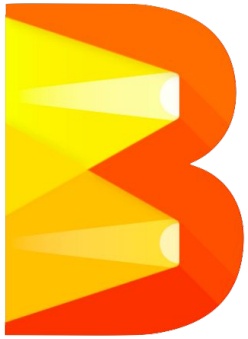
Apache Beam



An advanced unified programming model

Raviprasad M R, Vaibhav Sharma

Introduction

- Beam: Open source, unified model designed to provide batch and streaming data-parallel processing pipelines that are efficient and portable.
- Using one of the Beam SDKs, you build a program that defines the data processing pipeline:

- Pipeline is executed by one of Beam's supported distributed processing back-ends:

Beam Model:

Let’s say we are building a mobile game.

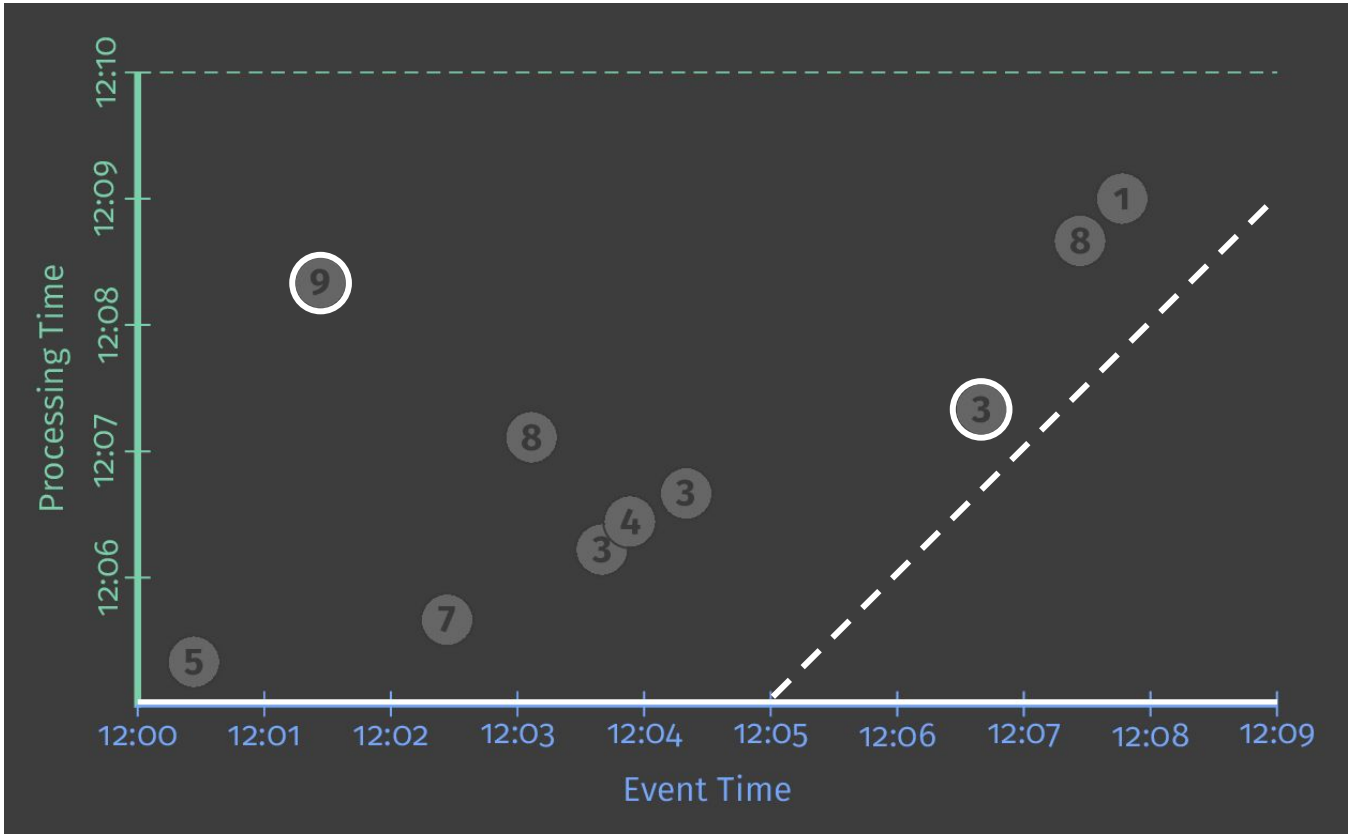
- As game gets popular with many end users, data gets bigger.
- In Beam model, we can write application once and can run it on small / big / organized or infinite data.
- When data becomes infinitely big, we would have streaming system in place.
- Notion of time comes into play in a streaming system: Event time and processing time. In ideal world event is processed as soon as it’s happened.
- But in reality, event may be processed several hours later as well.

