



Apache Beam Tutorial

Strata NYC 9/27/2016

Download these notes and today's exercises from
<http://tiny.jesse-anderson.com/beamtutorial>

Learn Stream Processing with Apache Beam

Outline / Schedule

8:30 - 9:00 - Finish up pre-work

9:00 - 9:45 - Intro & Writing a Pipeline

10:00 - 10:30 - Windowing and Time

10:30 - 11:00 - Break

11:30 - 12:00 - Triggers and Streaming

12:00 - 12:30 - Additional Structural Patterns

(Over, but we'll hang around for questions)

Prework

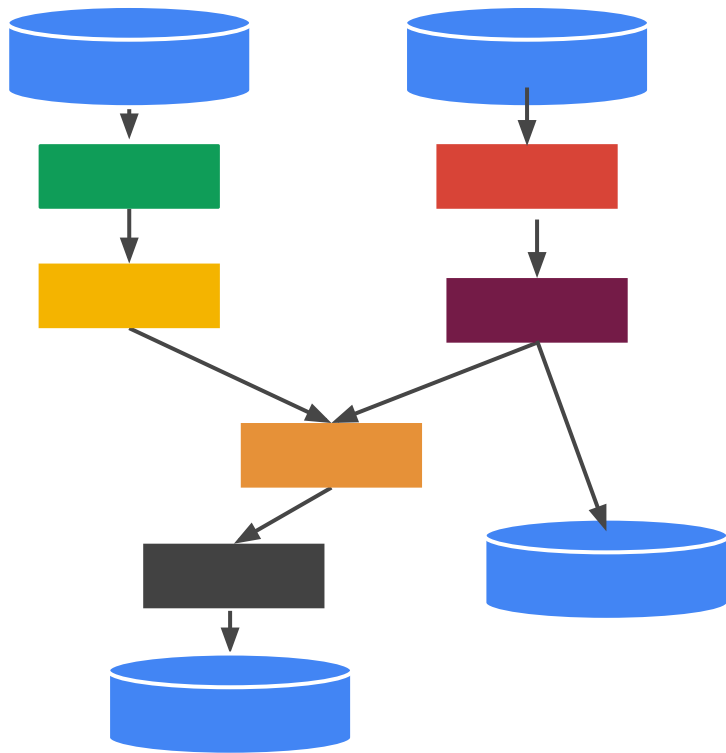
(<http://tiny.jesse-anderson.com/beamtutorial>)

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

Introduction

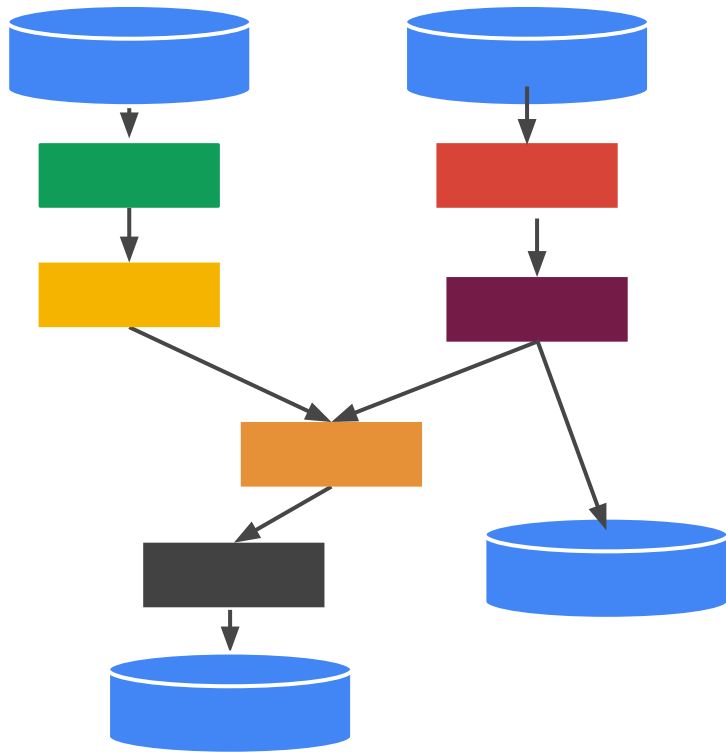
The Apache Beam programming model and usage on GCP

What is Apache Beam (Incubating)?



- Apache Beam is a **unified model** for building data processing pipelines that handle bounded and unbounded data
- Apache Beam is a **collection of SDKs** for building parallelized data processing pipelines
- Google Cloud Dataflow is a **managed service** for executing parallelized data processing pipelines written using Apache Beam

What is a pipeline?



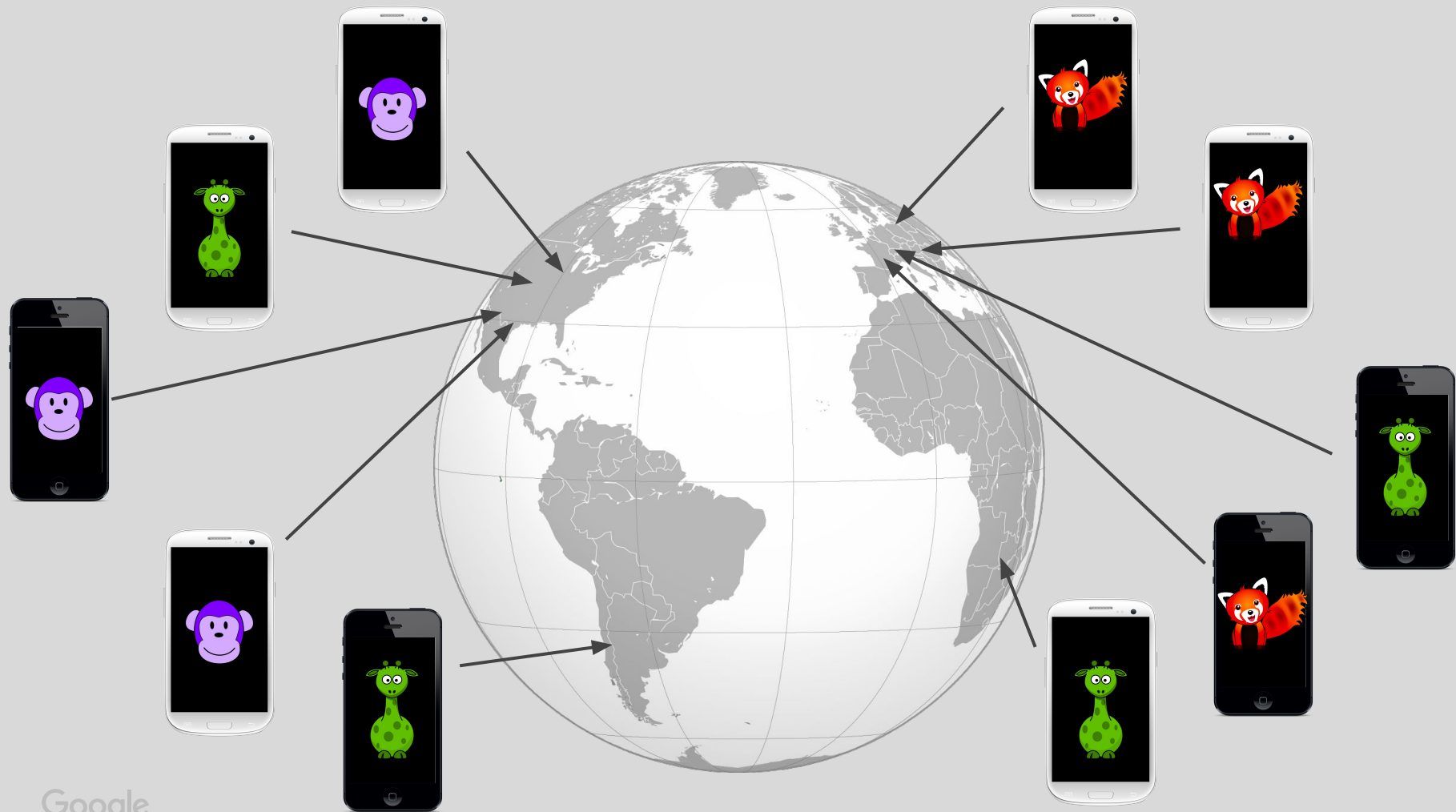
- A Direct Acyclic Graph of data **transformations**
- Possibly **unbounded collections** of data flow on the edges
- May include multiple sources and multiple sinks
- Optimized and executed as a unit

The pipeline describes...

