Learn stream processing with Apache Beam (incubating)

https://goo.gl/3O5sZi

Tyler Akidau
Google
@takidau

Slava Chernyak
Google

Sandeep Deshmukh
DataTorrent
@sandeep_pitamah

Dan Halperin
Google
@dhalperi

Aljoscha Krettek
data Artisans
@aljoscha

Strata Singapore 2016

Schedule

13:00 - 13:30
- 00 Finish up pre-work

13:30 - 14:15
- 01 Introduction
- 02 Writing a Pipeline
  - Exercise 1: Batch
  - Exercise 2: Batch on other runners

14:15 - 15:00
- 03 Windowing & Time
  - Exercise 3: Batch Windowing

15:00 - 15:30
- 04 Break

16:00 - 16:30
- 05 Triggers and Streaming
  - Exercise 4: Streaming

16:30 - 17:00
- 06 Side Inputs & Outputs
  - Exercise 5: Spam Detection

Prework
(http://tiny.jesse-anderson.com/beamtutorial)

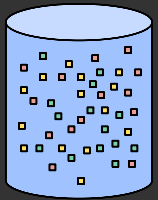
- Install Java 8
- Follow the instructions in the README
- Install IDE (Eclipse or IntelliJ)
- Import the project into Eclipse or IntelliJ

01 Introduction

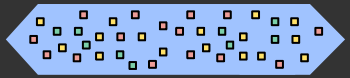
The Apache Beam programming model

What is part of Apache Beam?

One Model, Multiple Modes



Batch



Streaming

Multiple SDKs

Java

Python

Multiple Runners

Direct: local for testing

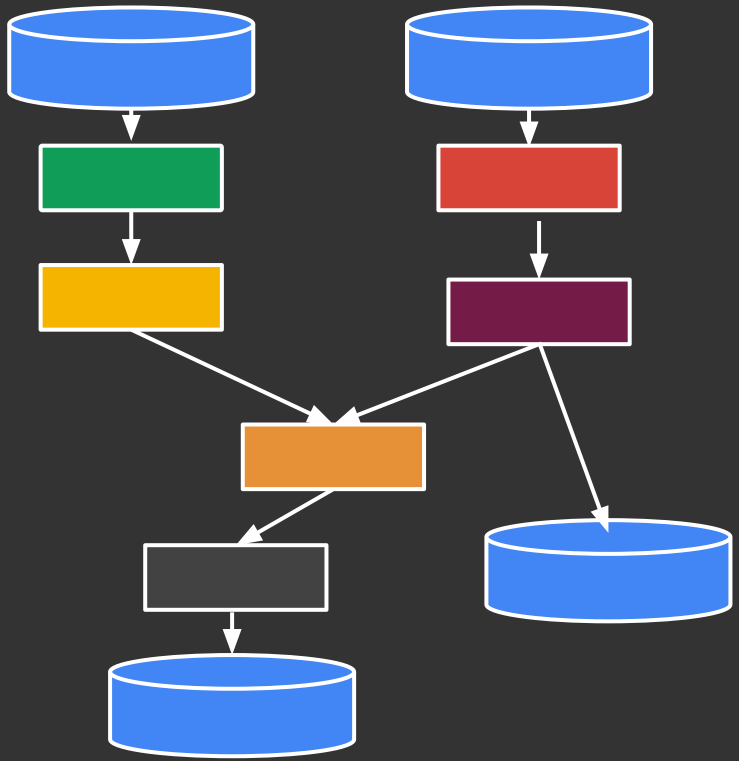
Apache Apex: local, on-premise, cloud

Apache Flink: local, on-premise, cloud

Apache Spark: local, on-premise, cloud

Cloud Dataflow: fully managed service on Google Cloud

What is a pipeline?



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

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

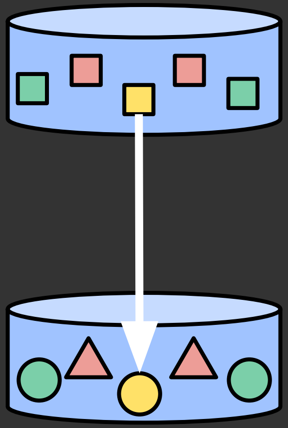
When = Watermarks + Triggers

How = Accumulation

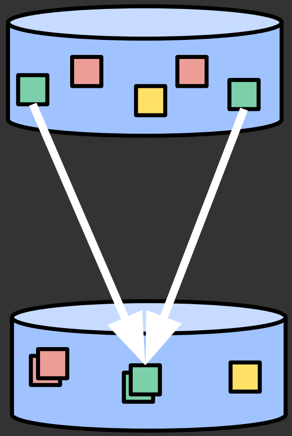
02 Writing a pipeline

What results are calculated?

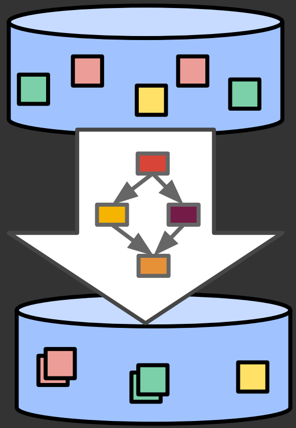
What are you computing?



Element-Wise
(map)

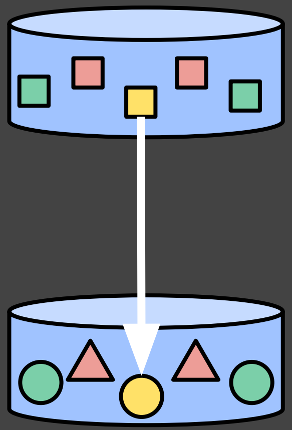


Aggregating
(reduce)

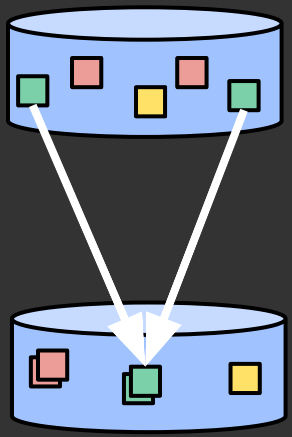


Composite
(reusable combinations)

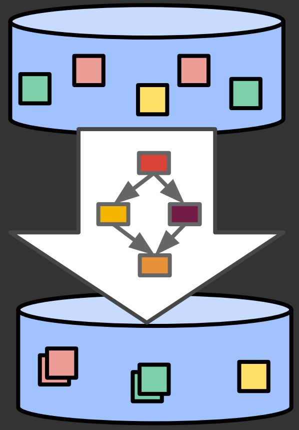
What are you computing?



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

```
PCollection<String> input = …;

// Example of a ParDo
input.apply(ParDo.of(
    new DoFn<String, KV<Char, String>>() {
  @ProcessElement
  public void processElement(ProcessContext c) {
    String word = c.element();
    Char firstLetter = word.charAt(0);
    c.output(KV.of(firstLetter, word));
  }
}));
```

{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

```
// Filter Java 8
input.apply(Filter
  .byPredicate((String w) -> w.startsWith("S"));

// Filter Java 7 and Java 8
input.apply(Filter.byPredicate(
  new SerializableFunction<String, Boolean>() {
    @Override
    public Boolean apply(String w) {
      return 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

```
// MapElements Java 8
input.apply(MapElements
  .via((String w) -> KV.of(w, w.charAt(0))
  .withOutputType(
    new TypeDescriptor<KV<Character, String>>() {}))

// MapElements Java 7
input.apply(MapElements.via(
  new SimpleFunction<String, KV<Character, String>>() {
    @Override
    public KV<Character, String> apply(String w) {
      return 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

```
// FlatMapElements Java 8
input.apply(FlatMapElements
  .via((String w) -> populateSuffixes(w))
  .withOutputType(new TypeDescriptor<String>() {}));

// FlatMapElements Java 7
input.apply(MapElements.via(
  new SimpleFunction<String, Iterable<String>>() {
    @Override
    public Iterable<String> apply(String w) {
      return 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

```
// WithKeys Java 8
input.apply(WithKeys.
  .of((String w) -> w.charAt(0))
  .withKeyType(new TypeDescriptor<Character>() {}))