Beam Model:

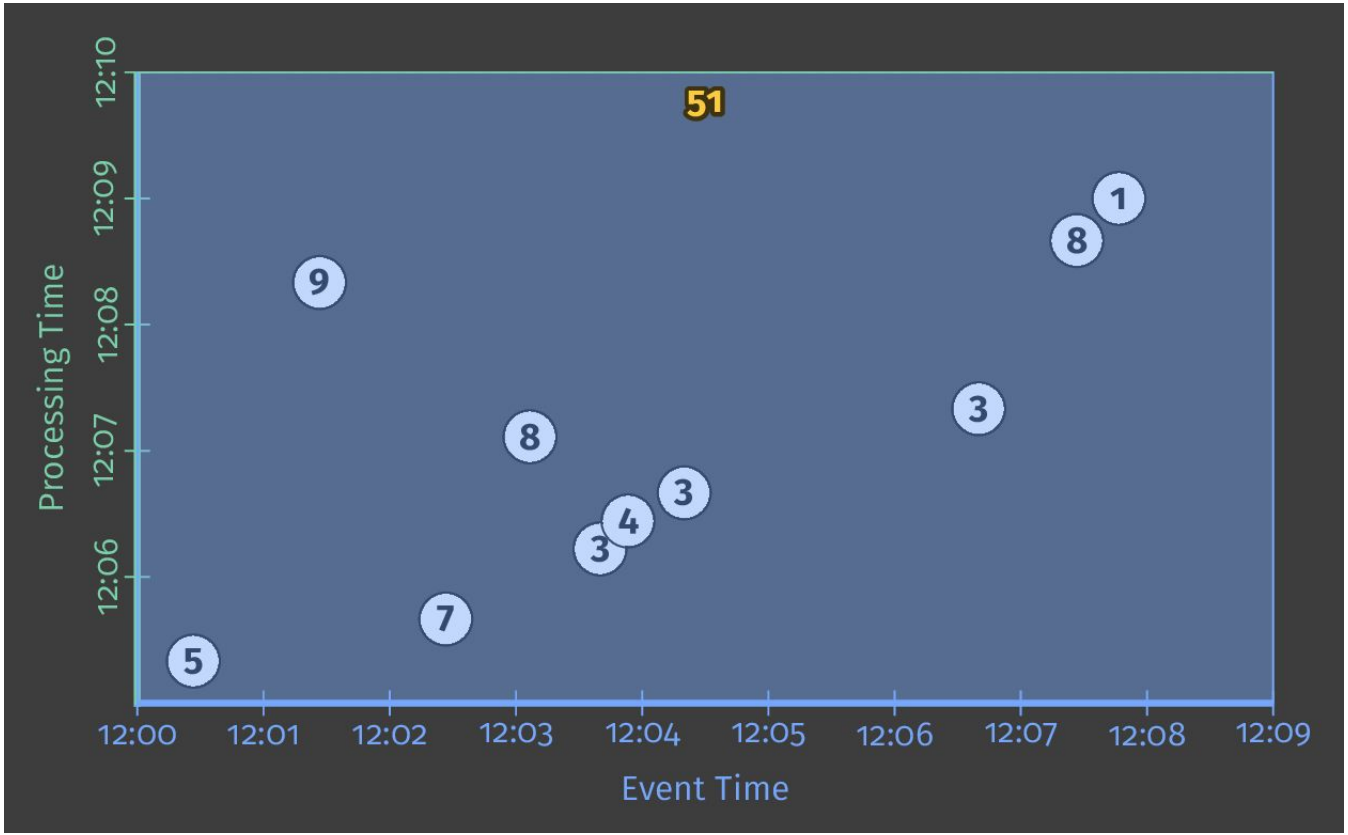
To be able to unify all Use Cases key thing is to separate certain questions into separate APIs.

- What are you computing?
- Where in event time?
- When in processing time?
- How do refinements relate?