What results are calculated?

Where in event time are results calculated?

When in processing time are results emitted?

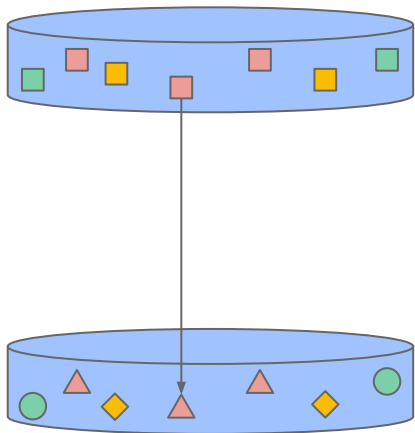


Writing a Pipeline

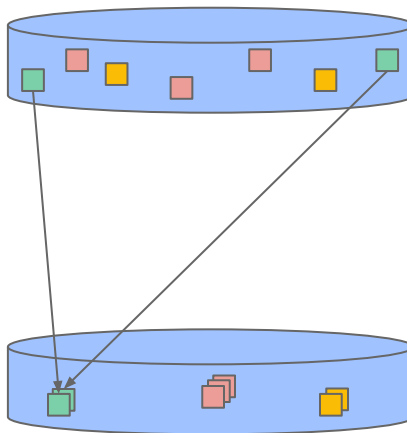
What results are calculated?

Writing a Pipeline = Gluing Together Pieces

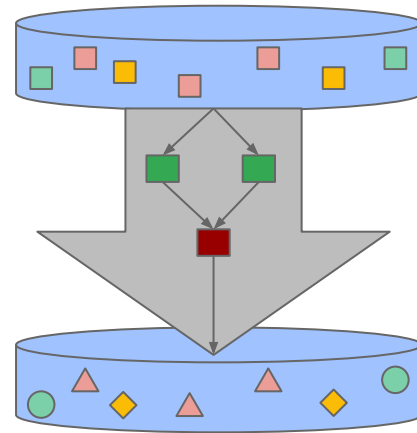
**Element-Wise Transforms
(map)**



**Grouping Transforms
(reduce)**



**Composite Transforms
(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...

`{Seahawks, NFC, Champions, Seattle, ...}`



ParDo(KeyByFirstLetter)



`{KV<S, Seahawks>, KV<N, NFC>, KV<C, Champions>, KV<S, Seattle>, ...}`

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));
    }
})));
```

{Seahawks, NFC, Champions, Seattle, ...}

ParDo(KeyByFirstLetter)

{KV<S, Seahawks>, KV<N, NFC>, KV<C, Champions>, KV<S, Seattle>, ...}

Element Wise Transforms: ParDo

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

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));
    }
})));
```

Element Wise Transforms: ParDo

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

{Seahawks, NFC, Champions, Seattle, ...}



ParDo(ExplodePrefixes)



{s, se, sea, seah, seaha, seahaw,
seahawk, seahawks, n, nf, nfc, c, ch,
cha, cham, champ, champi, champio,
champion, champions, s, se, sea, seat,
seatt, seattl, seattle, ...}

{Seahawks, NFC, Champions, Seattle, ...}



ParDo(FilterOutSWords)



{NFC, Champions, ...}

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

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

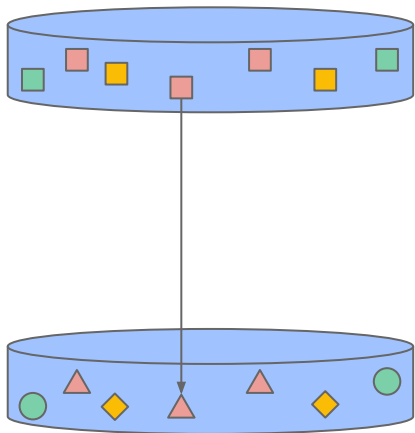
Values

KV(key, value) -> value

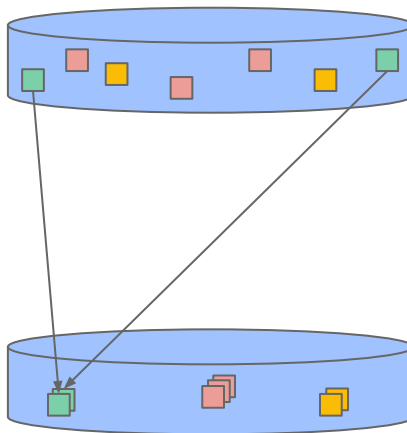
```
// Values
input.apply(Values.create())
```

Writing a Pipeline = Gluing Together Pieces

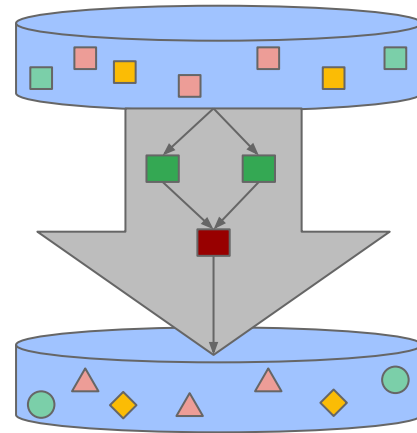
Element-Wise Transforms
(map)



Grouping Transforms
(reduce)



Composite Transforms
(reusable combinations)



Grouping Transforms: GroupByKey

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

{KV<S, Seahawks>, KV<C,Champions>, KV<S, Seattle>, KV<N, NFC>, ...}

GroupByKey

{KV<S, [Seahawks, Seattle, ...]>, KV<N, [NFC, ...]>, KV<C, [Champion, ...]>, ...}

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, Seahawks>, KV<C,Champions>, KV<S, Seattle>, KV<N, NFC>, ...}`

GroupByKey

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

`, ...]>, KV<C, [Champion, ...]>, ...}`

How can

on value for each key?

Grouping Transforms: GroupByKey

Computing the most common value for each key

{KV<S, Seahawks>, KV<C,Champions>, KV<S, Seattle>, KV<N, NFC>, ...}

GroupByKey

{KV<S, [Seahawks, Seattle, ...]>, KV<N, [NFC, ...]>, KV<C, [Champion, ...]>, ...}

ParDo(TopInIterable)

{KV<S, Seahawks>, KV<N, NFC>, KV<C, Champion>}

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

Grouping Transforms: GroupByKey

This is so common, that the SDK includes a short-hand

{KV<S, Seahawks>, KV<C,Champions>, KV<S, Seattle>, KV<N, NFC>, ...}



Combine.perKey(Top.TopFn) or Top.perKey()