// WithKeys Java 7
input.apply(MapElements.via(
  new SerializableFunction<String, Character>() {
    @Override
    public Character apply(String w) {
      return w.charAt(0);
    }
  }));
```

Element-wise transforms: Friends of ParDo

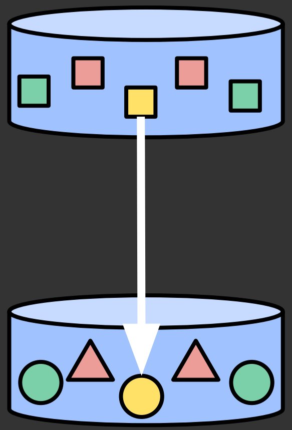
The SDK includes other Element Wise Transforms for convenience

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

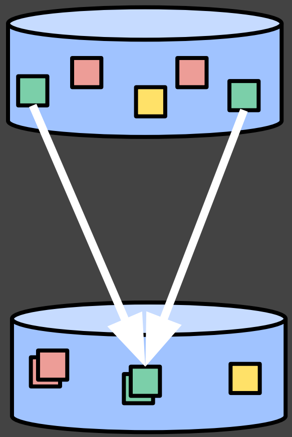
```
// Keys
input.apply(Keys.create())
```

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

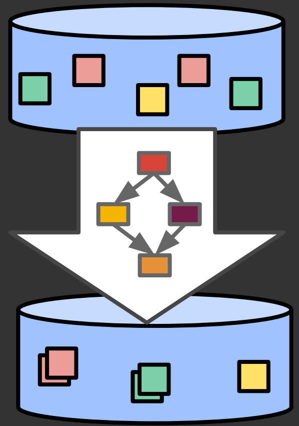
What are you computing?



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

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

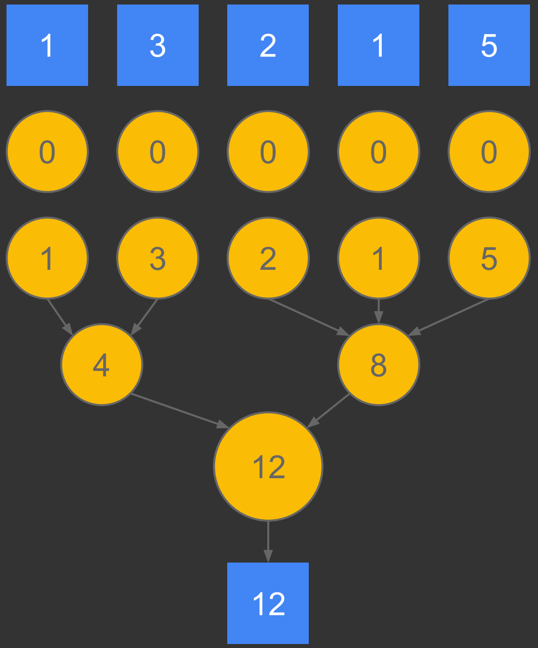
```
{KV<S, Storm>, KV<F, Flink>, KV<A, Apex>, KV<S, Spark>, ...}
```

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)

Exercise 1: Mobile Game Events