What: Computing Integer Sums



What: Computing Integer Sums



Where in event time?

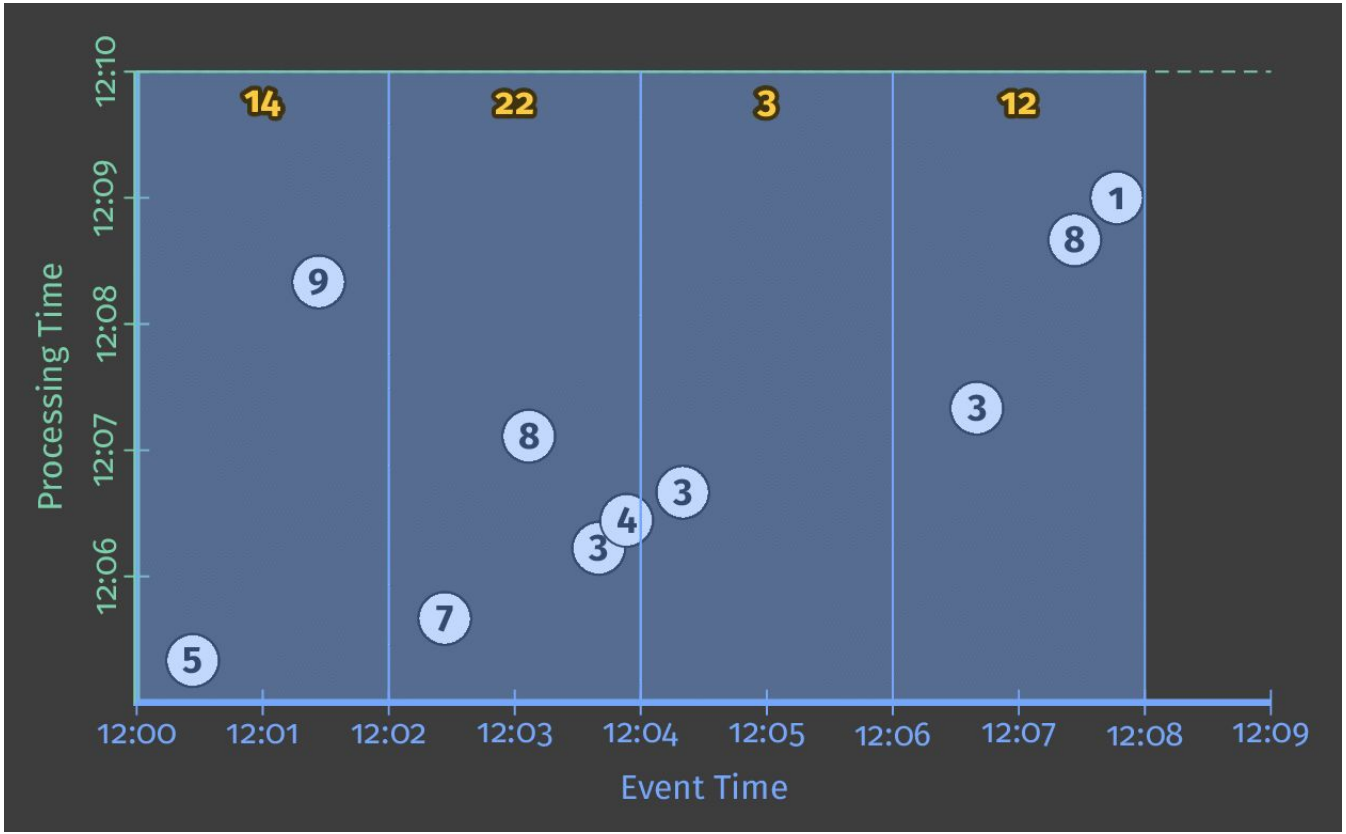
- Windowing lets us create individual results for different slices of event time.
  - Fixed time
  - Sliding windows
  - Session based windows
- Subdivides a PCollection according to the timestamps.
- Transforms works implicitly on a per-window basis.
- Process each PCollection as a succession of multiple, finite windows, though the entire collection itself may be of unbounded size.