{KV<S, Seahawks>, KV<N, NFC>, KV<C, Champion>}

Grouping Transforms: GroupByKey

This is so common, that the SDK includes a short-hand

`{KV<S, Seahawks>, KV<C,Champions>, KV<S, Seattle>, KV<N, NFC>, ...}`



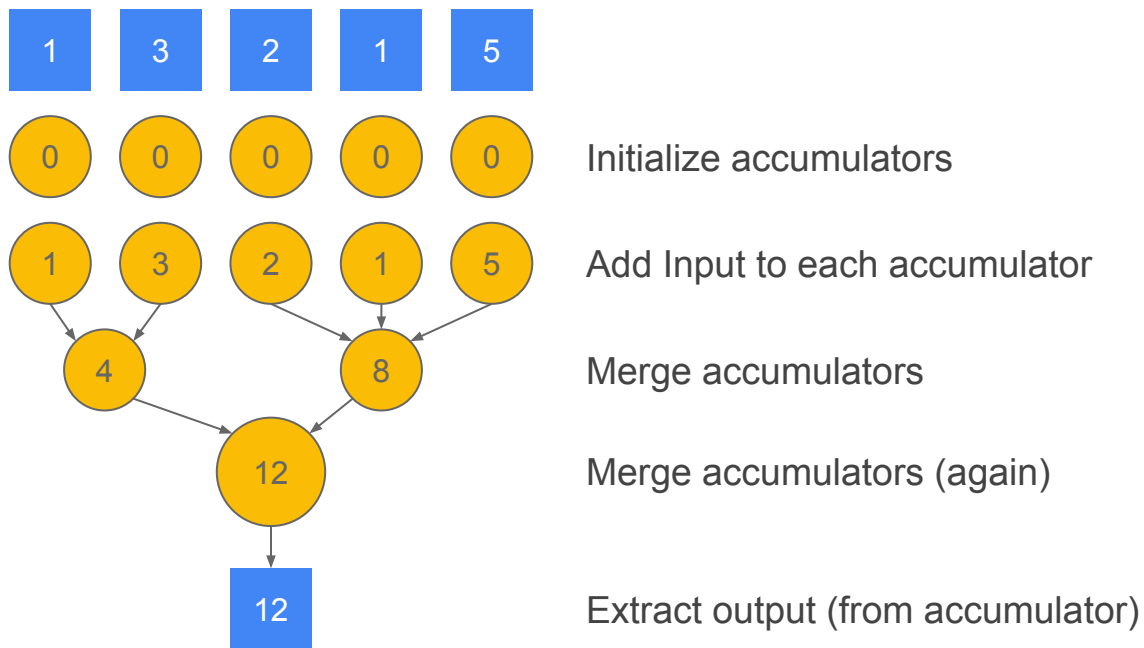
Combine.perKey(Top.TopFn) or Top.perKey()



```
input.apply(Top.perKey(10, new SerializableComparator<KV<String, String>>() {  
    ...  
}))
```

Grouping Transforms: Combine

CombineFns are user code too -- you can write your own for any associative/commutative operation



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)

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

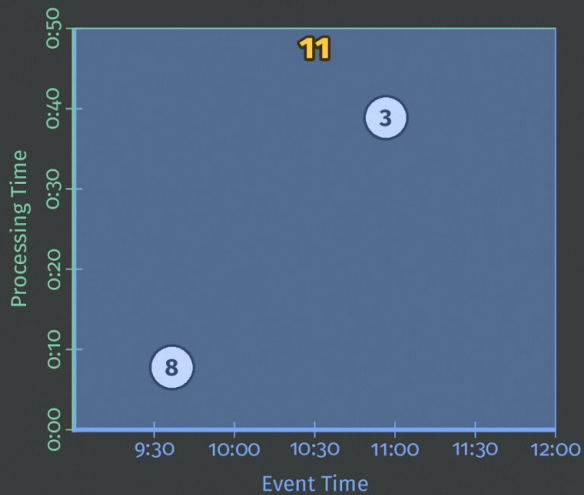
The **team ID** the user is on

A **score** for that particular play

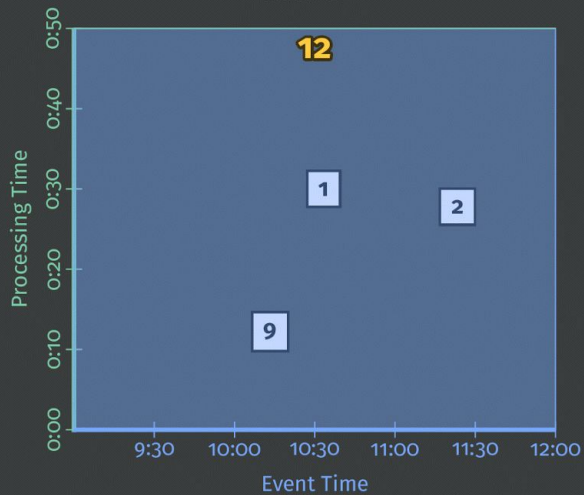
A **timestamp** that records when the play happened

ExtractAndSumScore

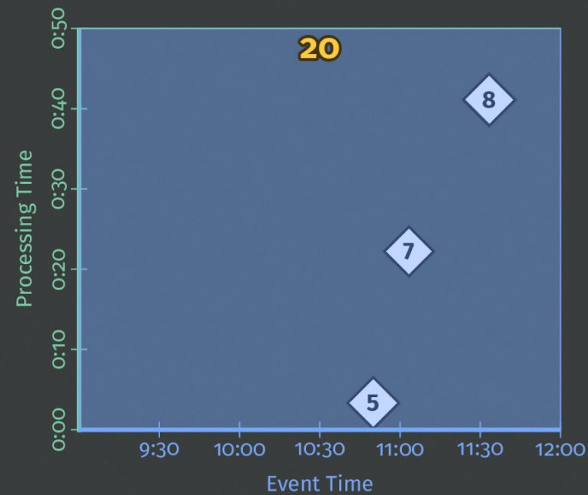
User A



User B



User C



Exercise 1: Implement the ExtractAndSumScore

Overview

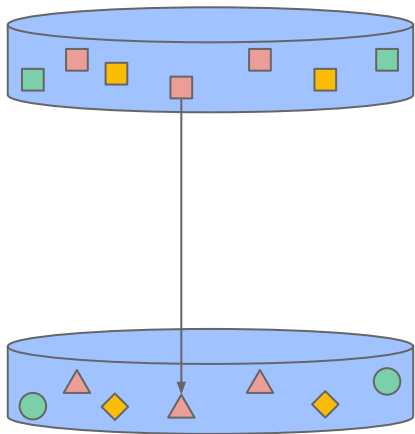
We're going to start with the **DirectPipelineRunner** -- this executes the pipeline locally (on your machine) and is great for testing

Instructions

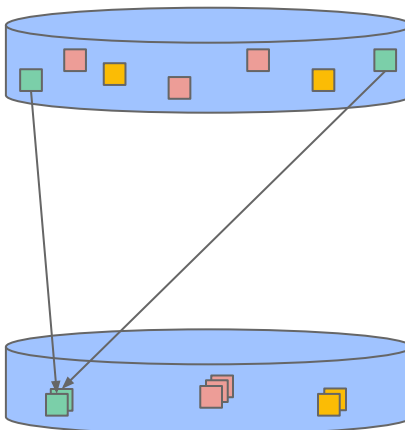
1. Find the empty **ExtractAndSumScore** **PTransform**
2. Add code to extract the score keyed by **user ID** and then compute the sum for each user
3. Run your pipeline using the **DirectPipelineRunner**

Writing a Pipeline = Gluing Together Pieces

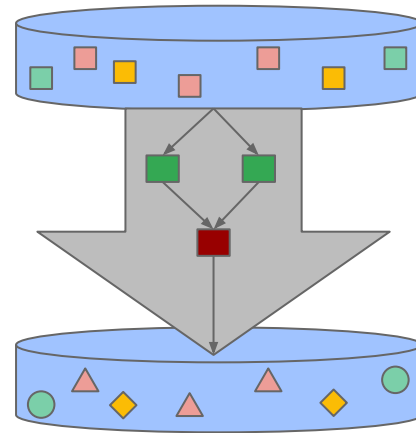
Element-Wise Transforms
(map)



Grouping Transforms
(reduce)



Composite Transforms
(reusable combinations)



