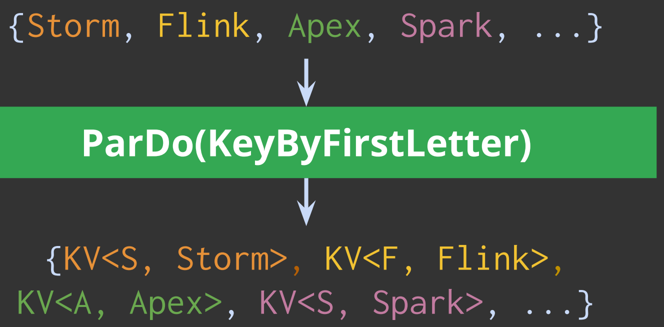

{Storm, Flink, Apex, Spark, ...}
ParDo(KeyByFirstLetter)
{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>, KV<S, Spark>, ...}

Element-wise transforms: ParDo (Java)

```java
Pipeline p = Pipeline.create(options);

PCollection<String> input = p.apply(...);

firstLetters = input.apply(ParDo.of(
    new DoFn<String, KV<Char, String>>() {
  @ProcessElement
  public void processElement(
   @Element String word, OutputReceiver<> o) {
    Char firstLetter = word.charAt(0);
    o.output(KV.of(word.charAt(0), word));
  }
}));
```



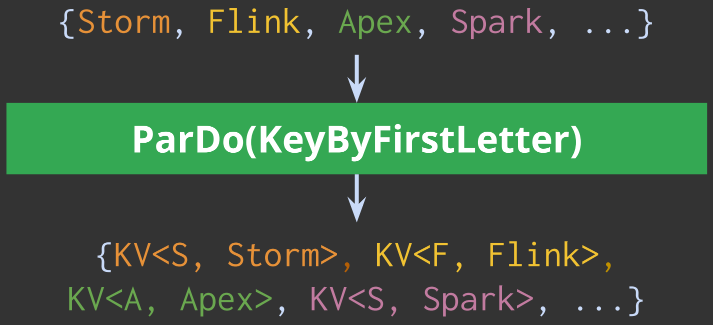
{Storm, Flink, Apex, Spark, ...}

ParDo(KeyByFirstLetter)

{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>, KV<S, Spark>, ...}

Element-wise transforms: ParDo (Python)

```
class FirstLetter(beam.DoFn):
  def process(self, word):
    return [word[0]]
input = …;
firstLetters = input | beam.ParDo(FirstLetter)
```



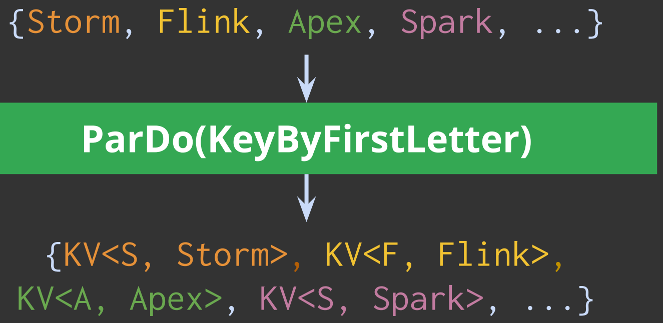
{Storm, Flink, Apex, Spark, ...}

ParDo(KeyByFirstLetter)

{KV<S, Storm>, KV<F, Flink>,
KV<A, Apex>, KV<S, Spark>, ...}

Element-wise transforms: ParDo (Go)

```
func firstLetter(w: string) string {
  return w[0]
}
lines := …
firstLetters := beam.ParDo(s, firstLetter, line)
```



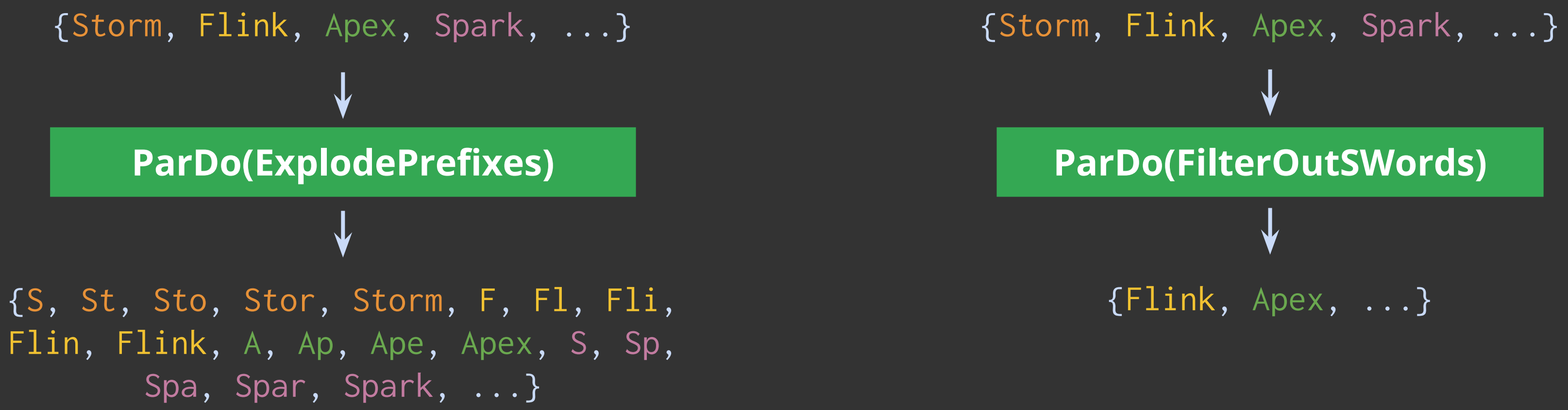
{Storm, Flink, Apex, Spark, ...}

ParDo(KeyByFirstLetter)

{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>, KV<S, Spark>, ...}

Element-wise transforms: ParDo

ParDo can output 1, 0 or many values for each input element



{Storm, Flink, Apex, Spark, ...}

ParDo(ExplodePrefixes)

{S, St, Sto, Stor, Storm, F, Fl, Fli,
Flin, Flink, A, Ap, Ape, Apex, S, Sp,
Spa, Spar, Spark, ...}

{Storm, Flink, Apex, Spark, ...}

ParDo(FilterOutSWords)

{Flink, Apex, ...}

Element-wise transforms: Friends of ParDo

The SDK includes other Element Wise Transforms for convenience

ParDo	General; 1-input to (0,1,many)-outputs; side-inputs and side-outputs
Filter	1-input to (0 or 1)-outputs
MapElements	1-input to 1-output
FlatMapElements	1-input to (0,1,many)-output
WithKeys	value -> KV(f(value), value)
Keys	KV(key, value) -> key
Values	KV(key, value) -> value

Element-wise transforms: Friends of ParDo

The SDK includes other Element Wise Transforms for convenience

ParDo	General; 1-input to (0,1,many)-outputs; side-inputs and side-outputs
Filter	1-input to (0 or 1)-outputs
MapElements	1-input to 1-output
FlatMapElements	1-input to (0,1,many)-output
WithKeys	value -> KV(f(value), value)
Keys	KV(key, value) -> key
Values	KV(key, value) -> value

```
// Example filter Java
input.apply(Filter
  .byPredicate((String w) -> w.startsWith("S"));
```

Element-wise transforms: Friends of ParDo

The SDK includes other Element Wise Transforms for convenience

ParDo	General; 1-input to (0,1,many)-outputs; side-inputs and side-outputs
Filter	1-input to (0 or 1)-outputs
MapElements	1-input to 1-output
FlatMapElements	1-input to (0,1,many)-output
WithKeys	value -> KV(f(value), value)
Keys	KV(key, value) -> key
Values	KV(key, value) -> value

```
# Example filter, Python
input | beam.Filter(lambda w: w[0] == ‘S’)
```