Events correspond to specific plays of our mobile game by a specific user

Each includes:

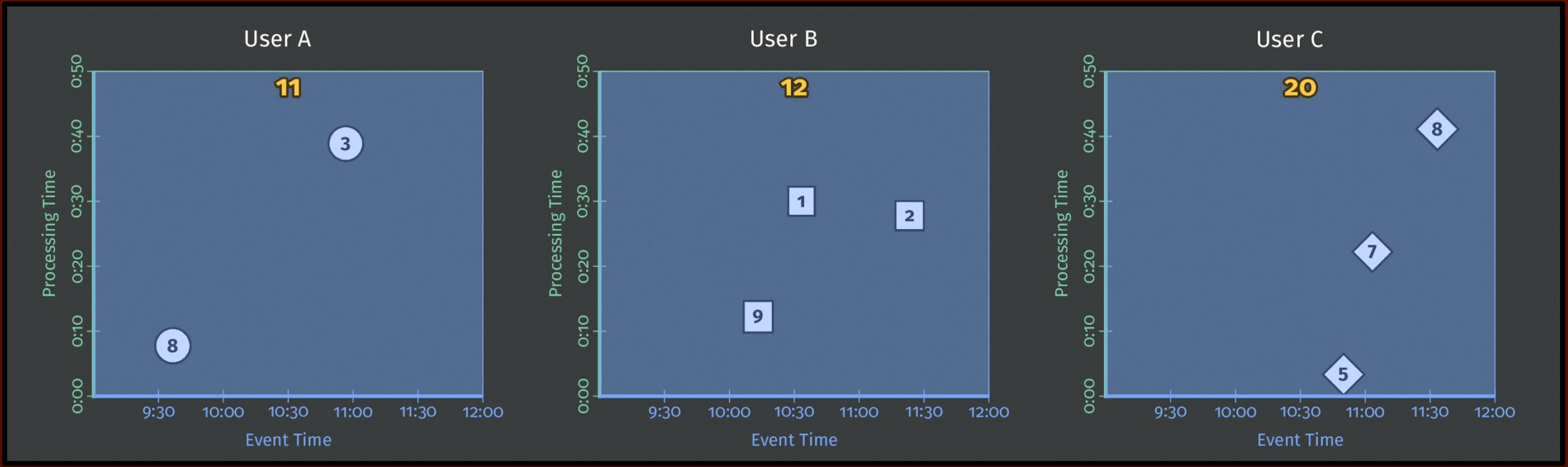
The **unique ID** of the user playing

The **team ID** the user is on

A **score** for that particular play

A **timestamp** that records when the play happened

Exercise 1: Mobile Game Events



Exercise 1: Implement ExtractAndSumScore

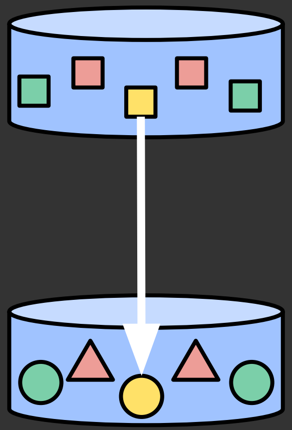
Overview

We're going to start with the **DirectRunner** -- this executes the pipeline locally (on your machine) and is great for testing

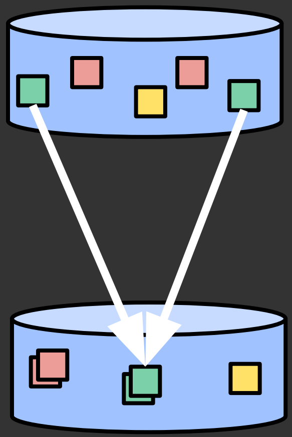
Instructions

1. Find the empty **ExtractAndSumScore** PTransform
2. Add code to extract the score keyed by **user ID** and then compute the sum for each user
3. Run your pipeline using the **DirectRunner**

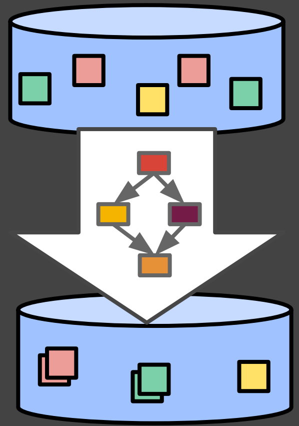
Writing a pipeline = Gluing pieces together



Element-Wise
(map)



Aggregating