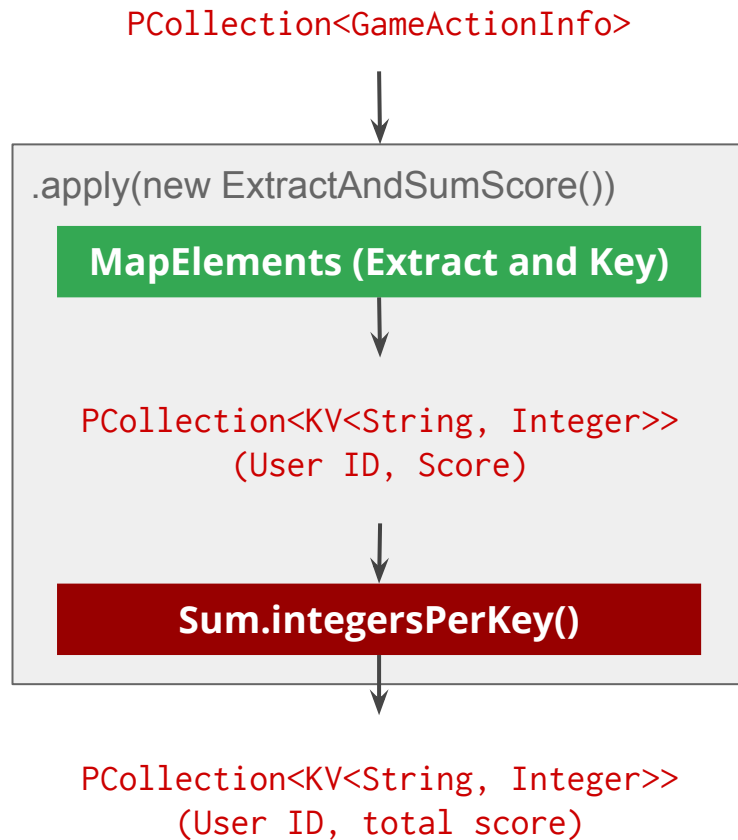
Composite Transforms

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

We've already seen some composite transforms

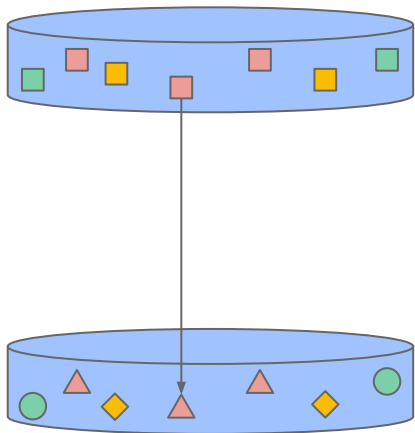
Creating higher level PTransforms is useful for organizing your pipeline

Each PTransform can be tested to ensure it behaves correctly

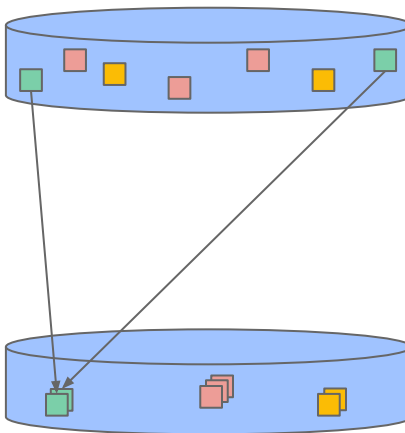


Writing a Pipeline = Gluing Together Pieces

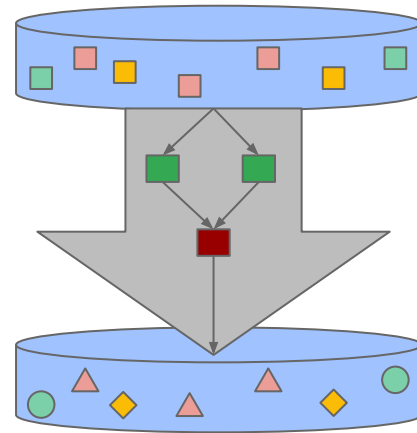
Element-Wise Transforms
(map)



Grouping Transforms
(reduce)



Composite Transforms
(reusable combinations)



Pipeline Runners

Direct Runner

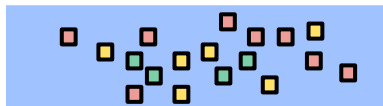
Run locally (on your machine) for testing and debugging



Dataflow Runner (Batch)

Run on the Cloud Dataflow managed service

Optimized for processing of large, bounded PCollections



Dataflow Runner (Streaming)

Run on the Cloud Dataflow managed service

Optimized for low-latency results and long-running pipelines over incoming data



Other Runners

Apache Beam supports Apache Flink, Apache Spark, etc.

Exercise 2: User Score in the Cloud

Overview

Ok, now let's run that on the Dataflow service...

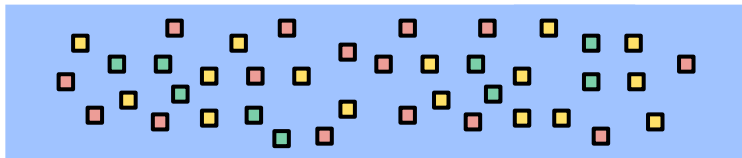
Instructions

1. Run your pipeline using the Dataflow **service**
2. Make sure you look at the **Cloud Console** for the Dataflow job
3. Look at **Cloud Logging** for the job as well

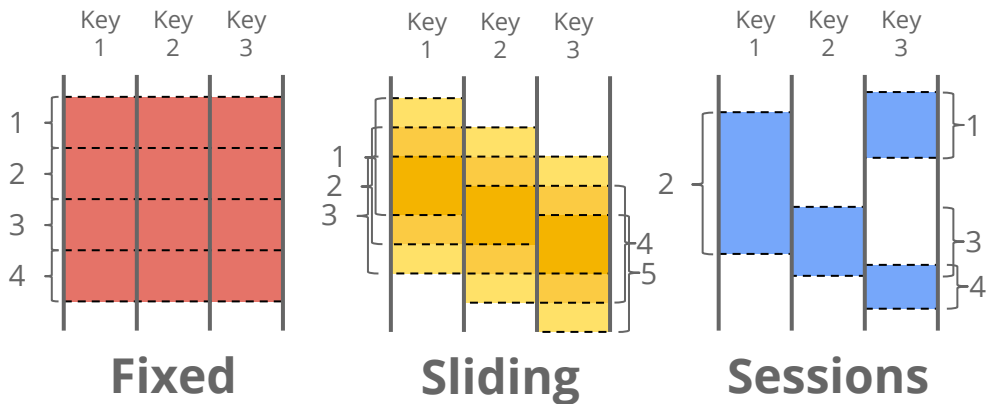
Windowing and Time

Where in event time are results calculated?

What is Windowing?