Element-wise transforms: Friends of ParDo

The SDK includes other Element Wise Transforms for convenience

ParDo	General; 1-input to (0,1,many)-outputs; side-inputs and side-outputs
Filter	1-input to (0 or 1)-outputs
MapElements	1-input to 1-output
FlatMapElements	1-input to (0,1,many)-output
WithKeys	value -> KV(f(value), value)
Keys	KV(key, value) -> key
Values	KV(key, value) -> value

```
// Example filter, Go
filter.Include(s, input, func(s string) bool {
  return w[0] == ‘S’
})
```

Element-wise transforms: Friends of ParDo

The SDK includes other Element Wise Transforms for convenience

ParDo	General; 1-input to (0,1,many)-outputs; side-inputs and side-outputs
Filter	1-input to (0 or 1)-outputs
MapElements	1-input to 1-output
FlatMapElements	1-input to (0,1,many)-output
WithKeys	value -> KV(f(value), value)
Keys	KV(key, value) -> key
Values	KV(key, value) -> value

```
// MapElements Java
input.apply(MapElements
  .into(TypeDescriptors.kvs(TypeDescriptors.characters(),
                            TypeDescriptors.strings()))
  .via((String w) -> KV.of(w, w.charAt(0))
```

Element-wise transforms: Friends of ParDo

The SDK includes other Element Wise Transforms for convenience

ParDo	General; 1-input to (0,1,many)-outputs; side-inputs and side-outputs
Filter	1-input to (0 or 1)-outputs
MapElements	1-input to 1-output
FlatMapElements	1-input to (0,1,many)-output
WithKeys	value -> KV(f(value), value)
Keys	KV(key, value) -> key
Values	KV(key, value) -> value

```
# MapElement Python
input | beam.Map(lambda w: (w, w[0]))
```

Element-wise transforms: Friends of ParDo

The SDK includes other Element Wise Transforms for convenience

ParDo	General; 1-input to (0,1,many)-outputs; side-inputs and side-outputs
Filter	1-input to (0 or 1)-outputs
MapElements	1-input to 1-output
FlatMapElements	1-input to (0,1,many)-output
WithKeys	value -> KV(f(value), value)
Keys	KV(key, value) -> key
Values	KV(key, value) -> value

```
// FlatMapElements Java
input.apply(FlatMapElements
  .into(TypeDescriptors.strings())
  .via((String w) -> populateSuffixes(w)));
```

Element-wise transforms: Friends of ParDo

The SDK includes other Element Wise Transforms for convenience

ParDo	General; 1-input to (0,1,many)-outputs; side-inputs and side-outputs
Filter	1-input to (0 or 1)-outputs
MapElements	1-input to 1-output
FlatMapElements	1-input to (0,1,many)-output
WithKeys	value -> KV(f(value), value)
Keys	KV(key, value) -> key
Values	KV(key, value) -> value

```
# FlatMapElement Python
input | beam.FlatMap(populateSuffixes)
```

Element-wise transforms: Friends of ParDo

The SDK includes other Element Wise Transforms for convenience

ParDo	General; 1-input to (0,1,many)-outputs; side-inputs and side-outputs
Filter	1-input to (0 or 1)-outputs
MapElements	1-input to 1-output
FlatMapElements	1-input to (0,1,many)-output
WithKeys	value -> KV(f(value), value)
Keys	KV(key, value) -> key
Values	KV(key, value) -> value

```
// WithKeys Java
input.apply(WithKeys.
  .of((String w) -> w.charAt(0))
  .withKeyType(TypeDescriptors.characters()))
```

Element-wise transforms: Friends of ParDo

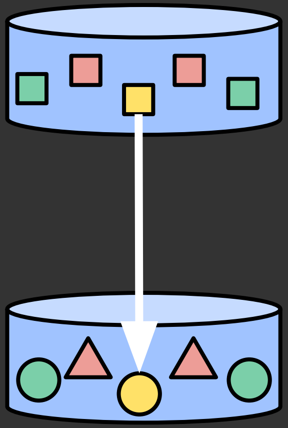
The SDK includes other Element Wise Transforms for convenience

ParDo	General; 1-input to (0,1,many)-outputs; side-inputs and side-outputs
Filter	1-input to (0 or 1)-outputs
MapElements	1-input to 1-output
FlatMapElements	1-input to (0,1,many)-output
WithKeys	value -> KV(f(value), value)
Keys	KV(key, value) -> key
Values	KV(key, value) -> value

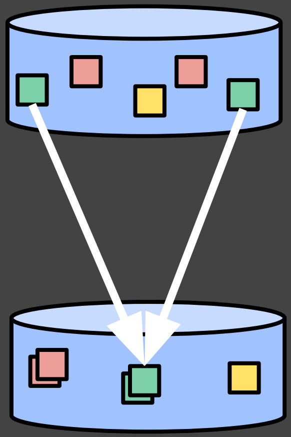
```
// Keys
input.apply(Keys.create())
```

```
// Values
input.apply(Values.create())
```

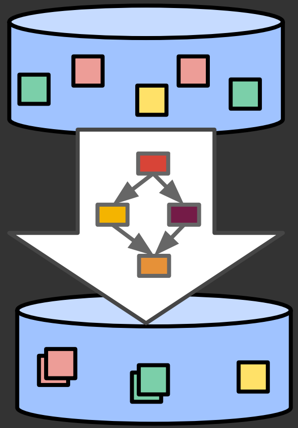
What are you computing?



Element-Wise (map)



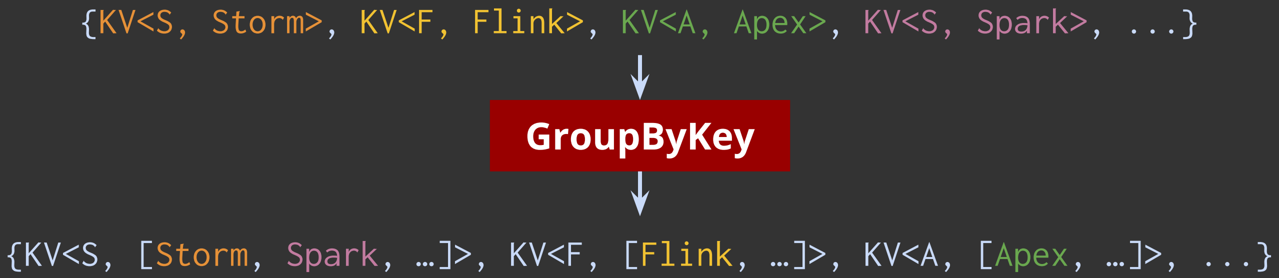
Aggregating (reduce)



Composite (reusable combinations)

Grouping transforms: GroupByKey

Takes a PCollection of key-value pairs and groups all values with the same key



```
{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>, KV<S, Spark>, ...}
```

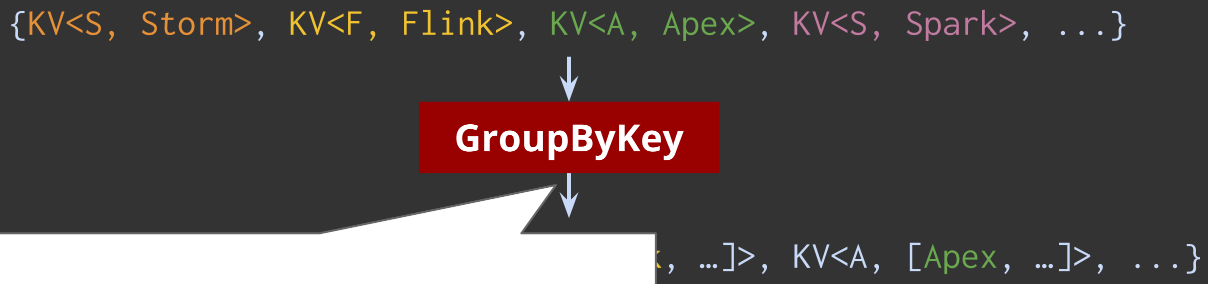
GroupByKey

```
{KV<S, [Storm, Spark, …]>, KV<F, [Flink, …]>, KV<A, [Apex, …]>, ...}
```

How can we use GroupByKey to compute the most common value for each key?