(reduce)



Composite
(reusable combinations)

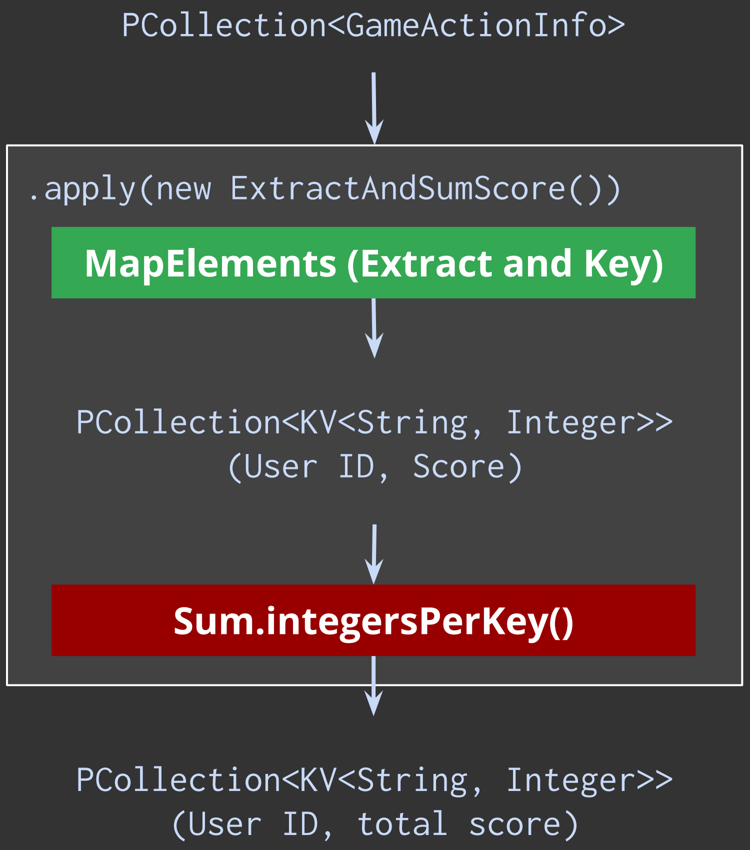
Composite Transforms

To simplify pipelines, multiple steps can be combined to make a composite transform

We’ve already seen some composite transforms

Creating higher level PTransforms is useful for organizing your pipeline

Each PTransform can be tested to ensure it behaves correctly



Pipeline Runners

Apache Beam Direct Runner	Apache Apex	Apache Flink	Apache Spark	Google Cloud Dataflow
Locally (on your machine) for testing and debugging	Locally, on-prem, or on a cloud service provider.	Locally, on-prem, or on a cloud service provider.	Locally, on-prem, or on a cloud service provider.	On Google Cloud as a managed service.

Exercise 2: UserScores on other runners

Overview

Ok, now let’s run that on a different runner...

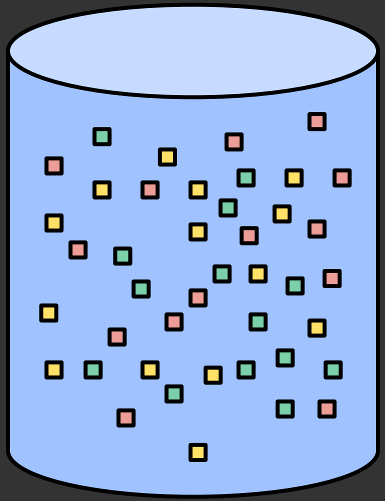
Instructions

1. Run your pipeline on the **runner of your choosing**