Where: Fixed Windowing

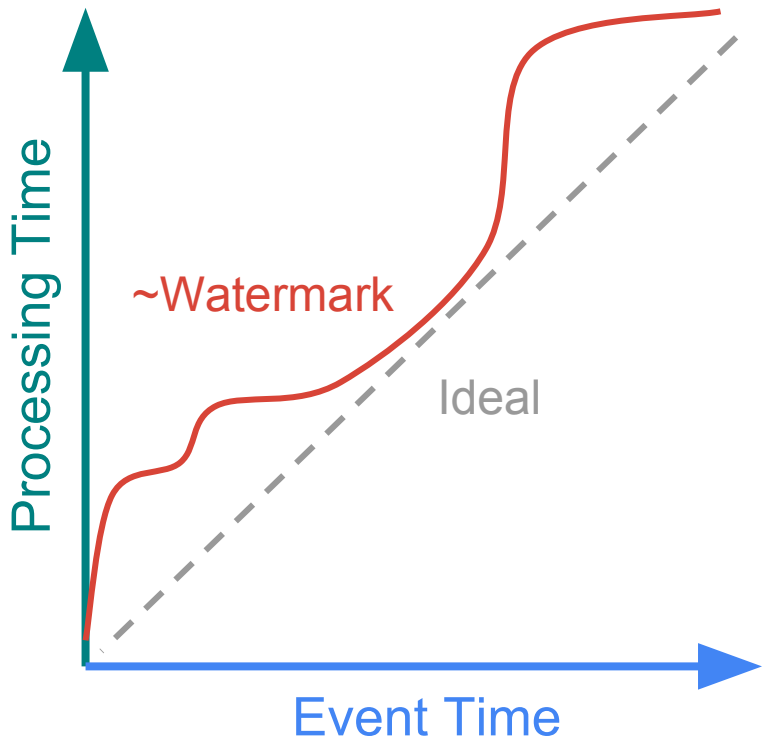
We can achieve this by adding just one line as below:

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

Where: Fixed 2-minute windows.



When in processing time?



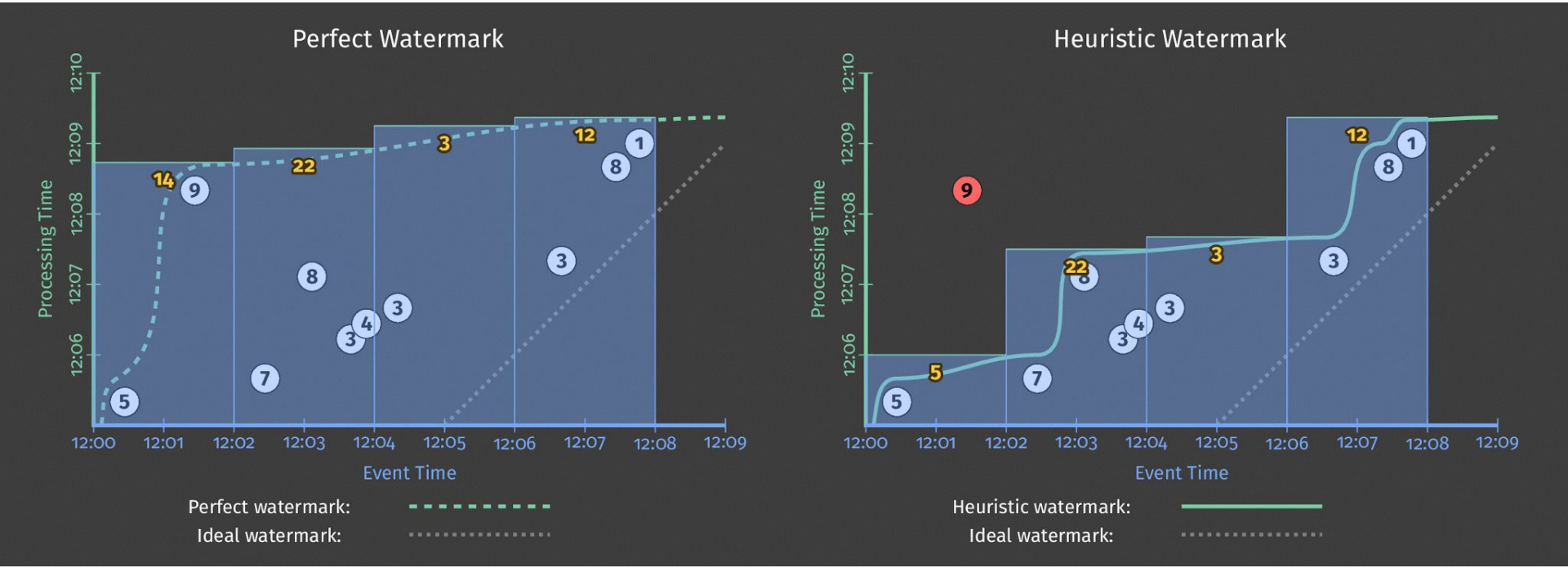
- Watermarks define progress in event time. At current time, the System would have processed all events that happened before watermark.
- Triggers control when results are emitted. Triggers are often relative to watermark.
- With default trigger config, Beam outputs the result when it estimates all data has arrived.