Grouping transforms: GroupByKey

Takes a PCollection of key-value pairs and groups all values with the same key



{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>, KV<S, Spark>, ...}

GroupByKey

, …]>, KV<A, [Apex, …]>, ...}

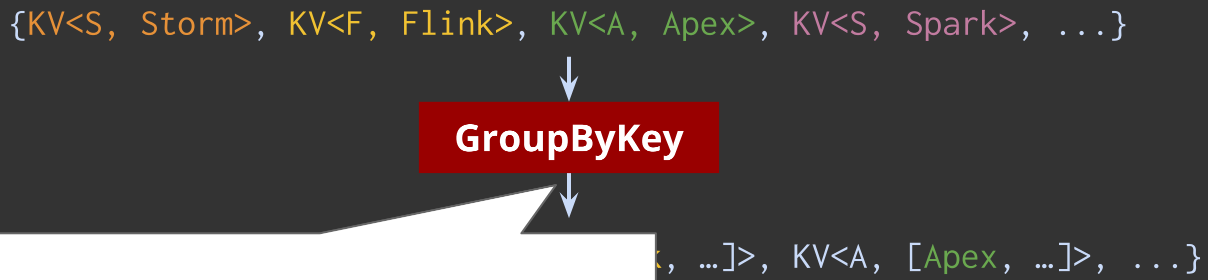
```
input.apply(GroupByKey.<Character, String>create())
```

How can

on value for each key?

Grouping transforms: GroupByKey

Takes a PCollection of key-value pairs and groups all values with the same key



{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>, KV<S, Spark>, ...}

GroupByKey

, …]>, KV<A, [Apex, …]>, ...}

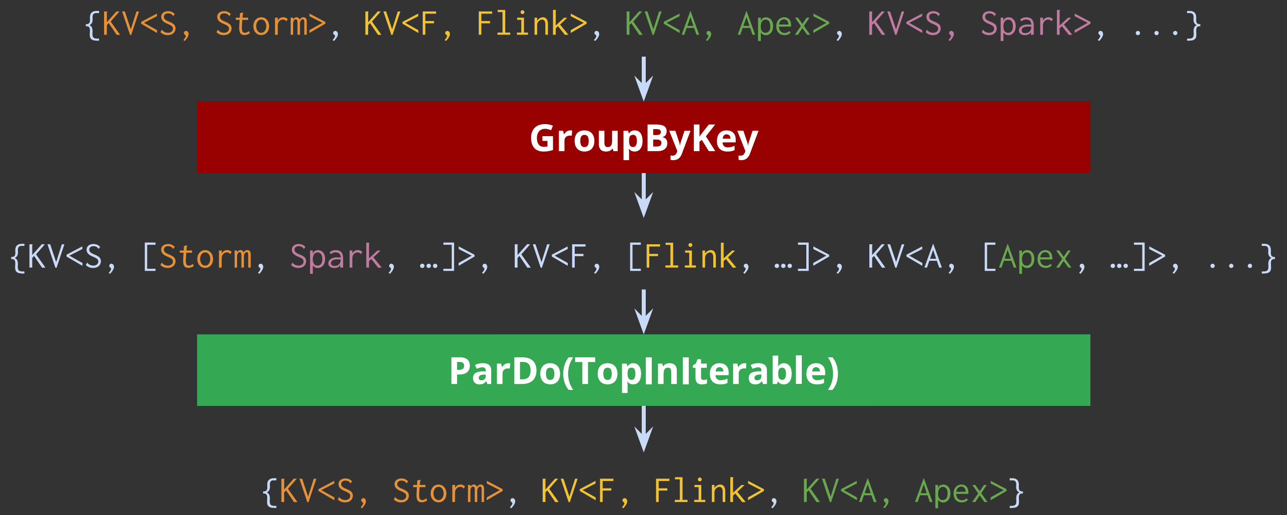
```
input | beam.GroupByKey()
```

How can

on value for each key?

Grouping transforms: GroupByKey

Computing the most common value for each key



```
{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>, KV<S, Spark>, ...}
```

GroupByKey

```
{KV<S, [Storm, Spark, …]>, KV<F, [Flink, …]>, KV<A, [Apex, …]>, ...}
```

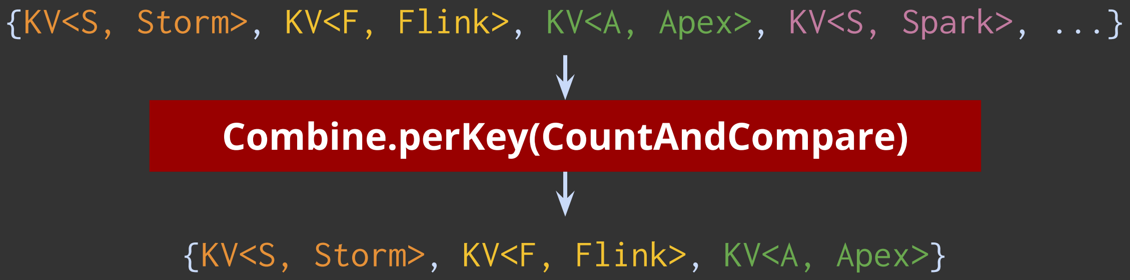
ParDo(TopInIterable)

```
{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>}
```

TopInIterable processes KV<K, Iterable<String>> and has to look at all of the values for each key...

Grouping transforms: GroupByKey

GroupByKey followed by ParDo can often be simplified (and optimized!): Combine



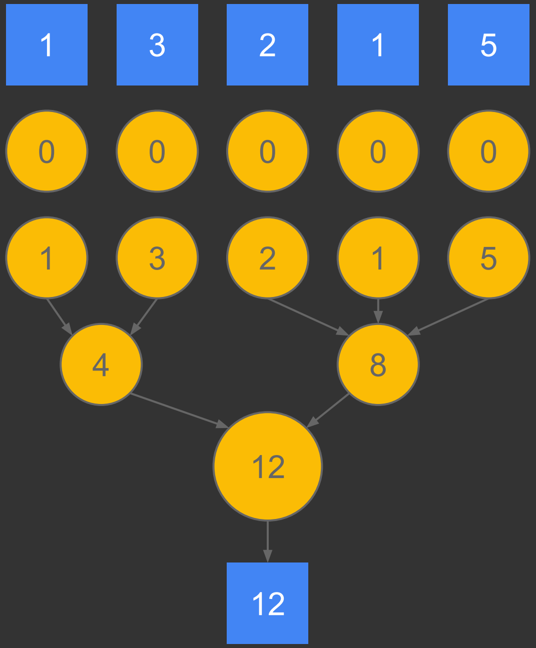
```
{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>, KV<S, Spark>, ...}
```

Combine.perKey(CountAndCompare)

```
{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>}
```

Grouping transforms: Combine

CountAndCompare is a CombineFn that counts words and then extracts the top-K.
You can write your own for any operation that is associative & commutative.