2. Compare the output to that from Exercise 1. **Does it look the same?**

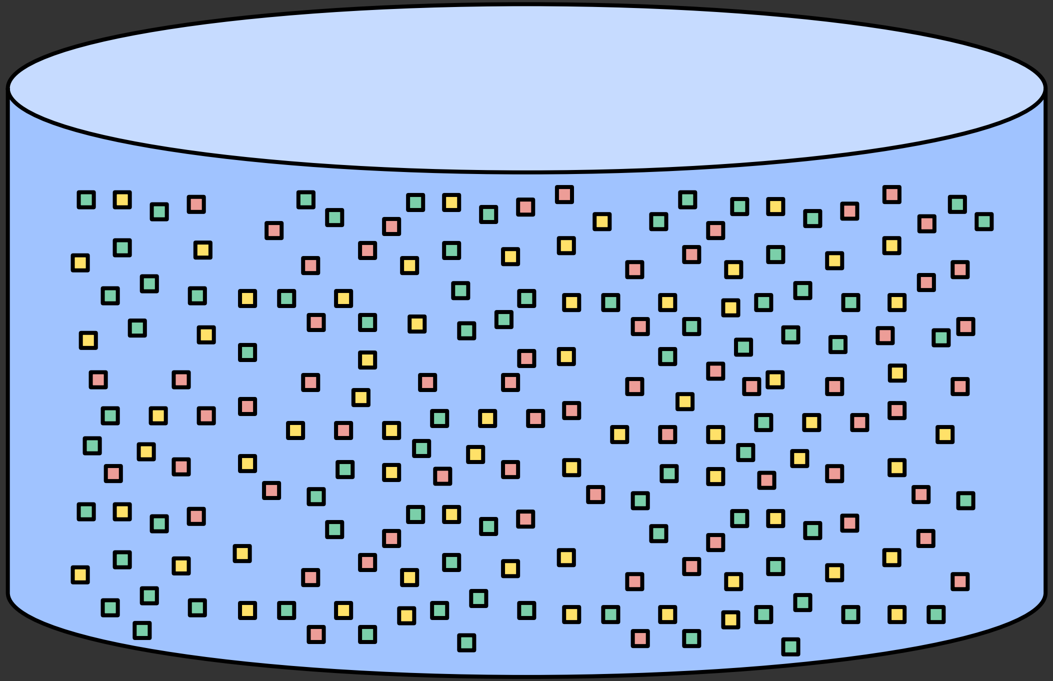
03 Windowing & Time

Where in event time?

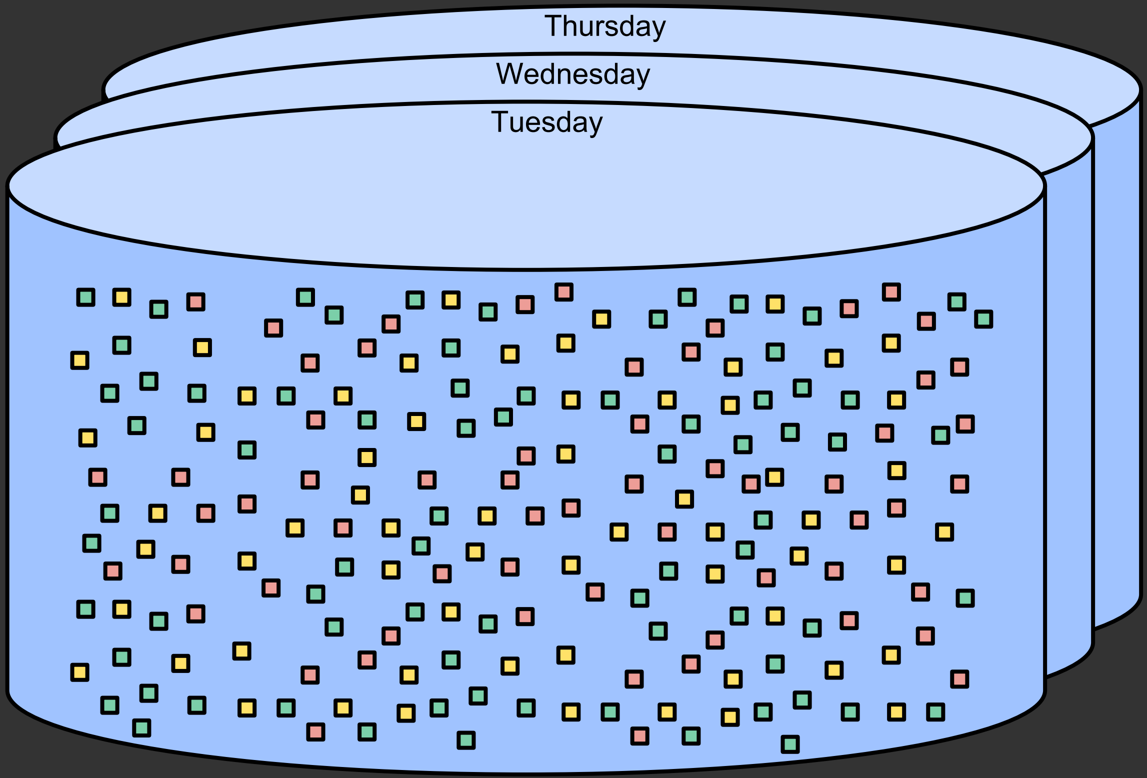
Data...



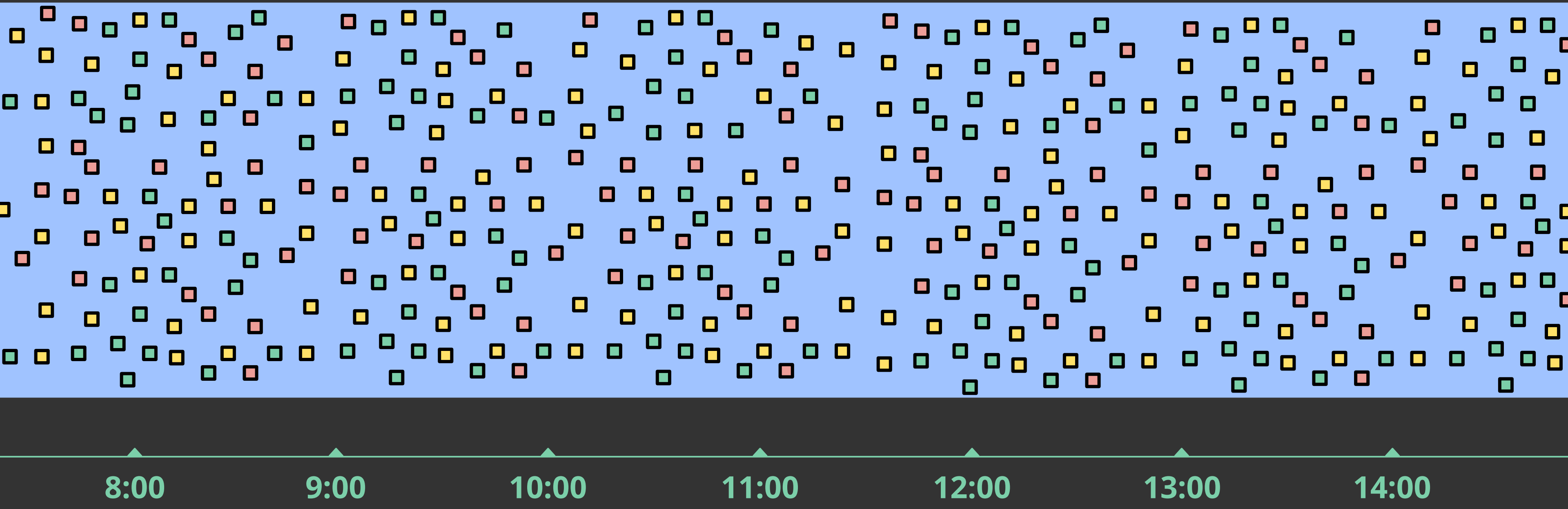
...can be big...



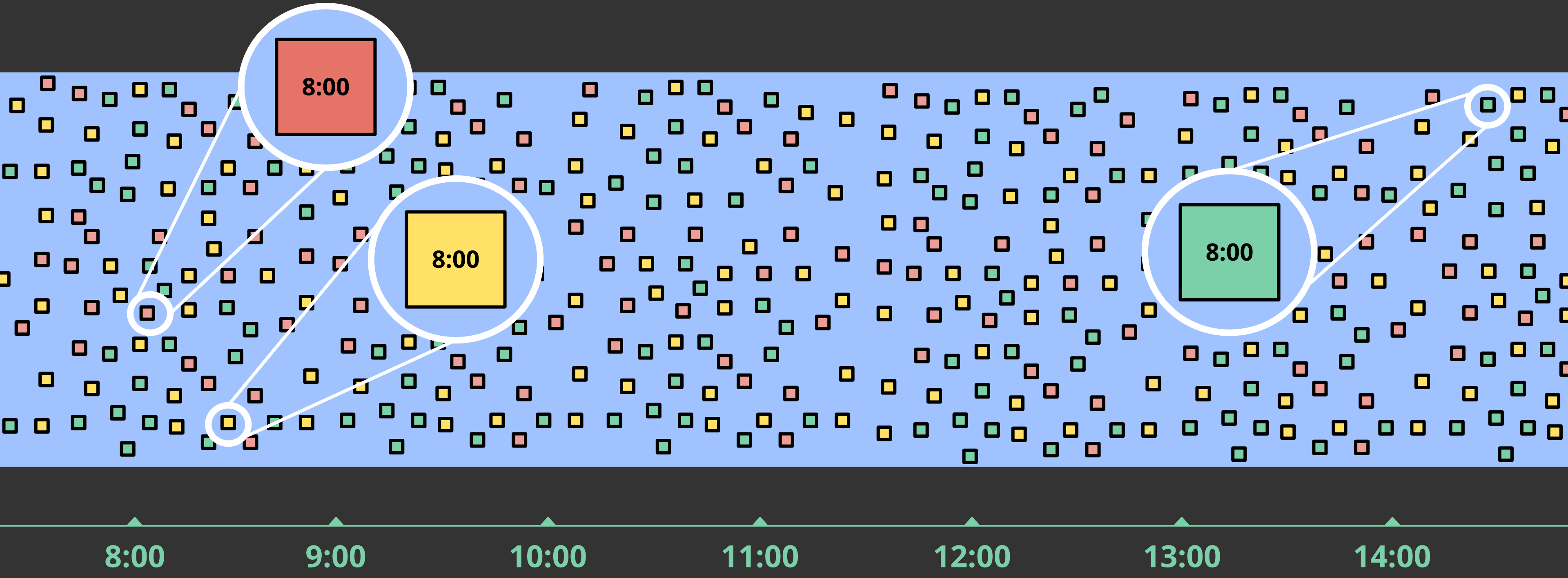
...really, really big...



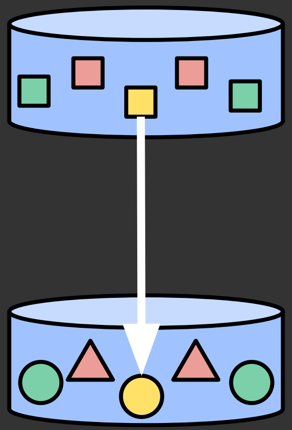
...maybe infinitely big...



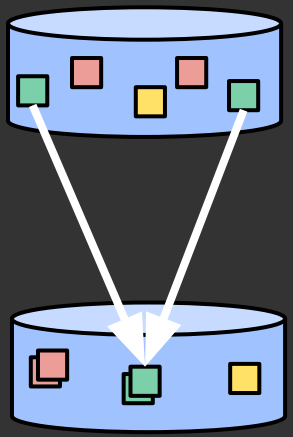
...with unknown delays.



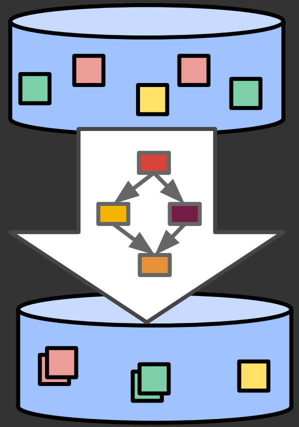
What are you computing?



Element-Wise
(map)

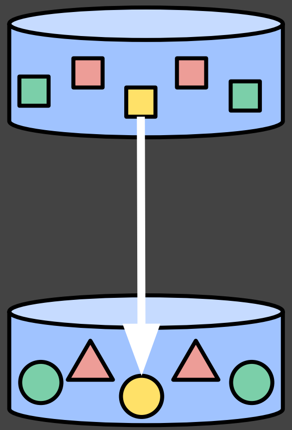


Aggregating
(reduce)

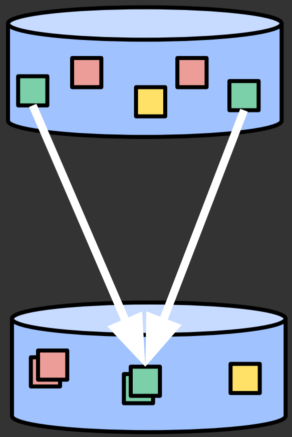


Composite
(reusable combinations)

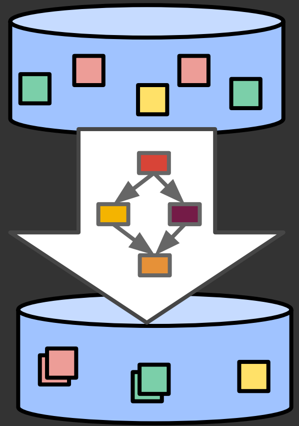
What are you computing?



Element-Wise
(map)

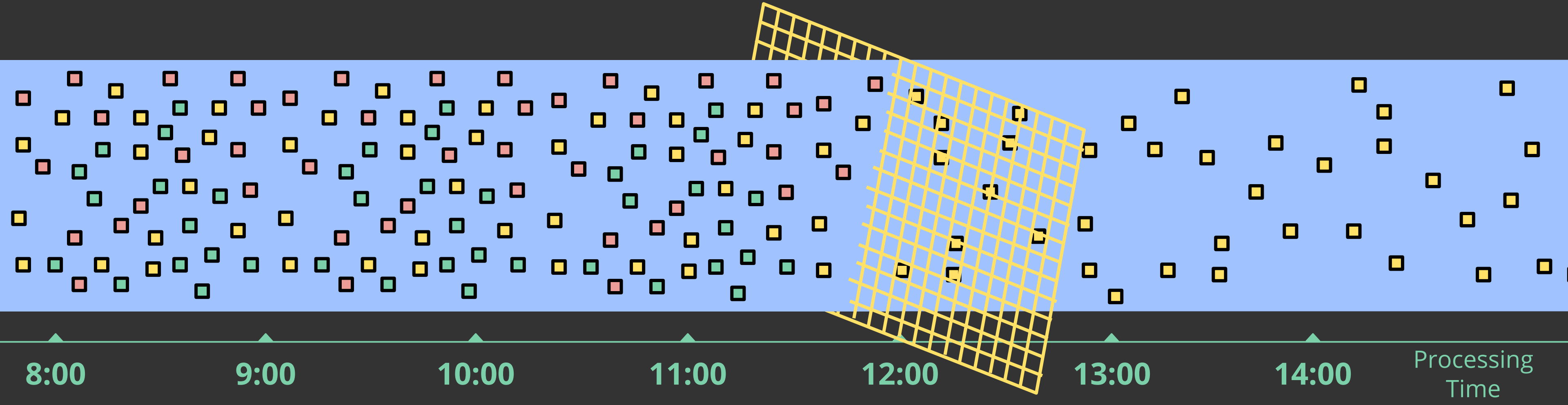


Aggregating
(reduce)

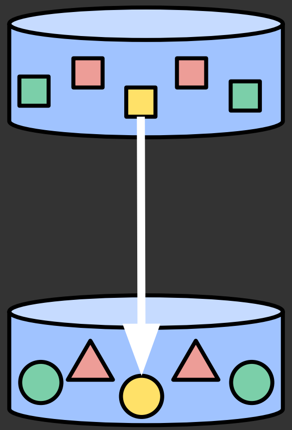