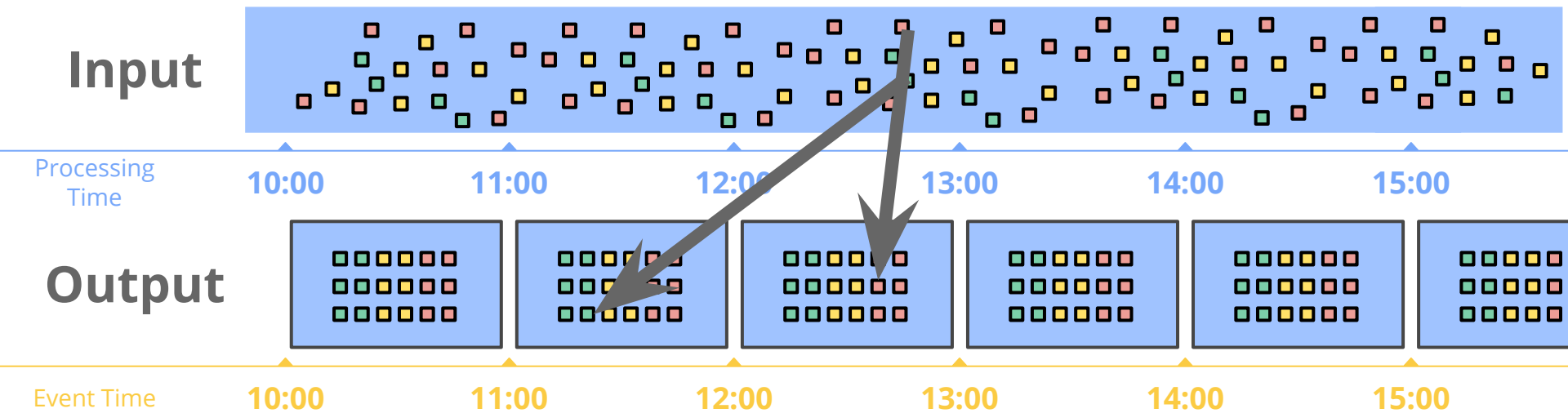


Windowing partitions
data based on the
timestamps
associated with events



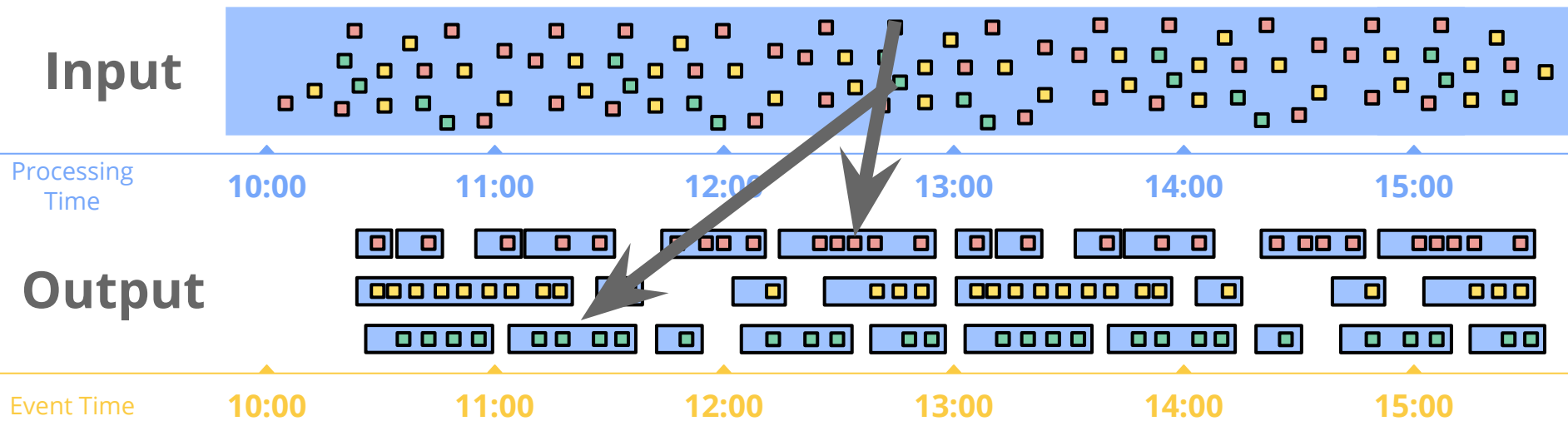
Windowing and Time: Fixed Windows

Apache Beam makes it easy to divide your input into windows based on timestamps (event time)



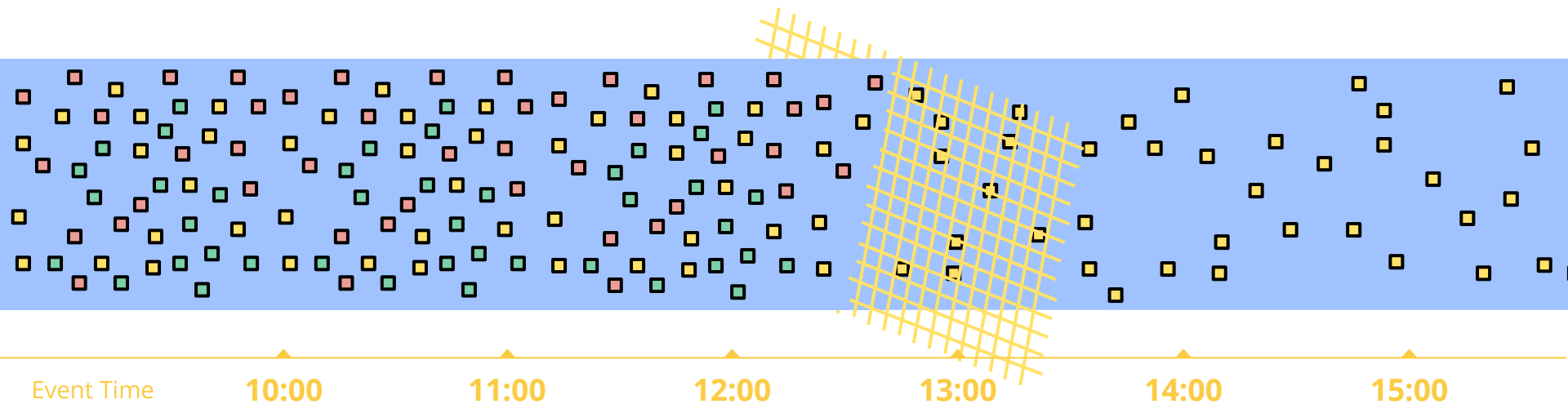
Windowing and Time: Session Windows

Windows can even be assigned per-key
For example: User Sessions



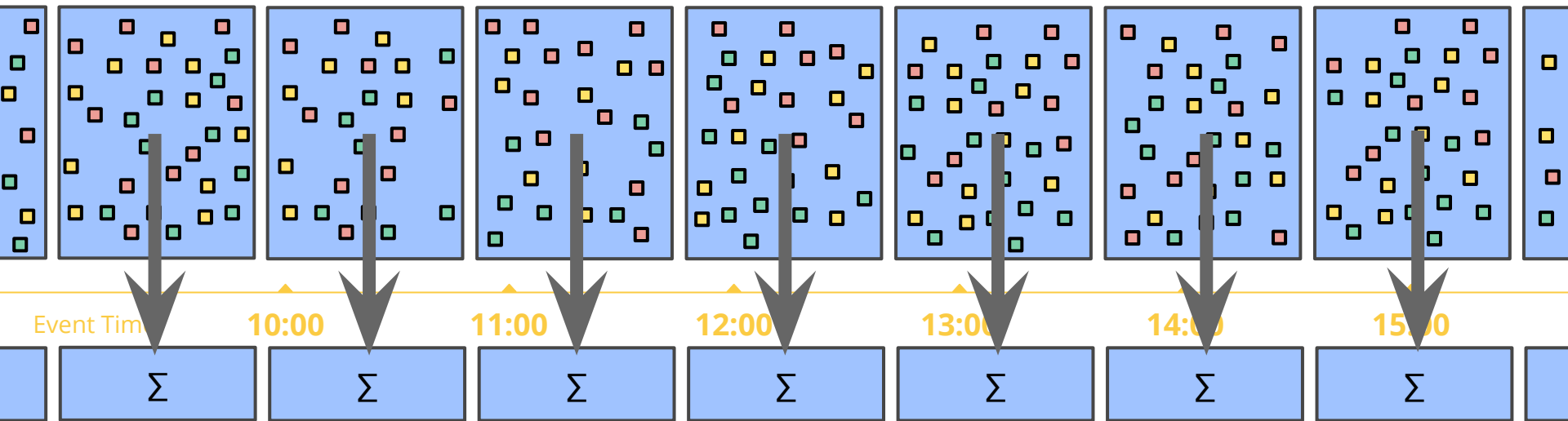
Windowing and Time: Element Wise Transforms