Initialize accumulators

Add Input to each accumulator

Merge accumulators

Merge accumulators (again)

Extract output (from accumulator)

Grouping transforms: Built-in CombineFns

The SDK includes many pre-defined Combiners:

Top.perKey(1)

Min.longsPerKey()

Count.perKey()

Max.longsPerKey()

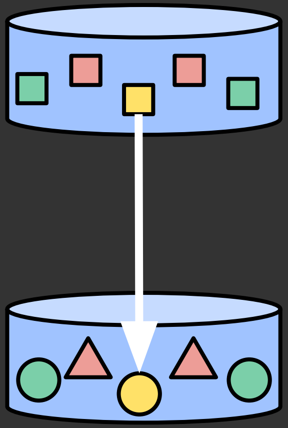
Sum.longsPerKey()

Mean.longsPerKey()

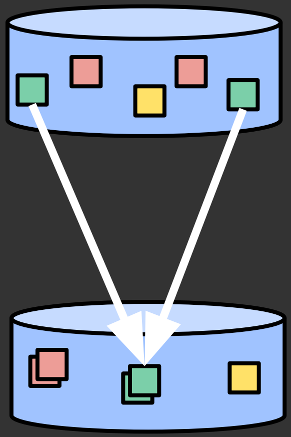
ApproximateQuantiles.perKey(5)

ApproximateUnique.perKey(10)

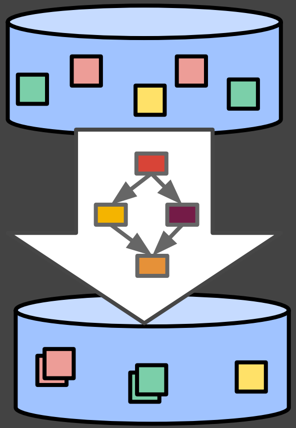
Writing a pipeline = Gluing pieces together



Element-Wise
(map)



Aggregating
(reduce)



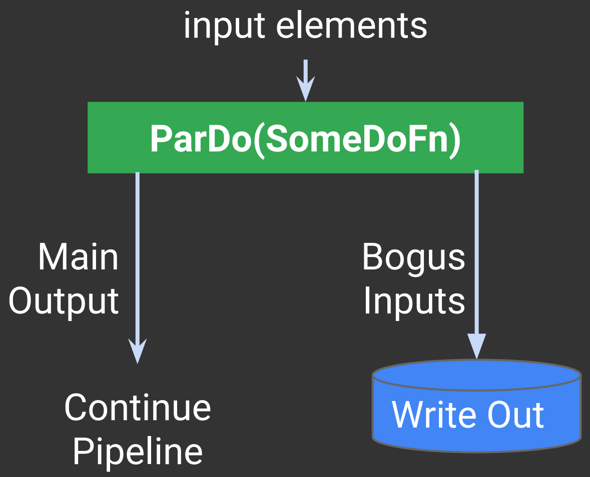
Composite
(reusable combinations)

Multiple outputs

ParDos can produce multiple outputs
Example usage: dead-letter pattern

A main output containing all the successfully processed results

A secondary output containing all the elements that failed to be processed



Example: Multiple outputs (Java)

```java
final TupleTag<Output> successTag = new TupleTag<>() {};
final TupleTag<Input> deadLetterTag = new TupleTag<>() {};

PCollection<Input> input = ...;
PCollectionTuple outputTuple = input.apply(ParDo
    .withOutputTags(successTag, TupleTagList.of(deadLetterTag))
    .of(new DoFn<Input, Output>() {
        @ProcessElement
        public void processElement(@Element InputT e, MultiOutputReceiver o) {
          try {
            o.output(successTag, validateElement(e));
          } catch (Exception e) {
            o.output(deadLetterTag, e);
          }
        }});
PCollection<Output> success = outputTuple.get(successTag);
PCollection<Input> deadLetters = outputTuple.get(deadLetterTag);
```

Example: Multiple outputs (Python)

```
Class Process(beam:DoFn):
 def process(self, element):
  try:
   yield value.TaggedOutput(‘success’, validateElement(element))
  Except:
   yield value.TaggedOutput(‘failure’, element)
success, failures = input | beam.ParDo(Process()).withOutputs(“success”, “failure”)
```

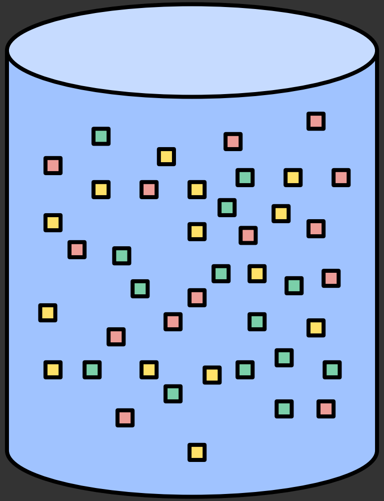
Example: Multiple outputs (Go)

```
success, failures := beam.ParDo2(s,
  func(input: InputT, emitSuccess, emitFailure func(InputT)) {
  if (validateElement(input) != nil) {
    emitSuccess(input)
 } else {
   emitFailure(input)
}
```

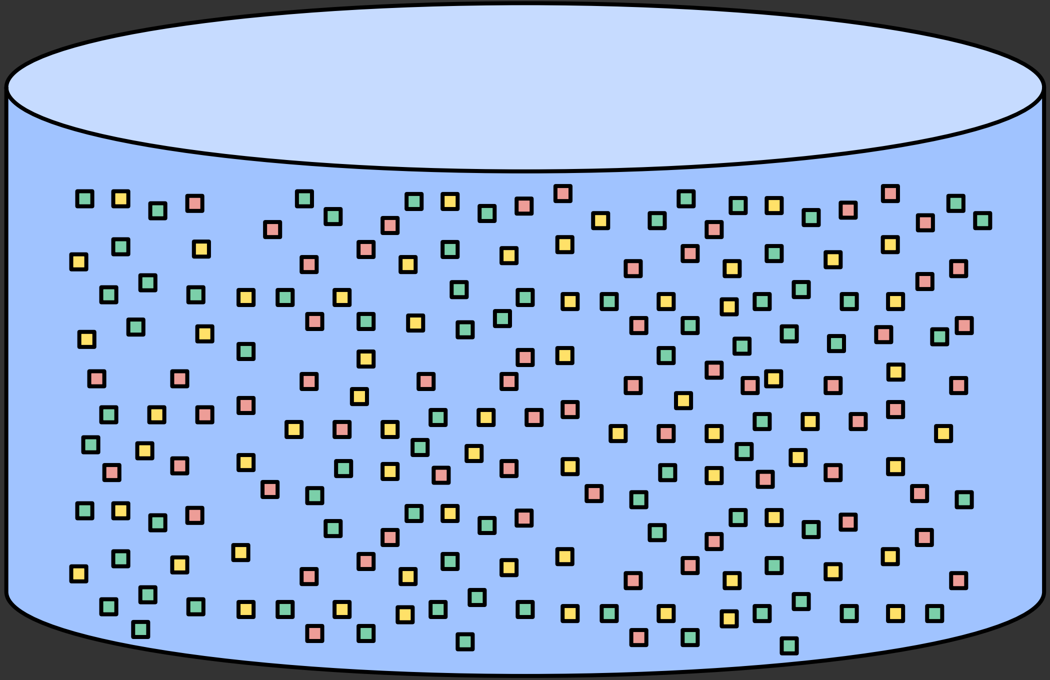
03 Windowing & Time

Where in event time?

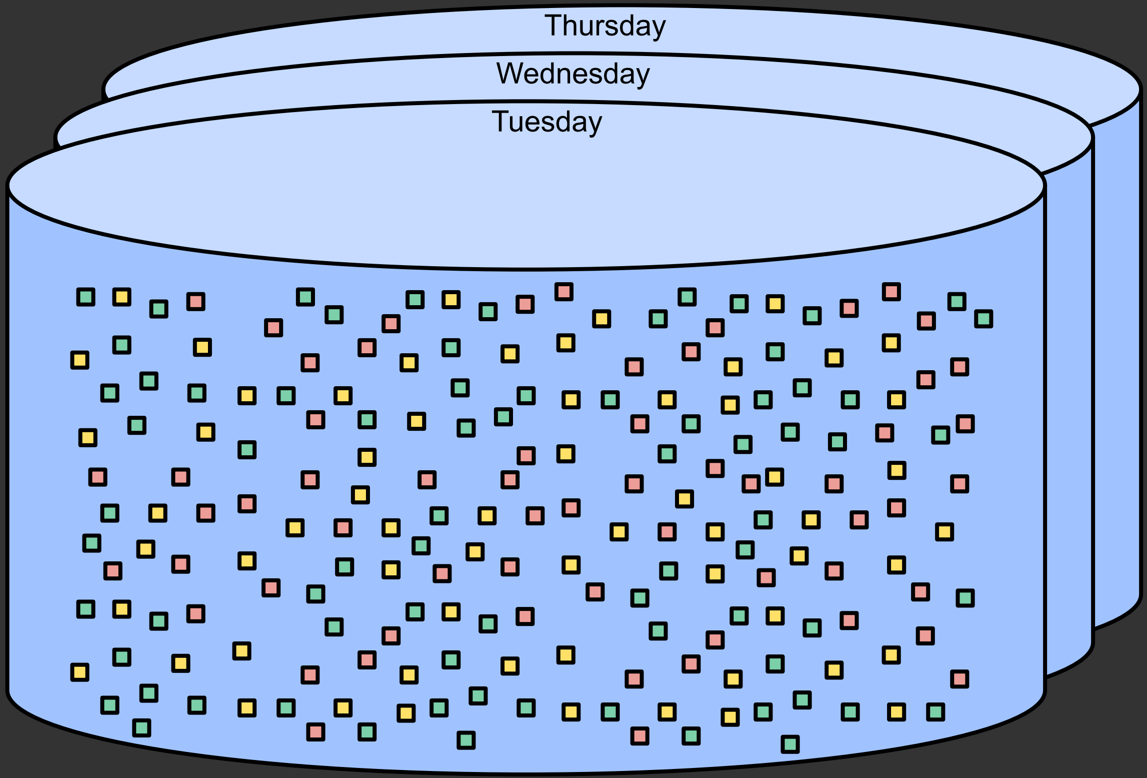
Data...