Composite
(reusable combinations)

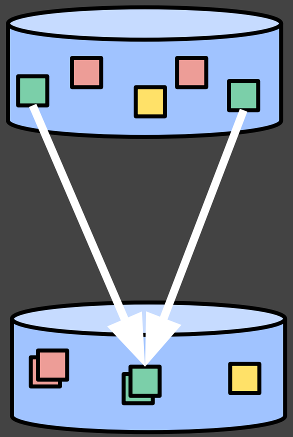
Element-wise transforms



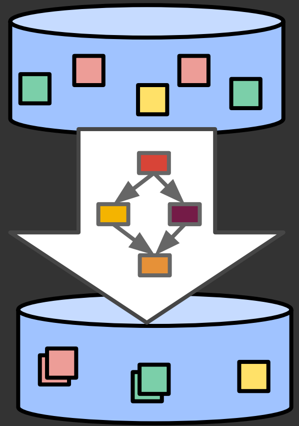
What are you computing?



Element-Wise
(map)

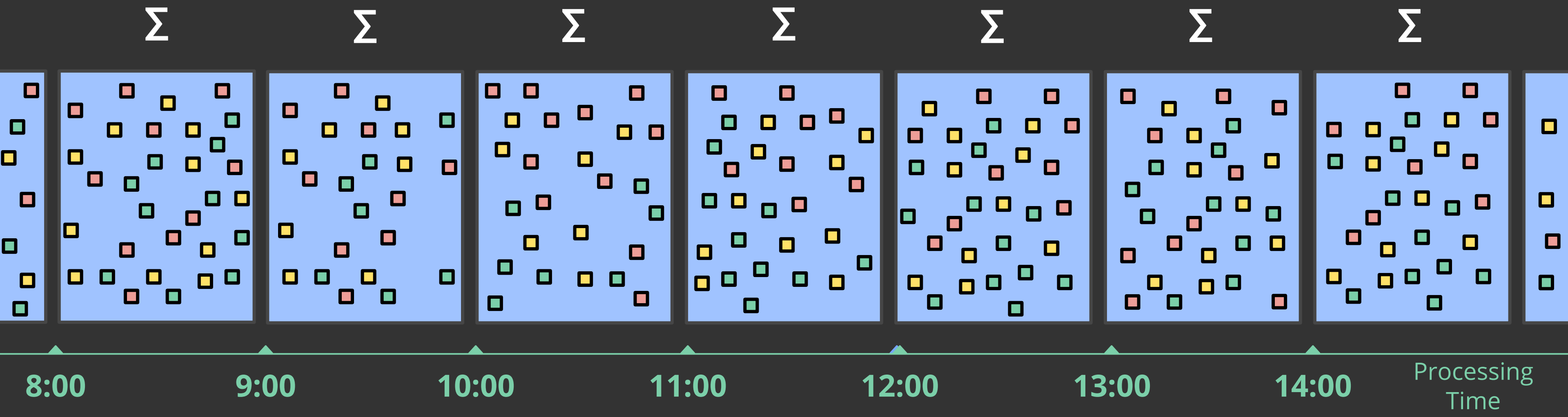


Aggregating
(reduce)

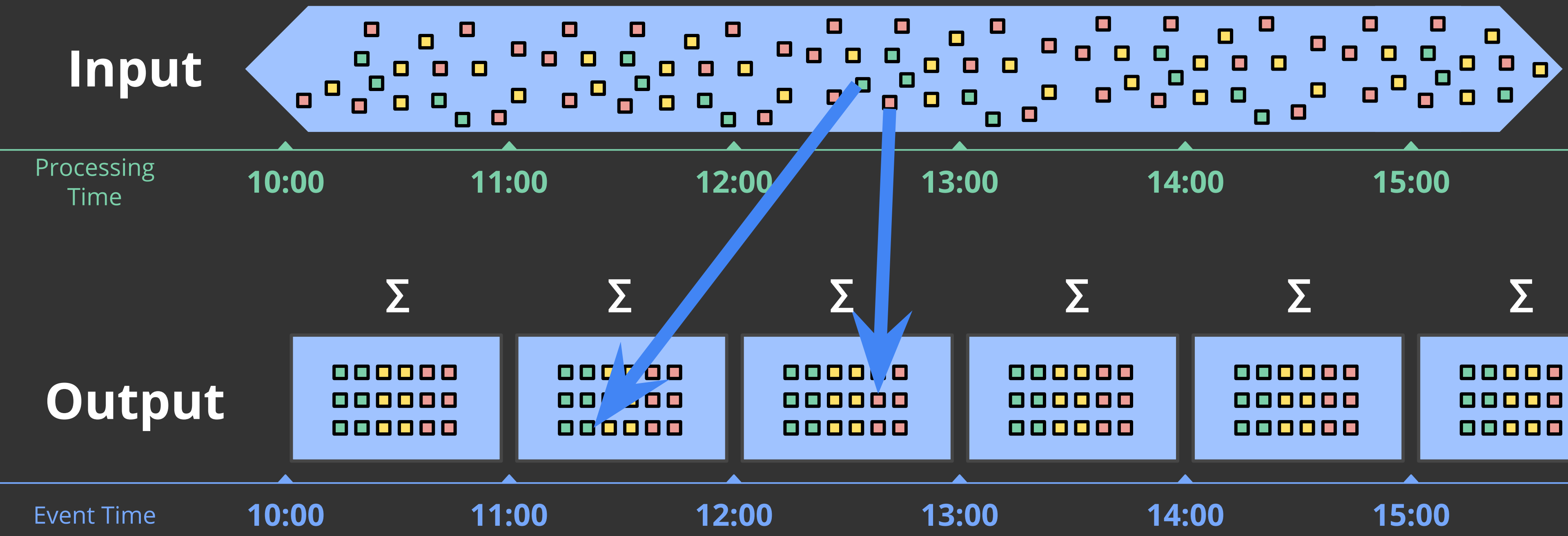


Composite
(reusable combinations)

Grouping via processing-time windows

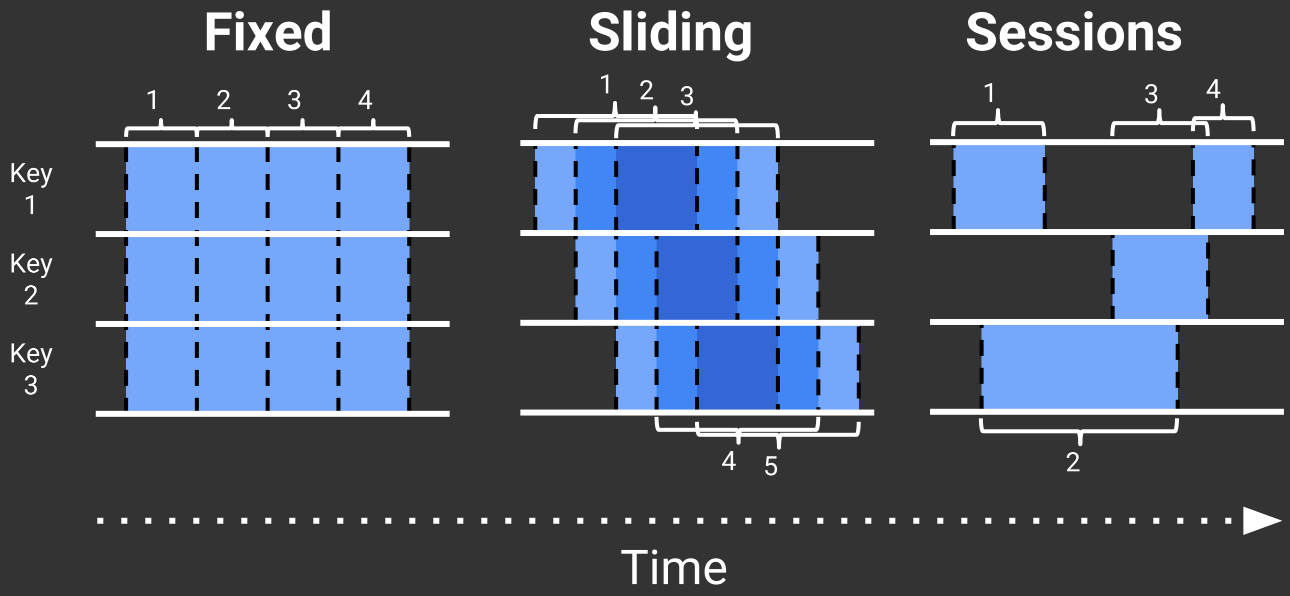


Grouping via event-time windows



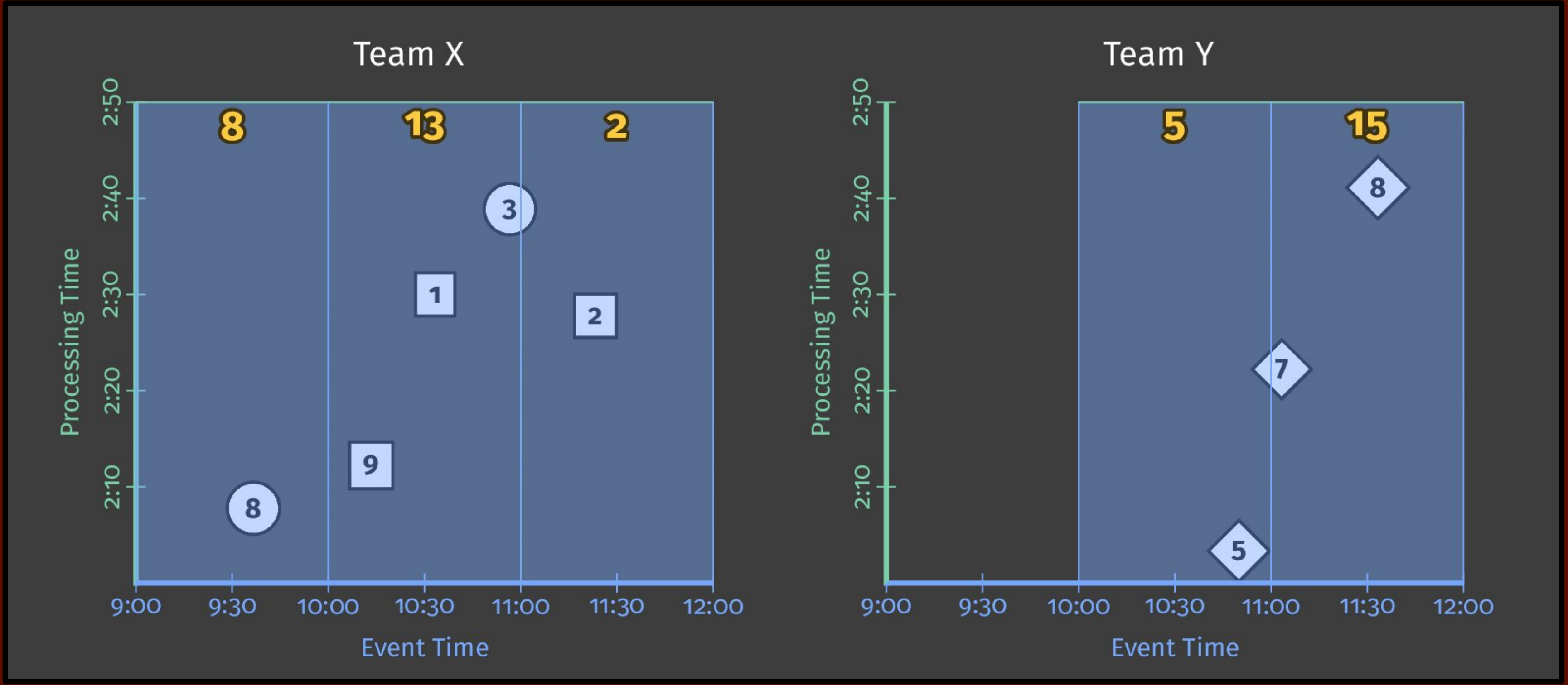
What is windowing?

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



Often required when doing aggregations over unbounded data.

Exercise 3: Hourly team scores



Exercise 3: Hourly team scores

Overview

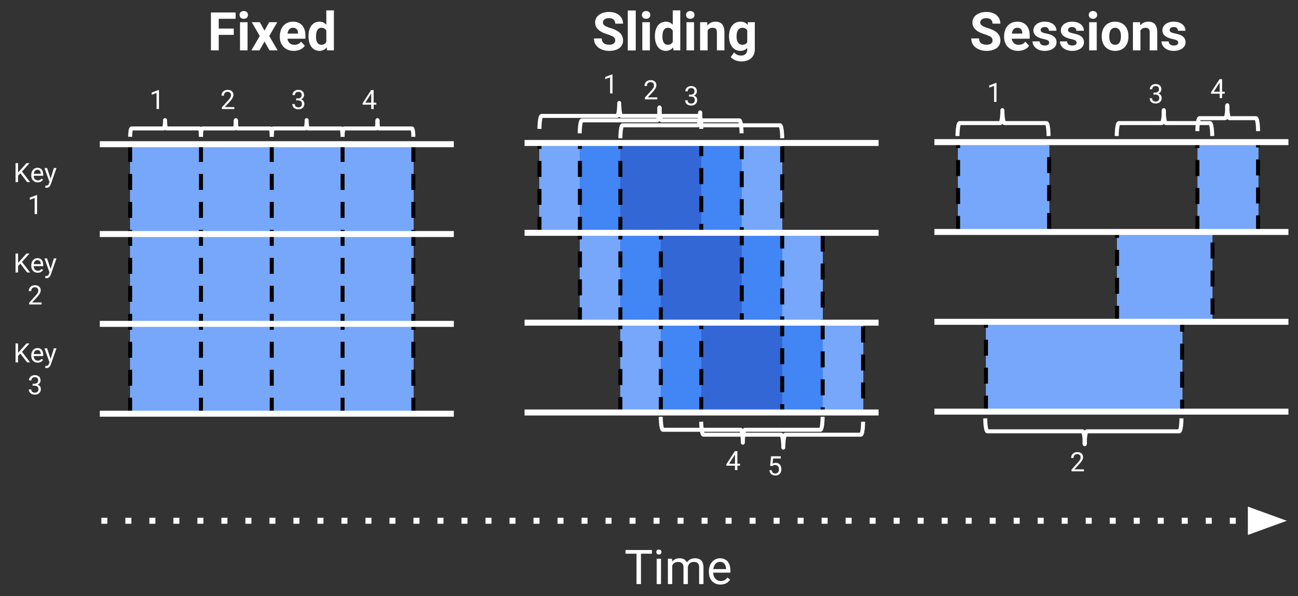
We're going to use windowing to compute the hourly team score

Instructions

1. Find the **WindowedTeamScore** PTransform