When: Triggering at Watermark

We can achieve this by adding just one line as below:

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

When: Triggering at Watermark.

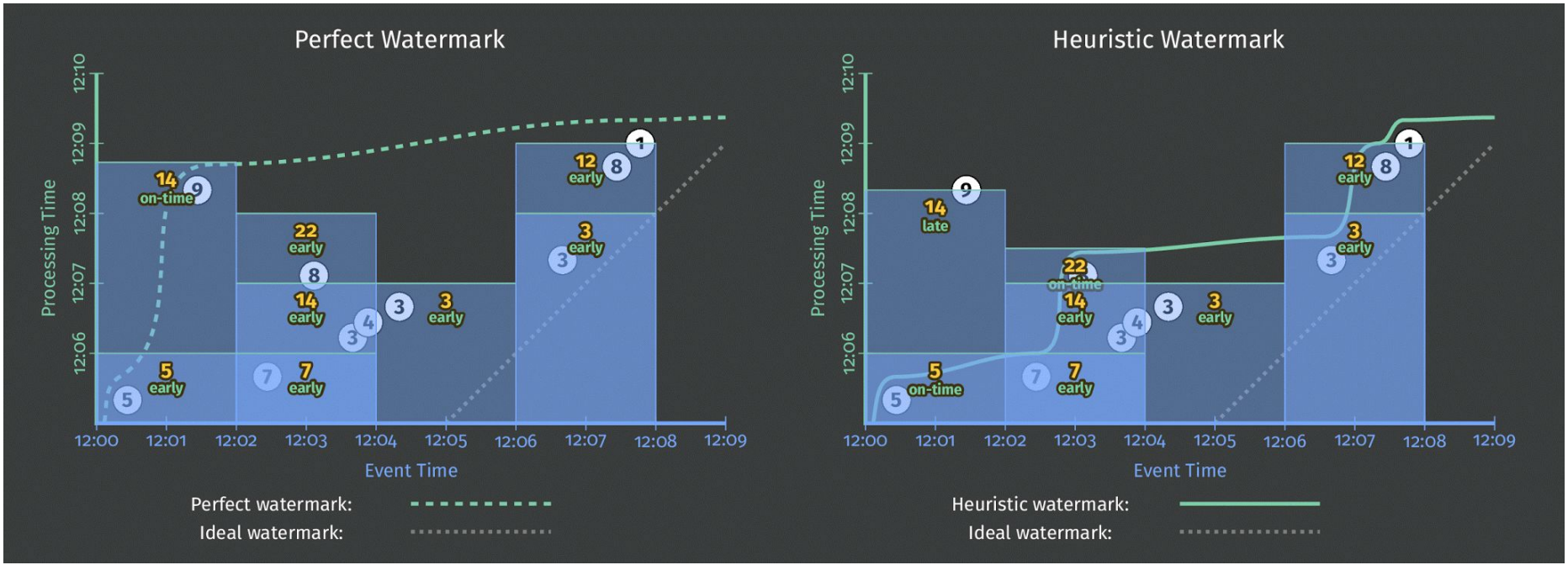


When: Early and Late Firings

We can achieve this by adding just two lines as below:

```
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 and Late Firings



How do refinements relate?

- How should multiple outputs per window accumulate?
- Appropriate choice depends on consumer.

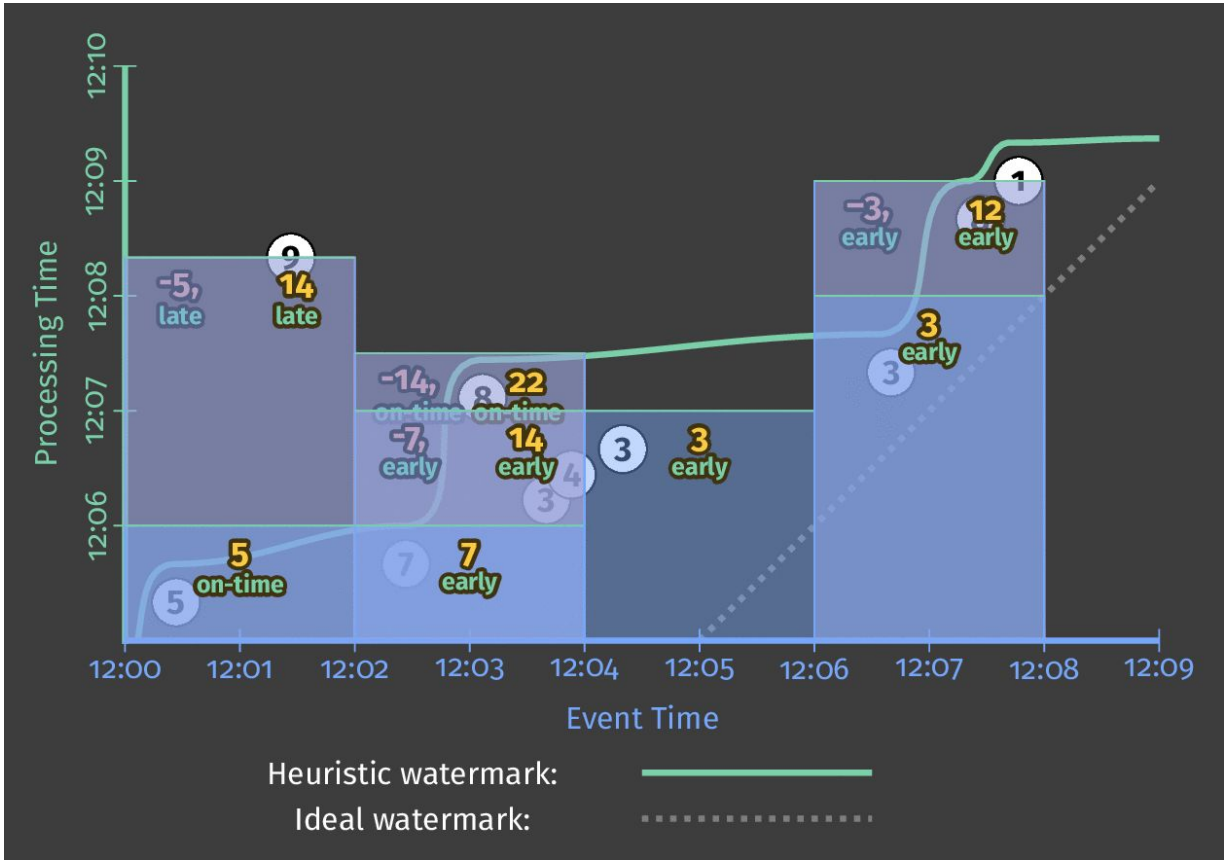
Firing	Elements	Discarding	Accumulating	Acc. & Retracting
Speculative	[3]	3	3	3
Watermark	[5, 1]	6	9	9, -3
Late	[2]	2	11	11, -9
Last Observed		***2***	***11***	***11***
Total Observed		***11***	***23***	***11***

(Accumulating & Retracting not yet implemented. Jira Feature - BEAM-91)

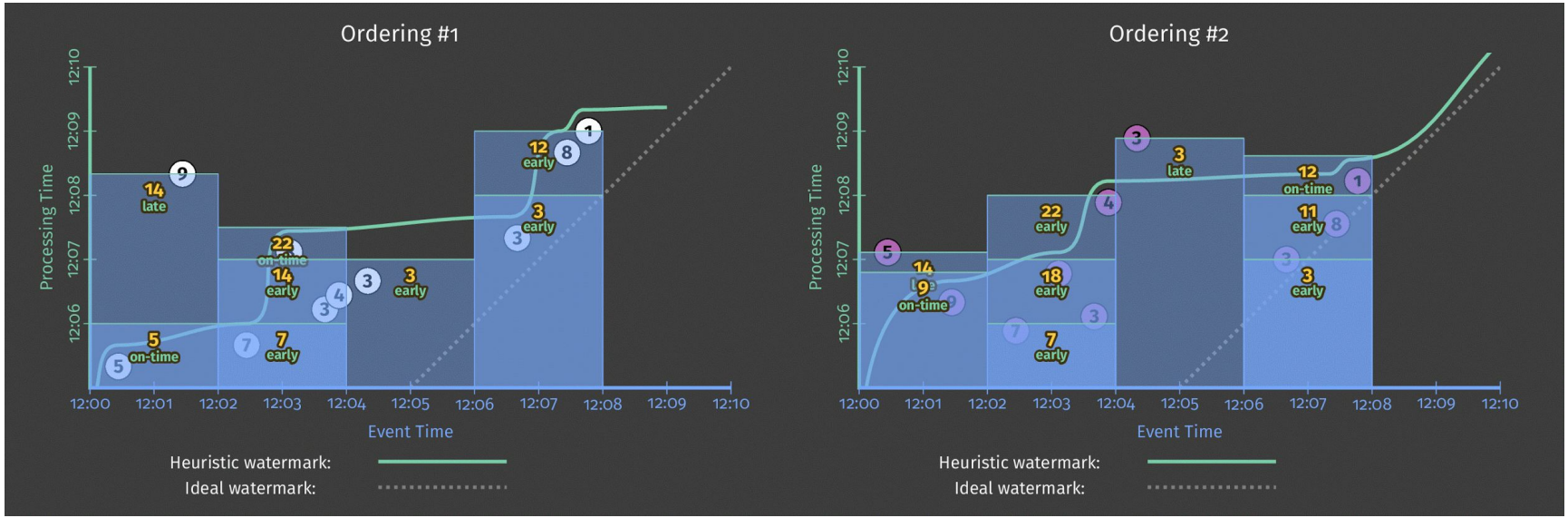
How: Add Newest, remove Previous