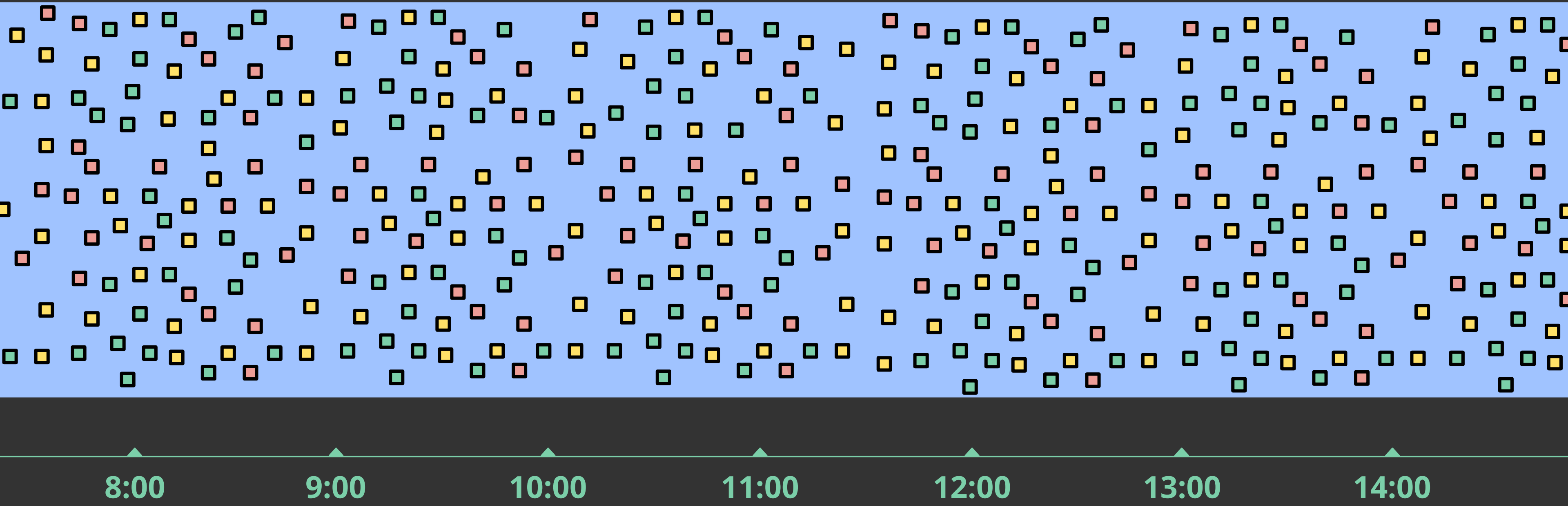
...can be big...



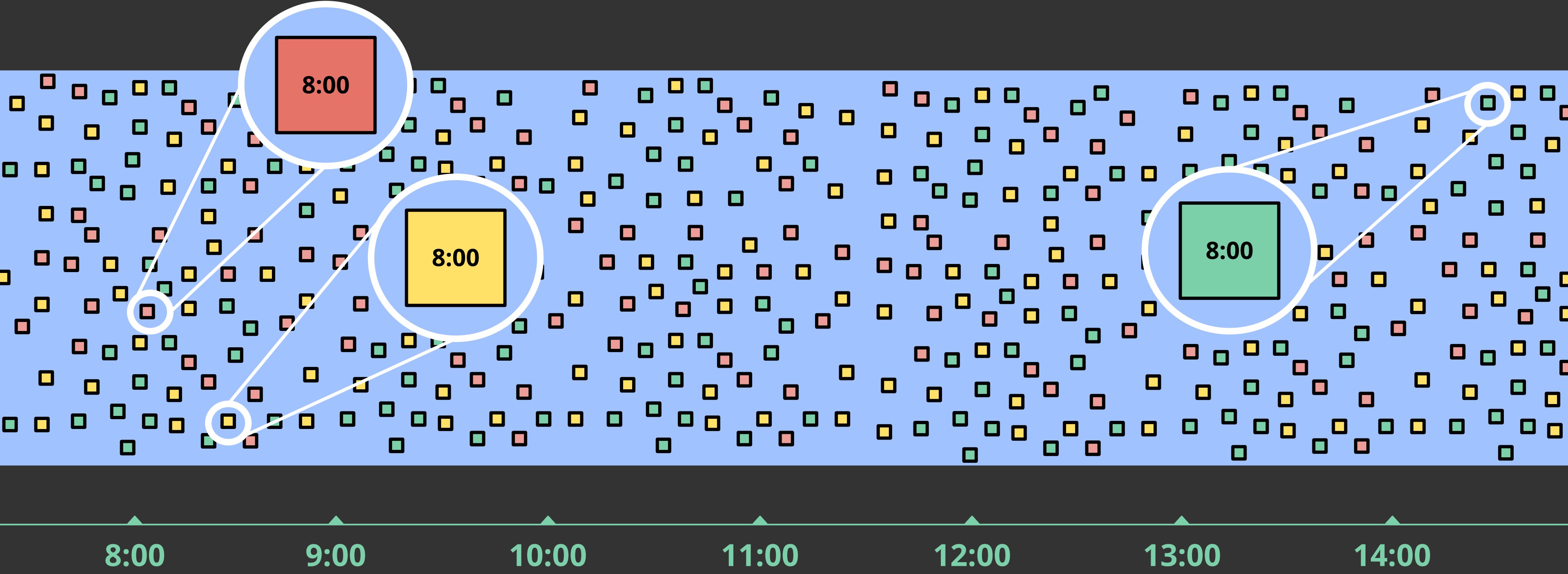
...really, really big...



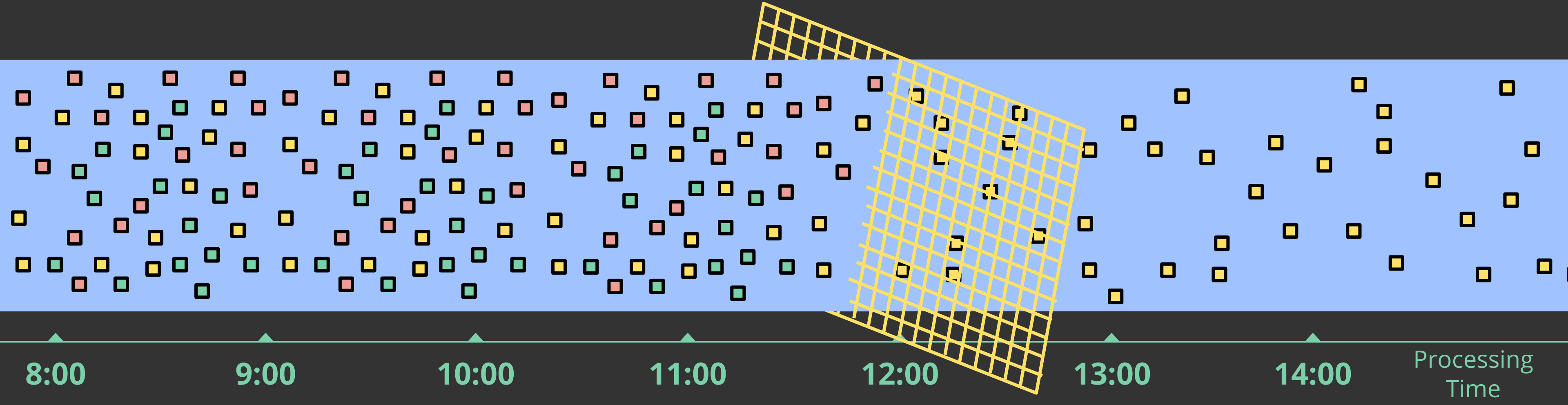
...maybe infinitely big...