Element-wise transforms are “easy” -- apply to each element

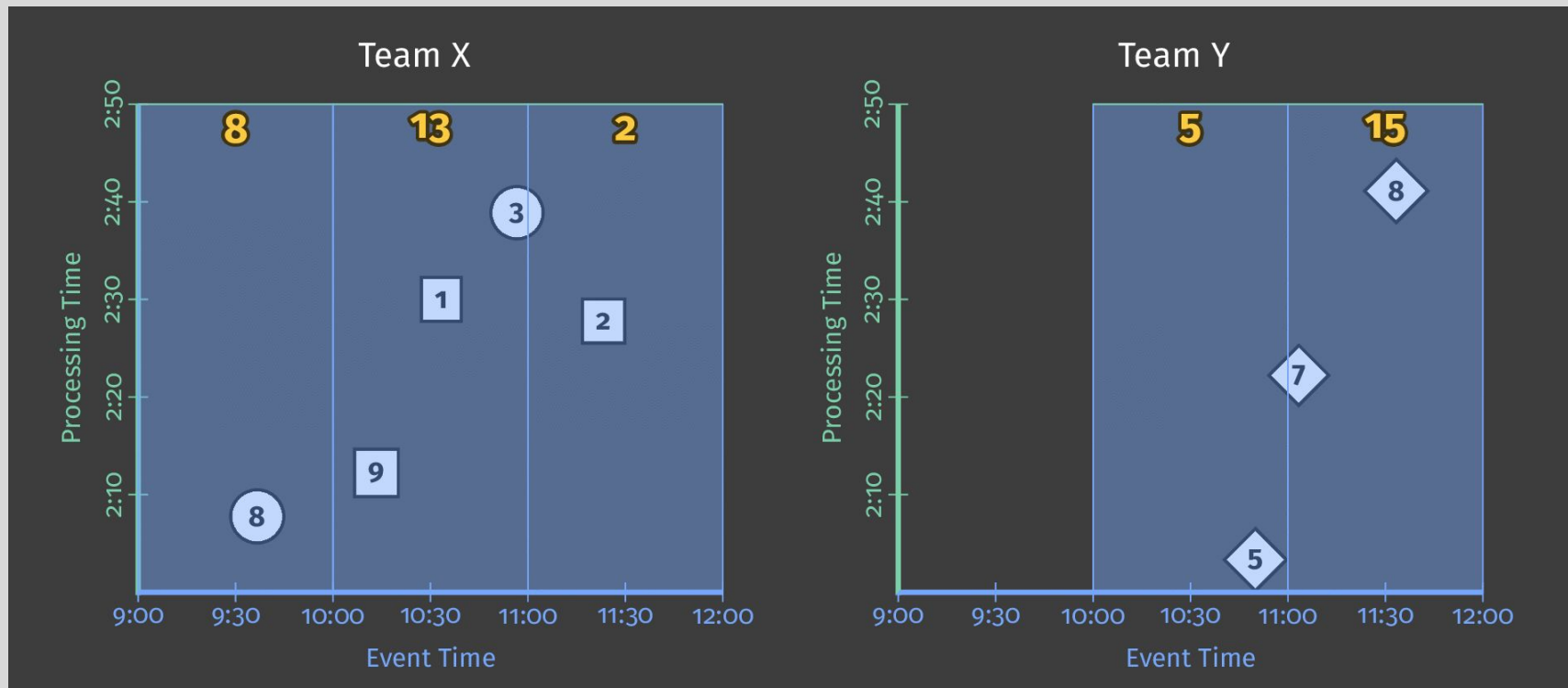


Windowing and Time: Grouping Transforms

Grouping transforms happen within each window



Exercise 3: Hourly Team Score



Exercise 3: Hourly Team Score

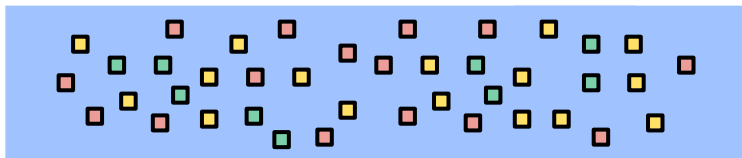
Overview

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

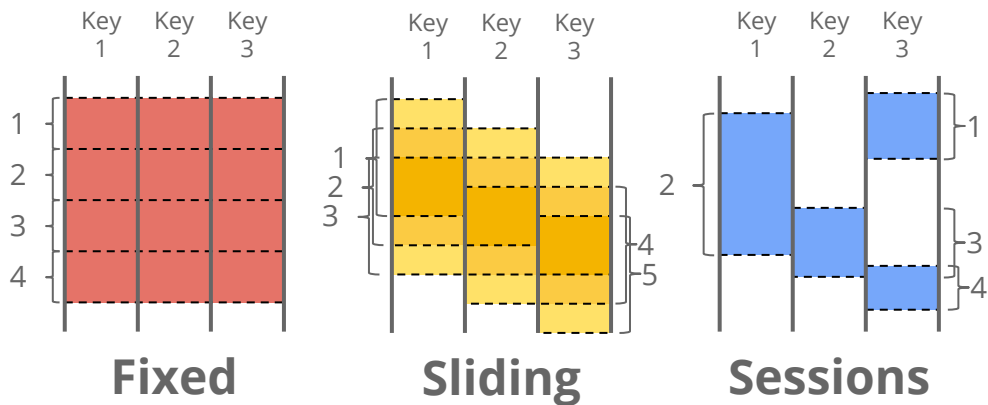
Instructions

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 finite event-time-based chunks.

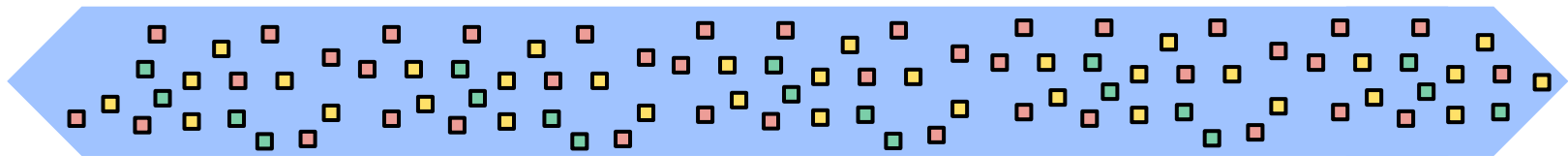


Triggers & Streaming

When in processing time are results emitted?

Streaming: Unbounded PCollections

Input



Processing Time

10:00

11:00

12:00

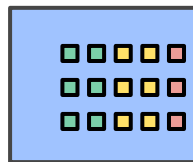
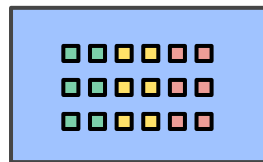
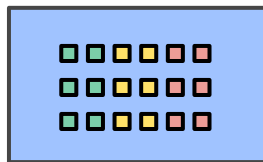
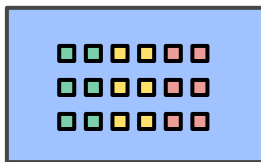
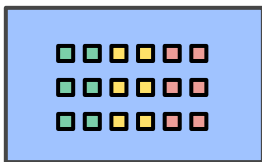
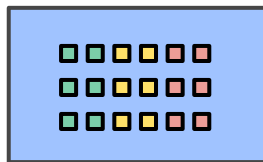
13:00

14:00

15:00

Windowing specifies what events get aggregated...

Output



Event Time

10:00

11:00

12:00

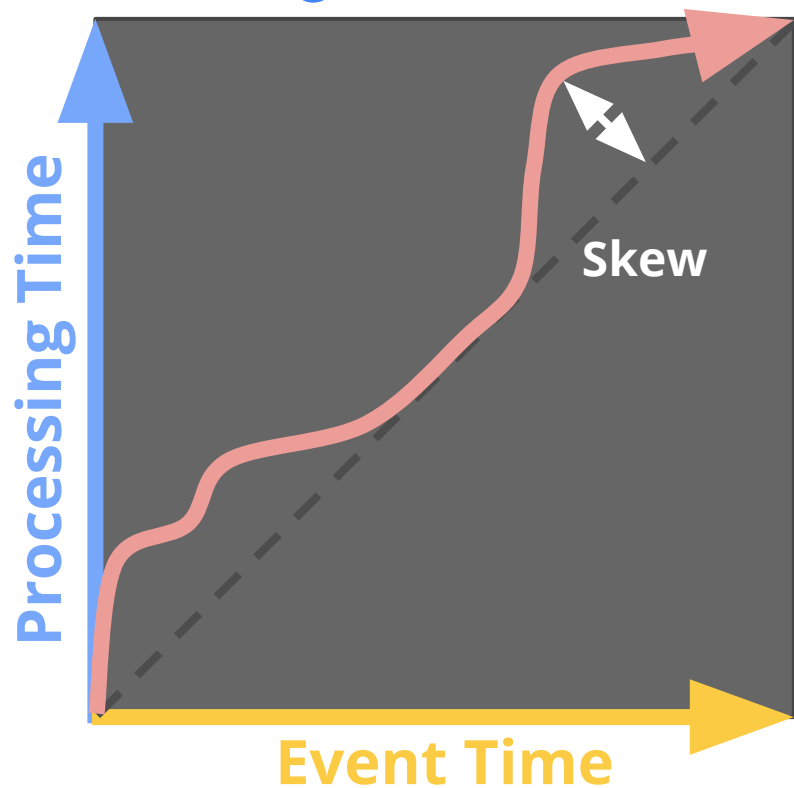
13:00

14:00

15:00

When do these aggregates get produced?