2. Fill it in using **FixedWindows**, keying by **team ID**, and computing the **hourly sum of scores**

Windowing recap

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



Often required when doing aggregations over unbounded data.

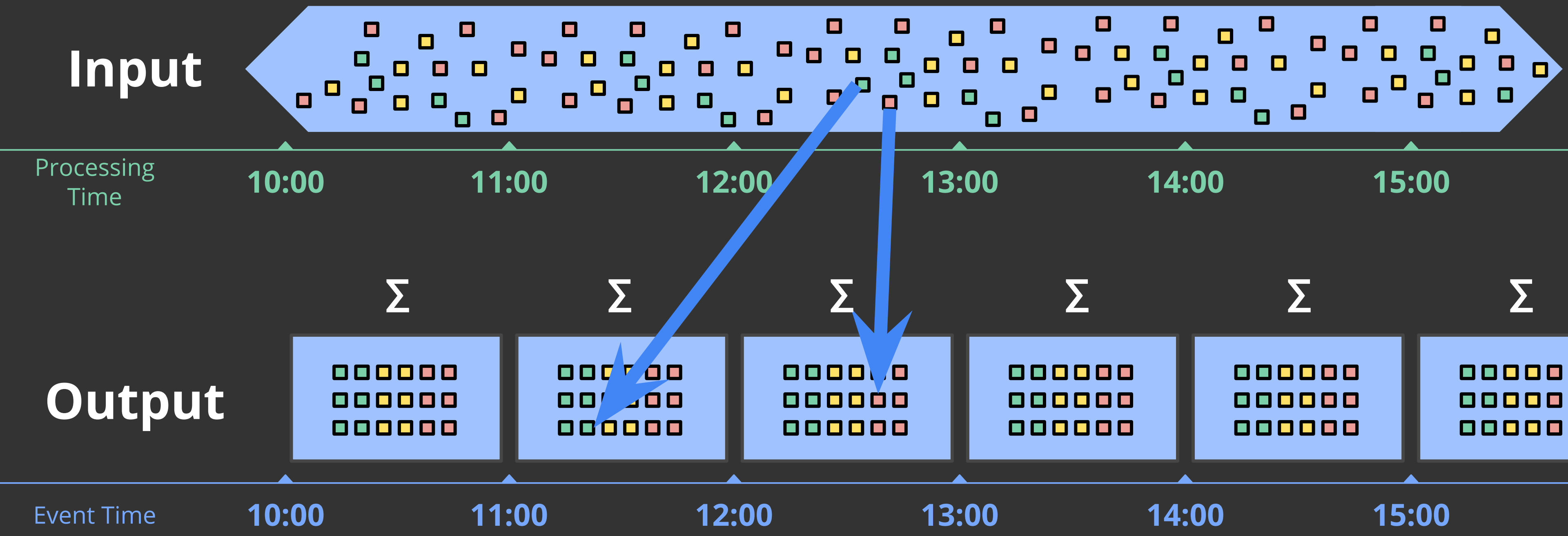
04 Break

Get up and stretch!

05 Triggers & Streaming

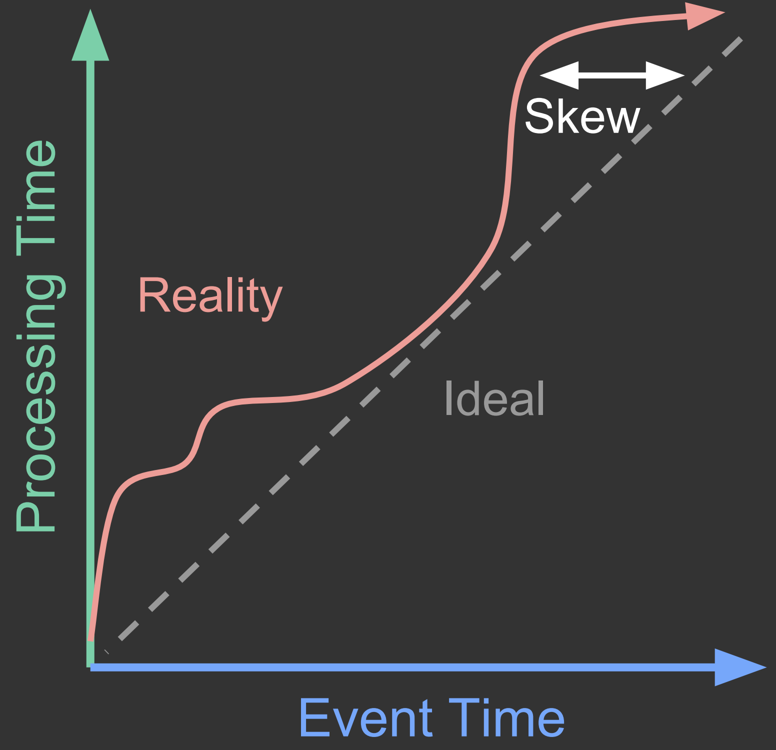
When in processing time are results emitted?

Streaming: Unbounded PCollections

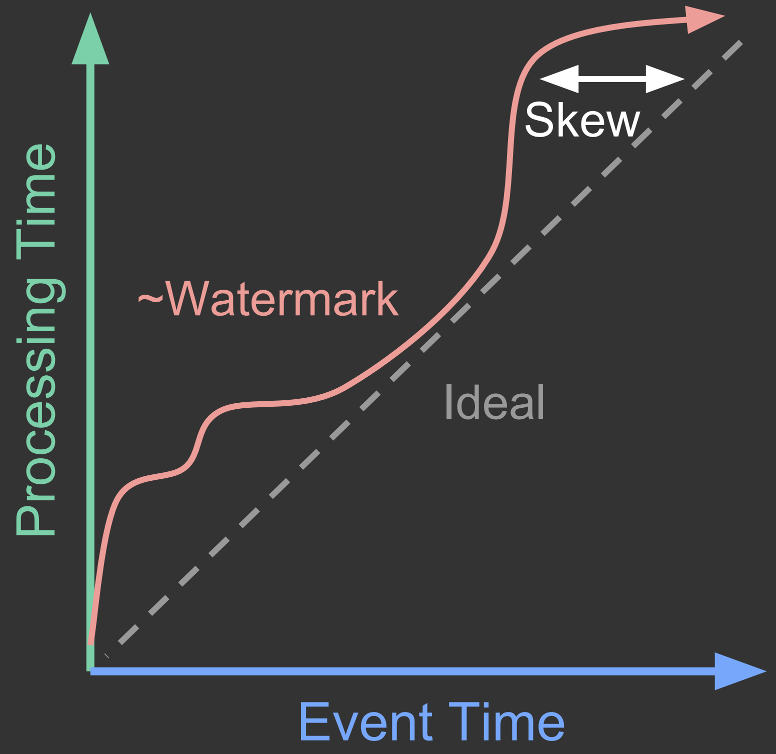


Windowing specifies ***where*** events are aggregated in event time, but ***when*** are events emitted in processing time?

Formalizing event-time skew



Formalizing event-time skew



Watermarks describe event time progress.

"No timestamp earlier than the watermark will be seen"

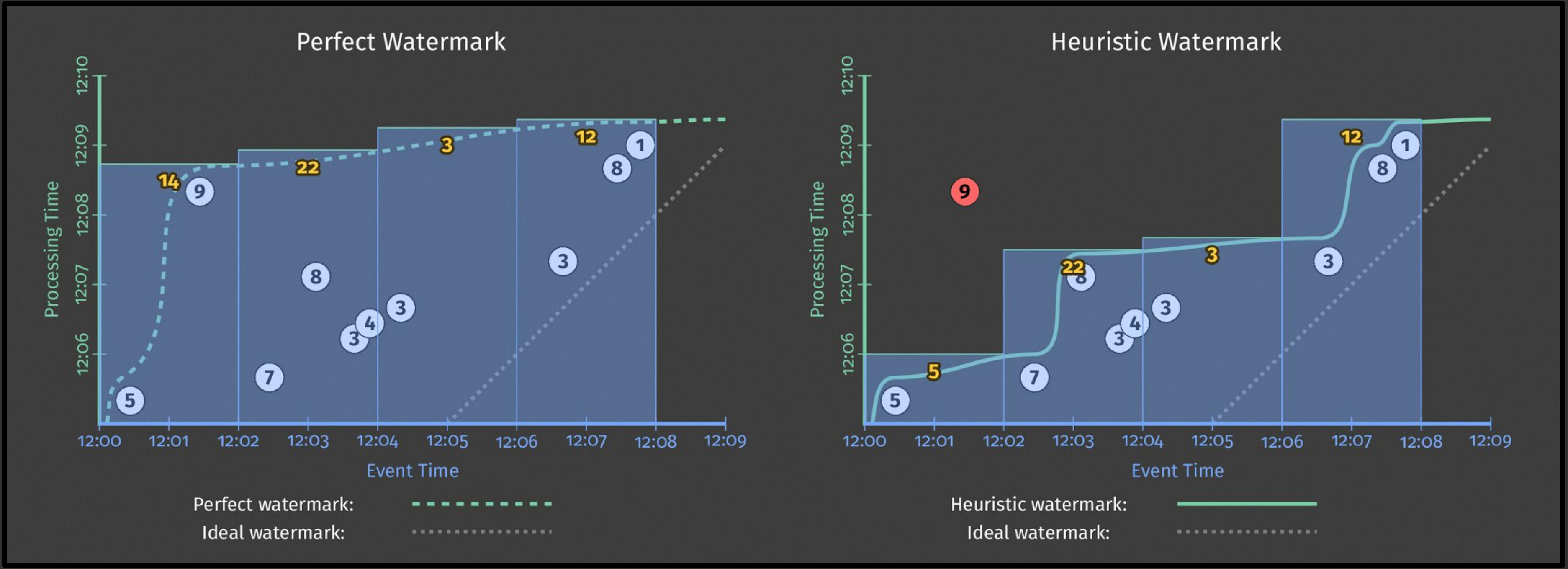
Often heuristic-based.

Too Slow? Results are *delayed*.
Too Fast? Some data is *late*.

When: triggering at the watermark