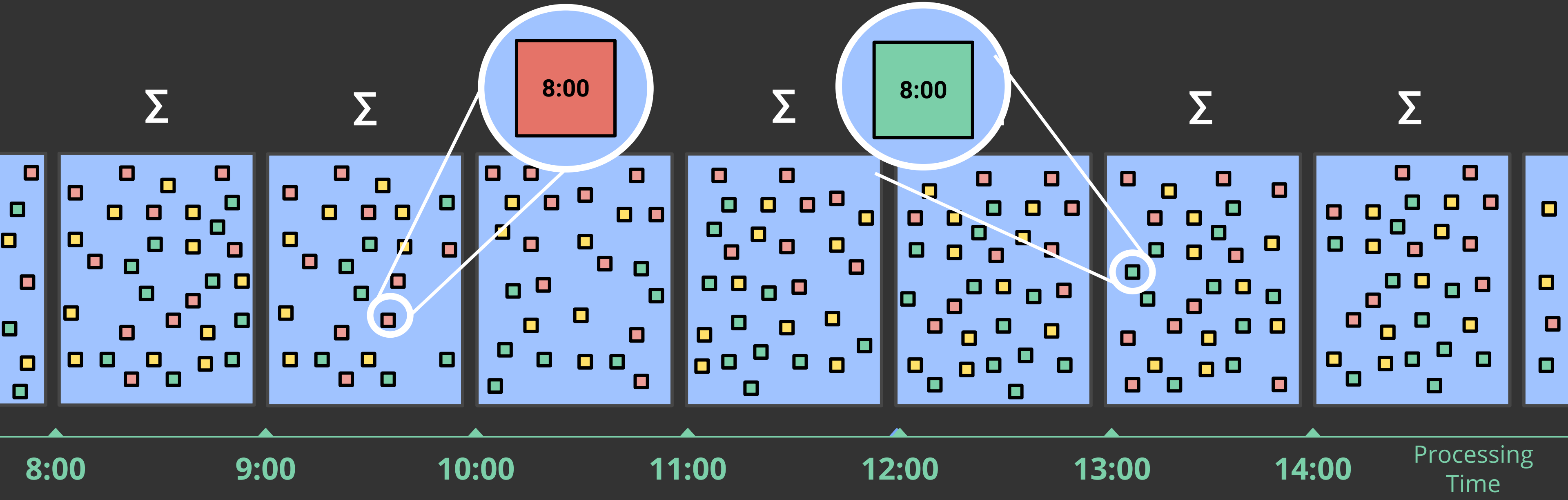
...with unknown delays.



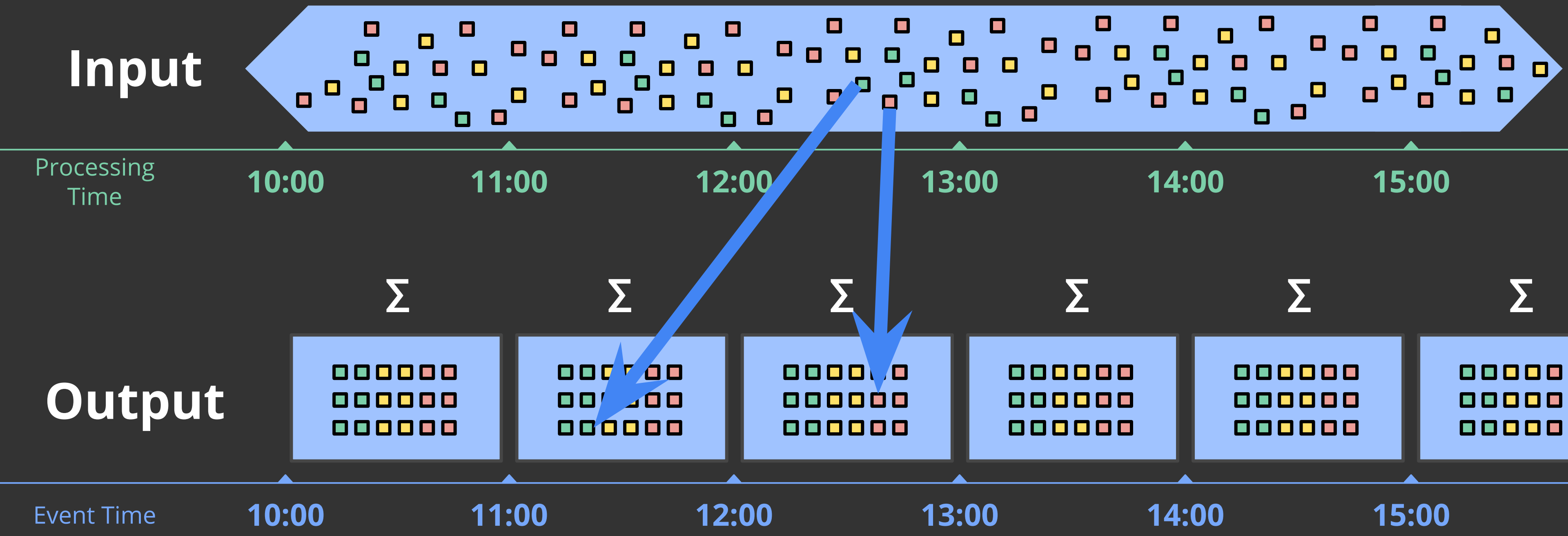
Element-wise transforms



Grouping via processing-time windows

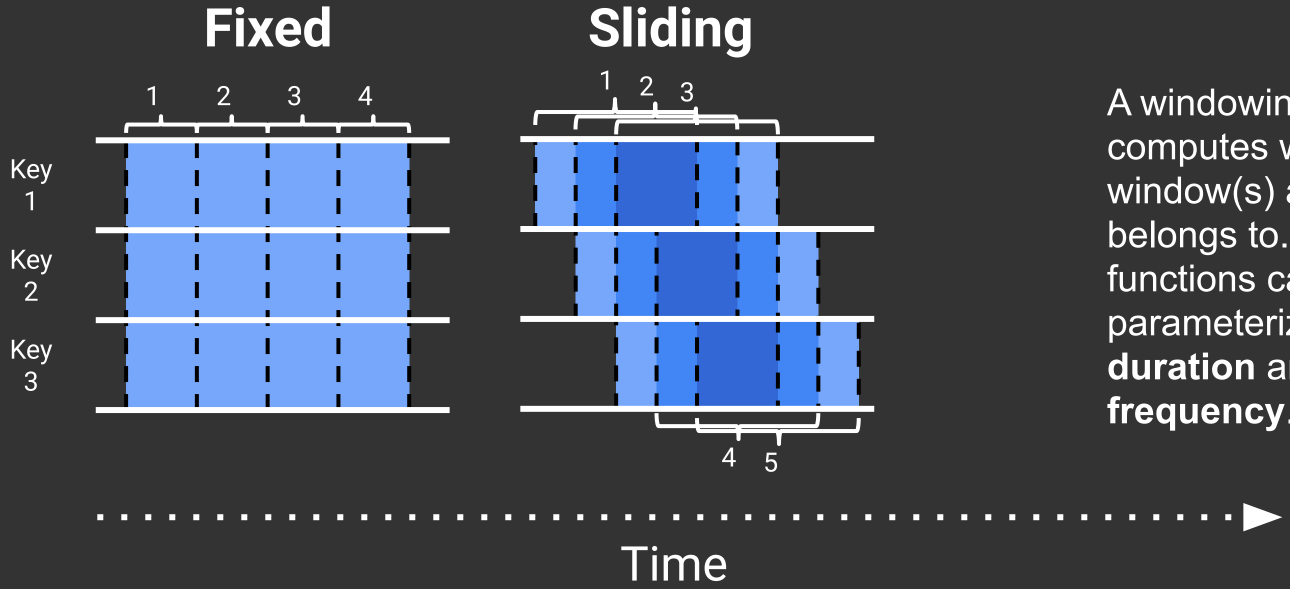


Grouping via event-time windows



What is windowing?