Streaming: Time and Skew



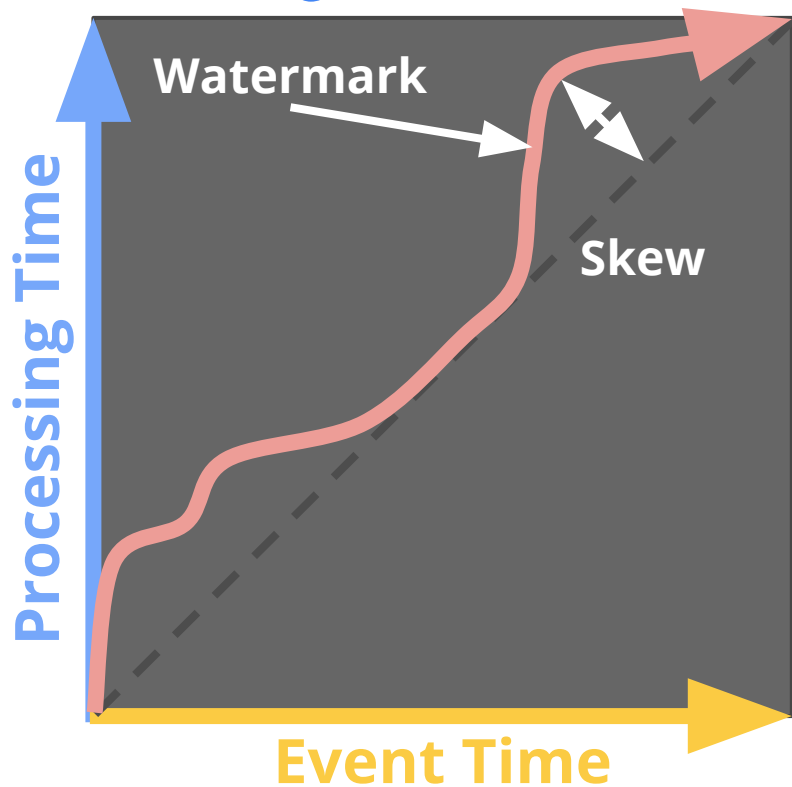
Processing time is current wall time

Event time comes from timestamps of each element (event)

Due to delays in the system processing time \geq event time

processing time = event time + skew
skew ≥ 0 and changes over time

Streaming: Time and Skew and the Watermark

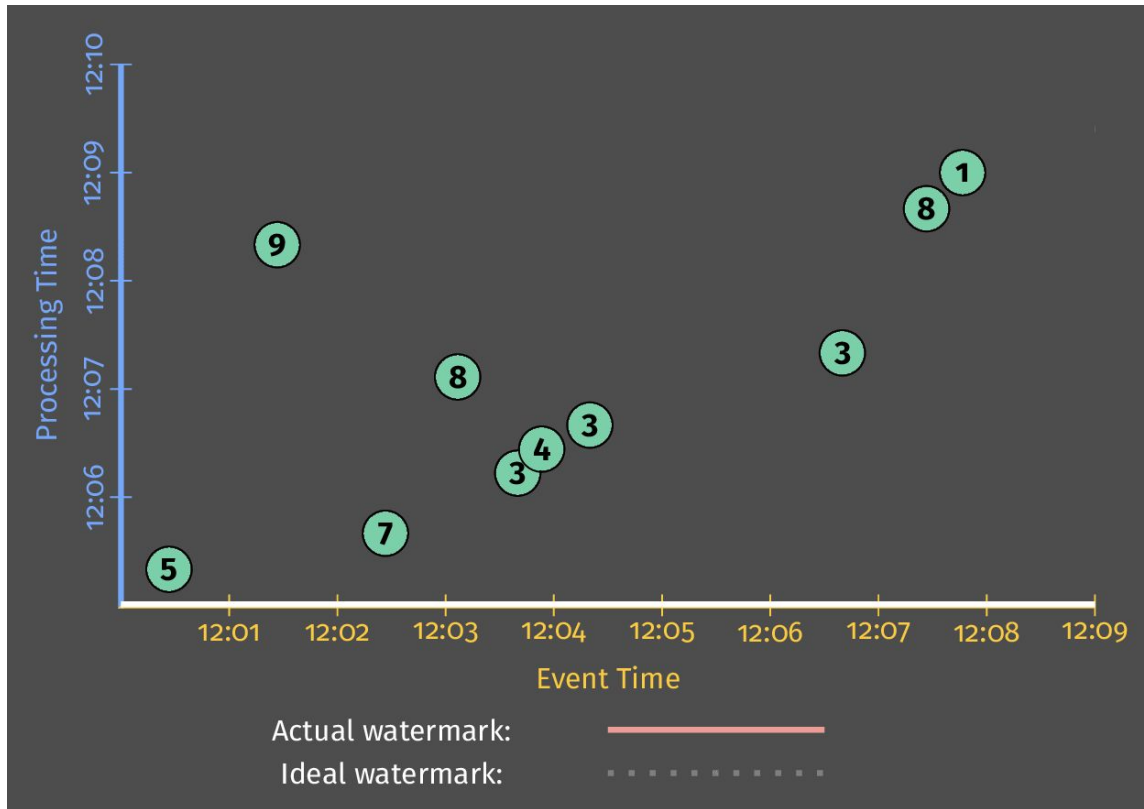


Watermarks are a relation between processing time and event time

"No timestamp earlier than the watermark will be seen"

Often a heuristic estimate of progress

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



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”*

Example: Triggering at the Watermark

```
PCollection<KV<String, Integer>> output = input
    .apply(Window
        .into(FixedWindows.of(Duration.standardMinutes(2)))
        .triggering(AfterWatermark.pastEndOfWindow()))
    .apply(Sum.integersPerKey());
```

Example: Triggering at the End of the Window

Streaming - Other kinds of Triggers

Element Count

Output after at least N elements

Processing Time

Output after at least N minutes

Combinators

After all of these

After any of these

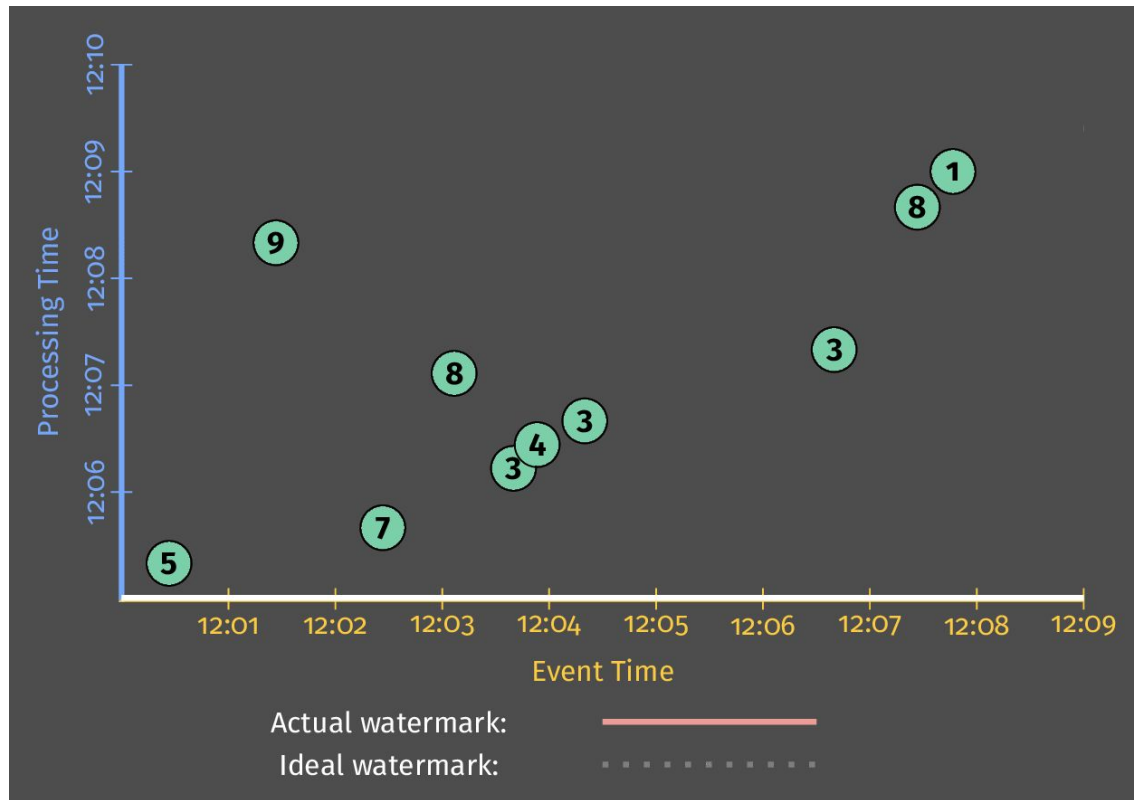
After each of these in order

etc.