```
PCollection<KV<String, Integer>> scores = input
    .apply(Window.into(FixedWindows.of(Minutes(2))
              .triggering(AtWatermark()))
    .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*"

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

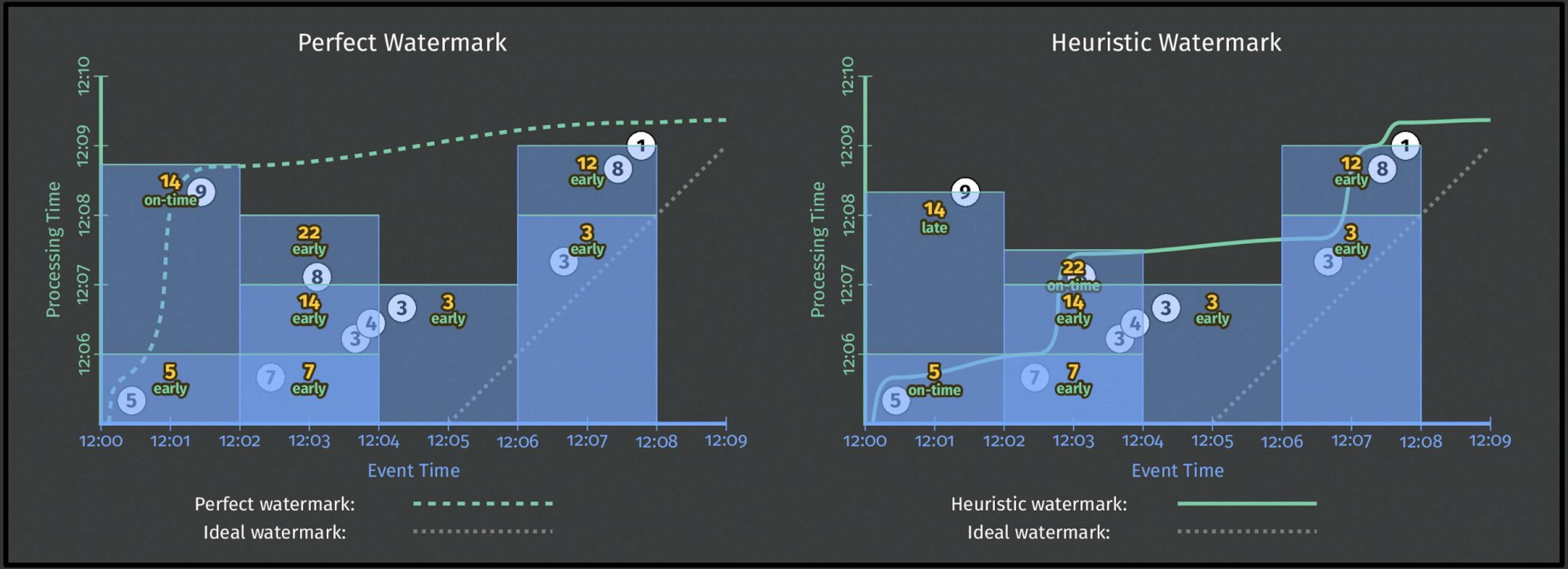
For example:

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

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).

Exercise 4: Streaming leaderboard

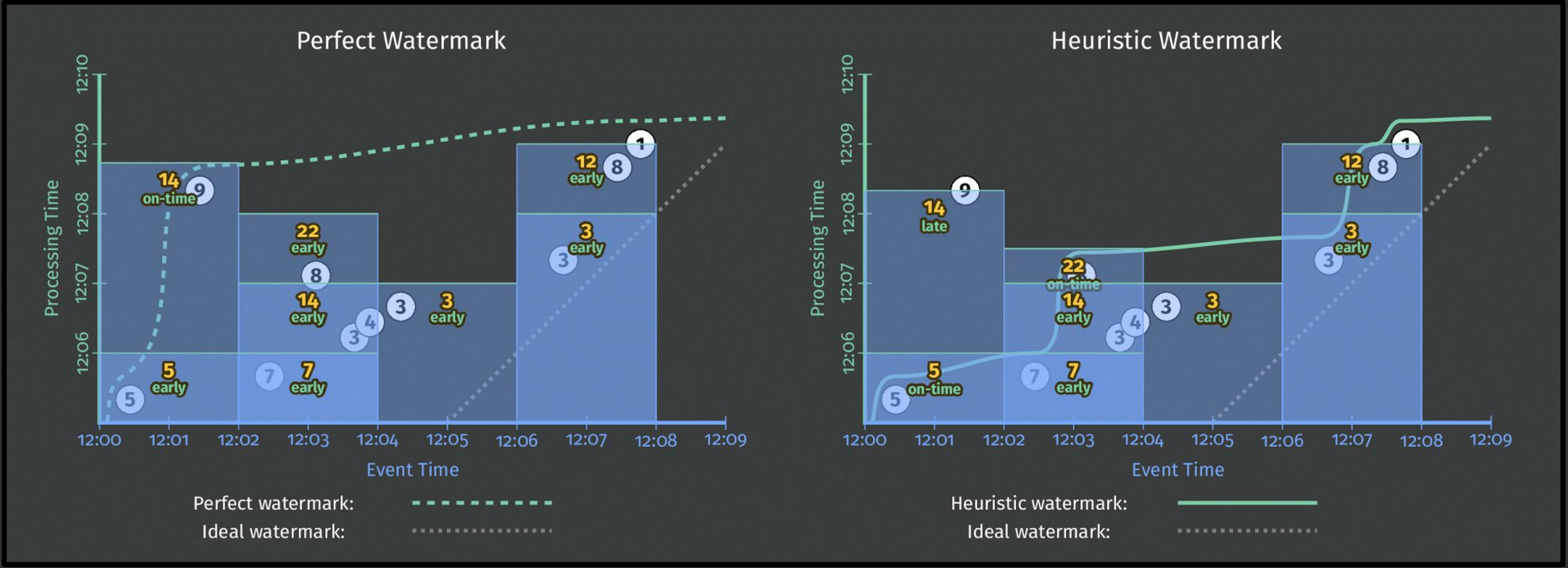
Part 1 - User Leader Board

Calculate the **total score** for every user and publish speculative **results every ten minutes**

Part 2 - Team Leader Board

1. Calculate the **team scores** for **each hour** that the pipeline runs
2. For **each team**, identify the **top scoring user**

Streaming recap



Windowing and triggers enable streaming by:

1. Dividing data into chunks within *event time*
2. Specifying when to produce results in *processing time*

06 Side Inputs & Outputs

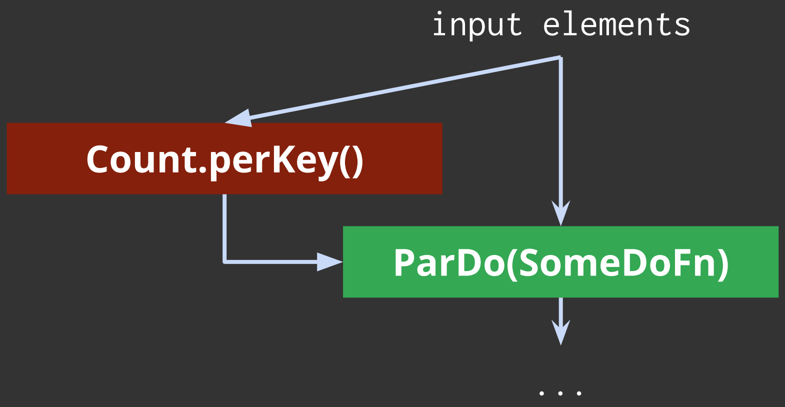
Broadcasts and complex results

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

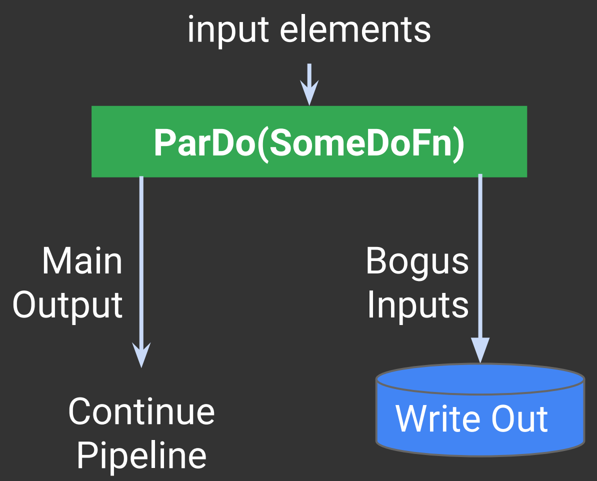
```
PCollection<String> words = ...; // the input PCollection
PCollection<Integer> wordLengths = ...;

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

// Apply a ParDo that takes maxWordLengthCutOffView as a side input.
PCollection<String> wordsBelowCutOff = words.apply(ParDo
    .withSideInputs(maxWordLengthCutOffView).of(new DoFn<String, String>() {
        @ProcessElement
        public void processElement(ProcessContext c) {
          ...
          int lengthCutOff = c.sideInput(maxWordLengthCutOffView);
          ...
        }}));
```

Side outputs

ParDos can produce multiple outputs
For example:

A main output containing all the successfully processed results

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



Example: ParDo with side outputs

```
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(ProcessContext c) {
          try {
            c.output(... c.element() ...);
          } catch (Exception e) {
            c.sideOutput(deadLetterTag, c.element());
          }
        }}));
PCollection<Output> success = outputTuple.get(successTag);
PCollection<Input> deadLetters = outputTuple.get(deadLetterTag);
```

Exercise 5: Game Stats

Part 1 - Find Spammy Users

Complete the **CalculateSpammyUsers** PTransform to determine users who have a score that is 2.5x the global average in each window.

Part 2 - Remove Spammy Users

Complete the **WindowedNonSpamTeamScore** PTransform to compute the team score in each window ignoring users who were identified as spammy.

Summary

We’ve seen how to:

- … use the library of operations in the Apache Beam SDK to create a data processing pipeline
- … use windowing to perform aggregation over specific slices of event time
- … use triggers to control when output is produced
- … use additional structural patterns for more powerful pipelines

Beam mailing list: http://beam.incubator.apache.org/use/mailing-lists/
Slides: http://goo.gl/3O5sZi
Exercises: http://tiny.jesse-anderson.com/beamtutorial

Thank you!