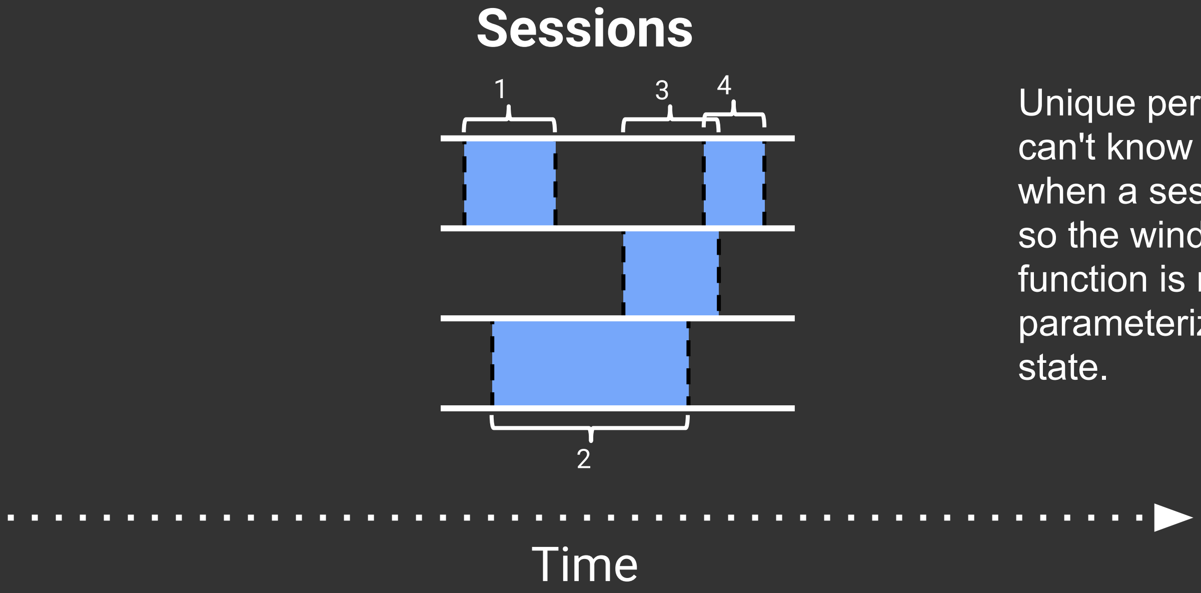
Windowing divides data into event-time-based finite chunks.



A windowing function computes which window(s) an element belongs to. Temporal functions can be parameterized with **duration** and **frequency**.

Often required when doing aggregations over unbounded data.

What about data-dependent windowing?

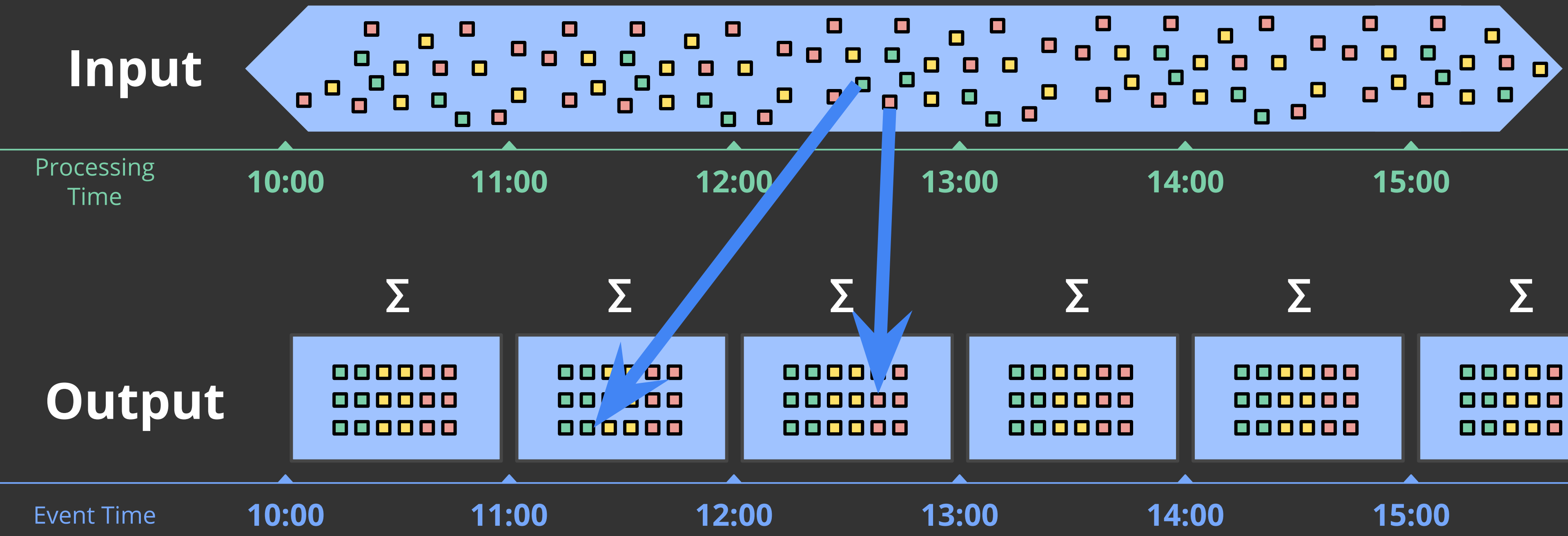


Unique per key - you can't know a priori when a session ends, so the windowing function is now also parameterized by state.

04 Triggers & Streaming

When in processing time are results emitted?