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

For example:

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



Speculative: every minute

On-Time: when watermark predicts the window is complete

Late: after every element

Example: Multiple outputs

```
PCollection<KV<String, Integer>> output = input
    .apply(Window
        .into(
            FixedWindows.of(Duration.standardMinutes(1)))
        .triggering(AfterWatermark.pastEndOfWindow()
            .withEarlyFirings(AfterProcessingTime
                .pastFirstElementInPane()
                .plusDelayOf(Duration.standardMinutes(1))
                .alignedTo(Duration.standardMinutes(1)))
            .withLateFirings(AfterPane
                .elementCountAtLeast(1)))
        .withAllowedLateness(Duration.standardDays(1)))
    .apply(Sum.integersPerKey());
```

Speculative: every minute

On-Time: when watermark
predicts the window is
complete

Late: after every element

Example: Speculative and Late Outputs

Exercise 4: Streaming Leaderboard

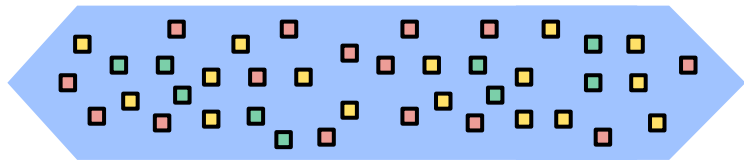
Part 1 - User Leader Board

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

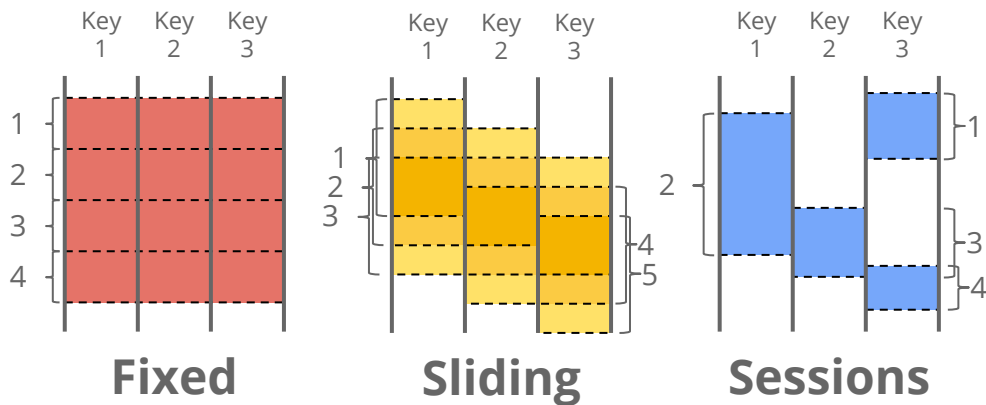
Part 2 - Team Leader Board

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 dividing data into chunks and specifying when to produce results



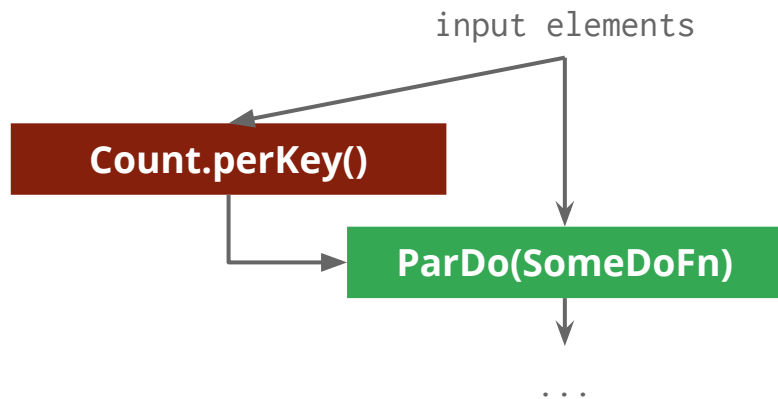
Additional Structural Patterns

Advanced Topics: 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



```
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);
            ...
        }
    }));
```

Example: ParDo with Side Inputs

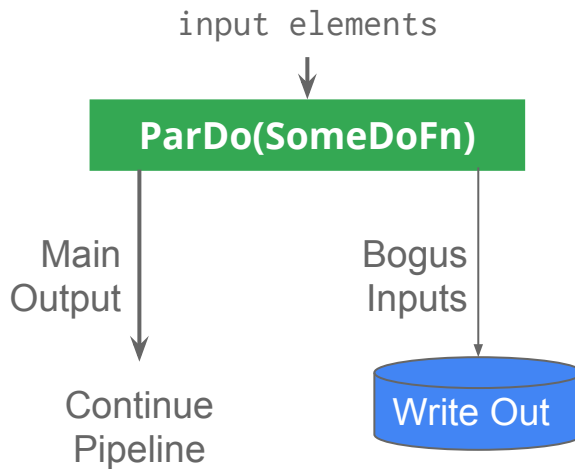
Advanced Topics: Side Outputs

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



```

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);

```

Example: ParDo With Side Outputs

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

Drawing your pipelines

Applying Apache Beam to your problems!

Tips

1. Draw your pipelines first
2. Introduce composite PTransforms for common functionality
3. Prefer `Combine.perKey()/Combine.globally()` and their composites when you know your operation is associative and commutative. They are significantly faster than using `GroupByKey + ParDo`.

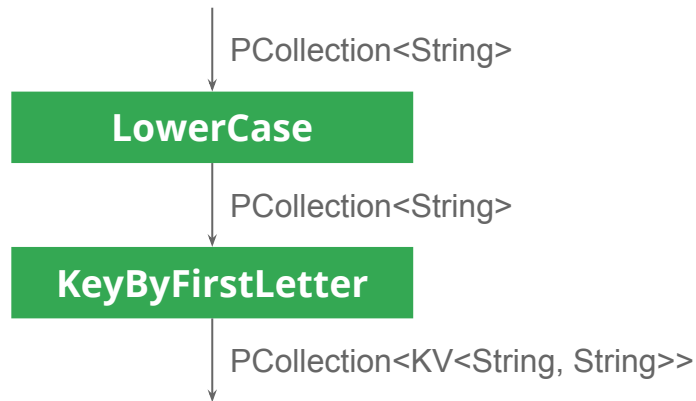
Detour: PCollections and Coders

Each edge in the pipeline is a `PCollection<T>`
`T` identifies the types of elements on that edge

A `Coder<T>` informs the system how to write them to disk and transfer them between machines

Every `PCollection<T>` needs a valid coder in case the runner decides to transfer those values between machines

Encoded keys are also used for grouping in which case the Coder must be deterministic



Detour: What are Bundles?

When running on the Cloud Dataflow service, your data is divided into bundles and distributed for processing

Each DoFn can define startBundle and finishBundle operations to do initialization/finalization

For example -- assembling elements into batches before producing output

```
class MyBatchingDoFn
    extends DoFn<In, List<In>> {
    List<In> batch = new ArrayList<>();

    @ProcessElement
    public void processElement(...) {
        this.batch.add(c.element());
        if (this.batch.size() > 50) {
            c.output(this.batch);
            this.batch = new ArrayList<>();
        }
    }

    public void finishBundle(...) {
        if (this.batch.size() > 0) {
            c.output(this.batch);
        }
    }
}
```

Draw your pipelines

Building a pipeline

We recommend you start with designing and drawing a pipeline

To help you get started we're going to help you draw some pipelines for your own problems

Instructions

Think about a data processing problem that interests you

Thanks for Coming

- Don't forget to tear down your Streaming Jobs!

Extra Slides

Detour: Watermark Propagation

A watermark is a lower bound on future event timestamps that step *should* see

We start at the top (with our PubSub source) with an estimate based on the queued data

Each step has a watermark based on the producing step(s) it consumes and how far behind it is