Streaming: Unbounded PCollections

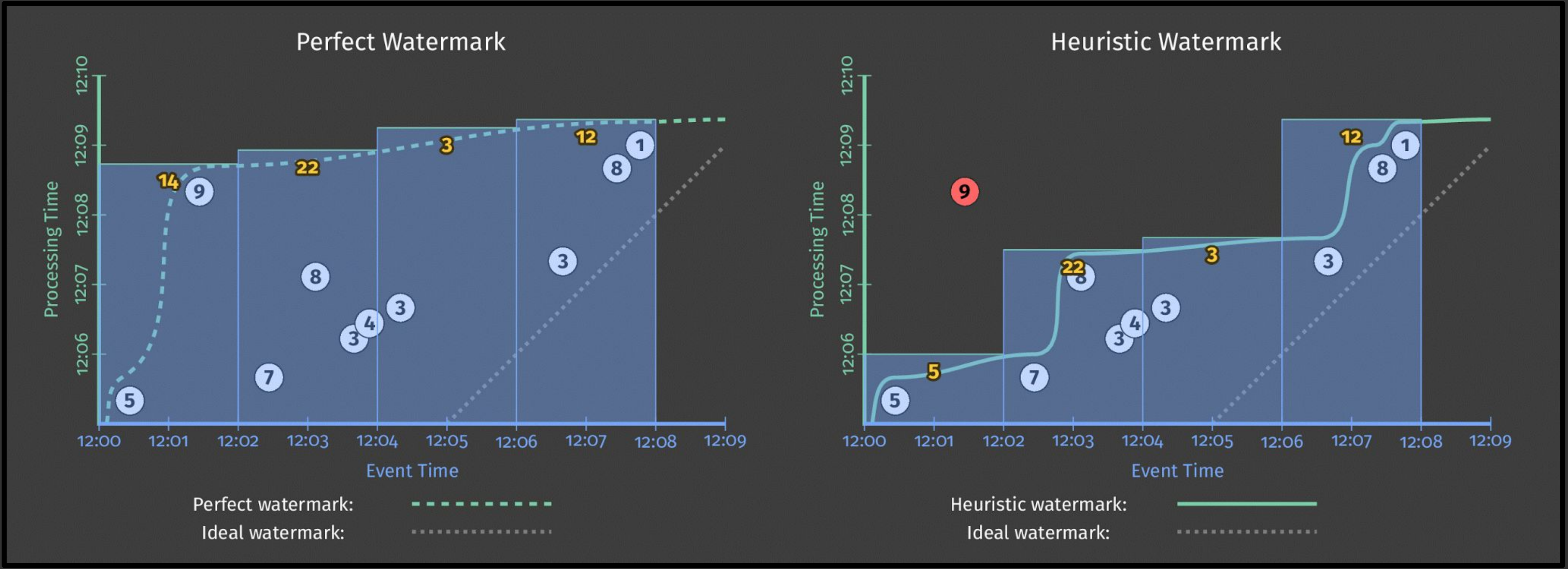


Windowing specifies ***where*** events are aggregated in event time, but ***when*** are events emitted in processing time?

When: triggering at the watermark

```
PCollection<KV<String, Integer>> scores = input
    .apply(Window.into(FixedWindows.of(Minutes(2))))
    .apply(Sum.integersPerKey());
```

When: triggering at the watermark



Triggers control ***when*** the aggregation is output.

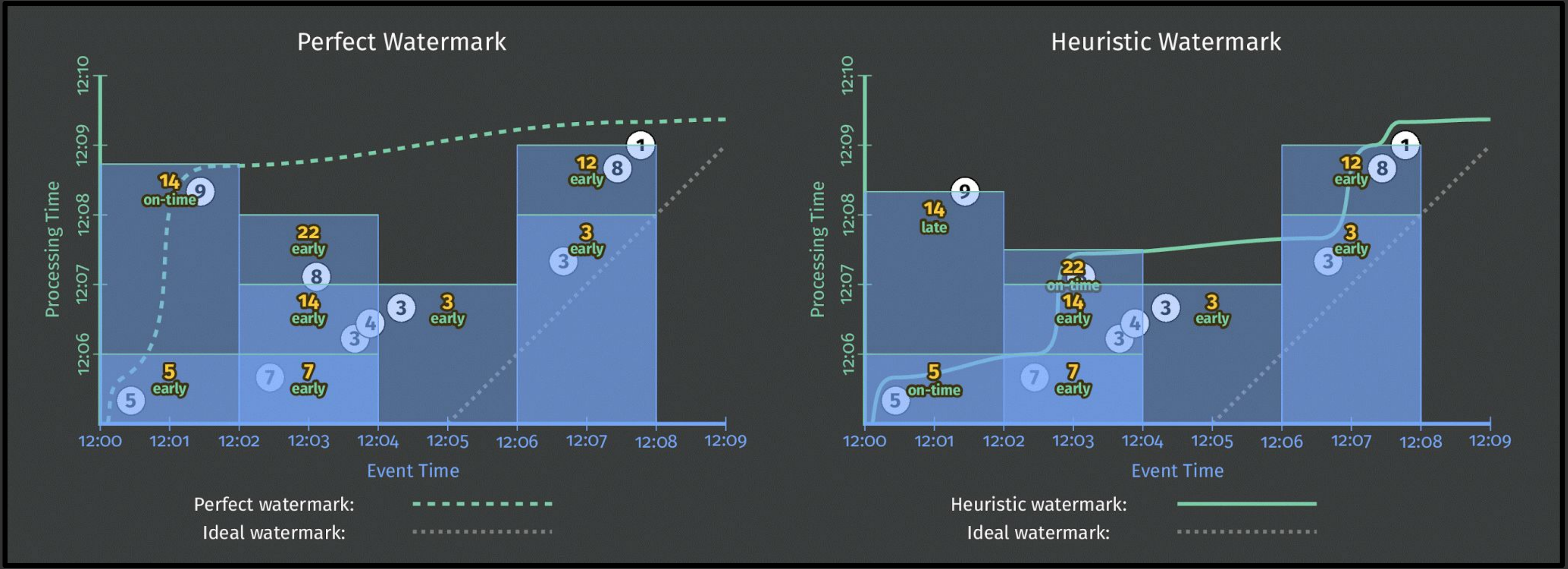
The default is “*when the watermark passes the end of the window*”.

This is the same as “*when we estimate the window is complete*”

When: early & late firings

```
PCollection<KV<String, Integer>> scores = input
   .apply(Window.into(FixedWindows.of(Minutes(2))
            .triggering(AtWatermark()
              .withEarlyFirings(AtPeriod(Minutes(1)))
              .withLateFirings(AtCount(1))))
  .apply(Sum.integersPerKey());
```

When: early & late firings



Speculative triggers provide early updates *before* the watermark passes.

Watermark triggers provide on-time updates when input is believed complete.

Late triggers provide late updates when data arrive *after* the watermark (late data).

Other kinds of triggers

Element Count
Output after at least N elements

Processing Time
Output after at least N minutes

Combinators
Early/on-time/late
After all of these
After any of these
After each of these in order
etc.

Together these can be used for fine-grained control of output

For example:

- Early: every minute
- On-Time: when watermark predicts the window is complete
- Late: after every element

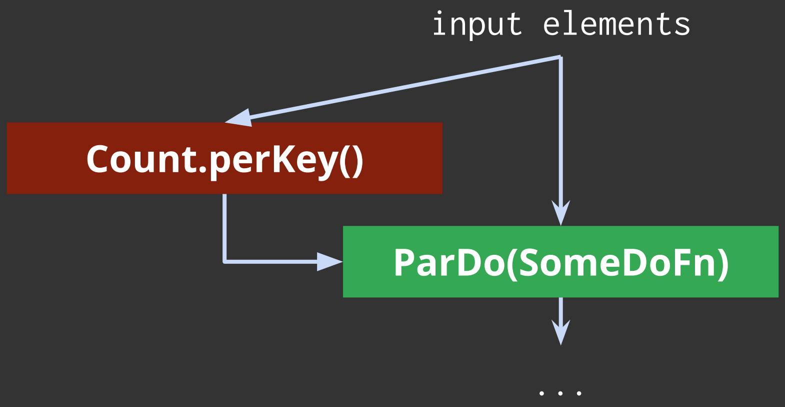
05 Side Inputs

Side inputs

ParDos can receive extra inputs “on the side”

For example broadcast the count of elements to the processing of each element

Side inputs are computed (and accessed) per-window



Example: ParDo with side inputs

```
PCollection<String> words = ...; // the input PCollection
PCollection<Integer> wordLengths = ...;  // map words to their lengths

// Create a PCollectionView (singleton in this case).
// See also View.asList, View.asMap, etc.
final PCollectionView<Integer> maxWordLengthView =
     wordLengths.apply(Combine.globally(new Max.MaxIntFn()).asSingletonView());

// Apply a ParDo that takes maxWordLengthView as a side input, and left pads words.
PCollection<String> rightPaddedWords = words.apply(ParDo
    .withSideInputs(maxWordLengthView).of(new DoFn<String, String>() {
        @ProcessElement
        public void processElement(ProcessContext c) {
         int length = c.sideInput(maxWordLengthView);
         String format = “%1-” + length + “s”;
         c.output(String.Format(format, c.element()));
        }}));
```

Scala API! Scio

```
sc.textFile(input)
  .map { w => w.trim }  // trim whitespace.
  .filter { w => w.nonEmpty }  // filter out empty lines.
  .flatMap(_.split(“[^a-ZA-Z’]+”).filter(_.nonEmpty)) // split lines
  .countByValue
  .map(t => t._1 + “:” t._2)  // format output word:count
  .saveAsTextFile(output)
```

Thank you!