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

How: Add Newest, remove Previous



Distributed Systems: Event time results are stable



Uses

- Embarrassingly Parallel data processing tasks.
  The problem can be decomposed into many smaller bundles of data that can be processed independently and in parallel

- Extract, Transform, and Load (ETL) tasks and Pure data integration.
  Moving data between different storage media and data sources, transforming data into a more desirable format, or loading data onto a new system.

Design Concepts: Typical Program Flow

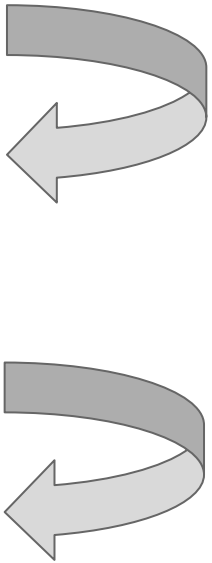
1. Create a Pipeline object.

2. Create an initial PCollection for pipeline data - Using Source API or Create.

3. Apply Transforms to each PCollection

4. Output the final, transformed PCollection(s), typically using the Sink API.

5. Run the pipeline using the designated Pipeline Runner, eg. Apache Flink.



Design Concepts: Pipelines

- Encapsulates the entire data processing task - reading input data, transforming that data, and writing output data

- Using one of the open source Beam SDKs, you build a program that defines the pipeline.

- The pipeline is then executed by one of Beam's supported distributed processing back-ends, eg., Apache Apex, Apache Flink, Apache Spark.

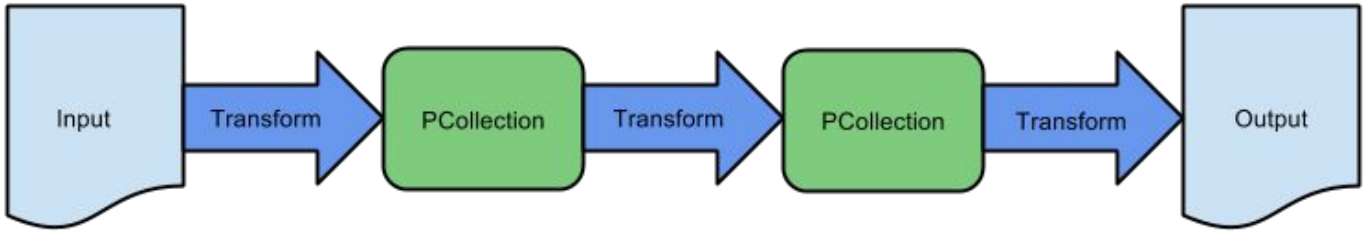
Designing a Pipeline:

- Where is it stored? How many sets?
  Determines the read transform to apply.

- Format?
  Plain Text, Log files, Tables, Key-Value pair...

- What do you want to do with the data? Output format?
  Determines the write transform to apply.

Pipelines: Linear Pipeline



Very simple ‘Linear’ flow of operations.

Pipelines: Multiple transforms

First Transform:

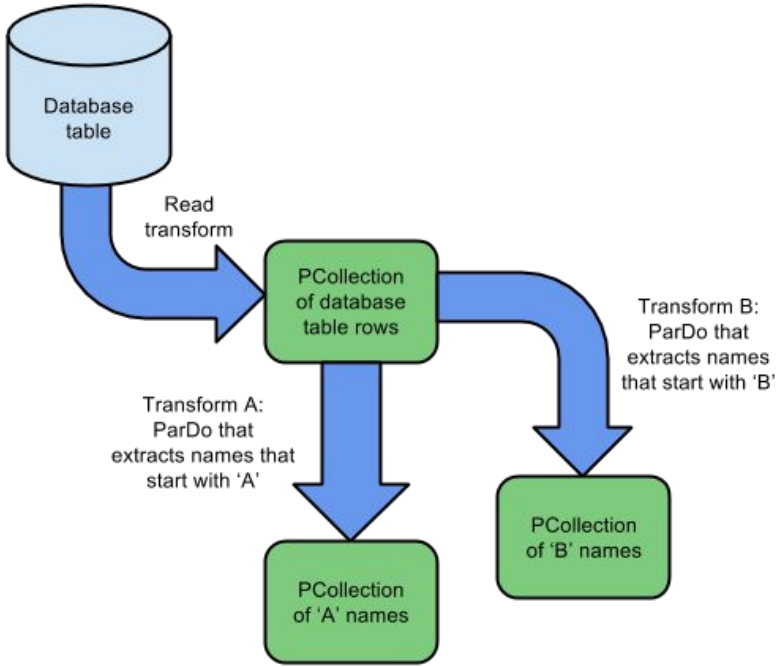
```
if (starts with 'A') { outputToPCollectionA }
```

Second Transform:

```
if (starts with 'B') { outputToPCollectionB }
```

For e.g.:

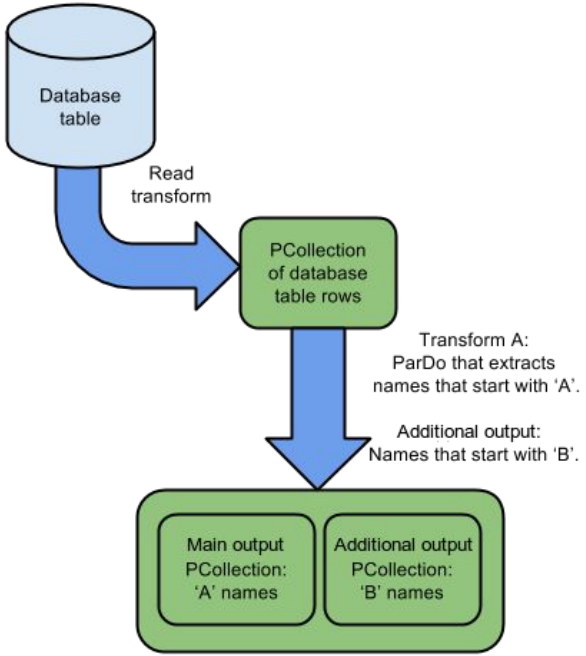
```
PCollection<String> aCollection = dbRowCollection.apply(
"aTrans", ParDo.of(new DoFn<String, String>(){
  @ProcessElement
  public void processElement(ProcessContext c) {
    if(c.element().startsWith("A")){
      c.output(c.element());
    }}
}));
```



Pipelines: Single transform - multiple Outputs

- Uses tagged outputs.
- Only one transform logic.
- Each element in the input PCollection is processed once.
- Using additional outputs makes more sense if the transform's computation per element is time-consuming.

```
if (starts with 'A') { outputToPCollectionA } else if (starts with 'B') {
outputToPCollectionB }
```



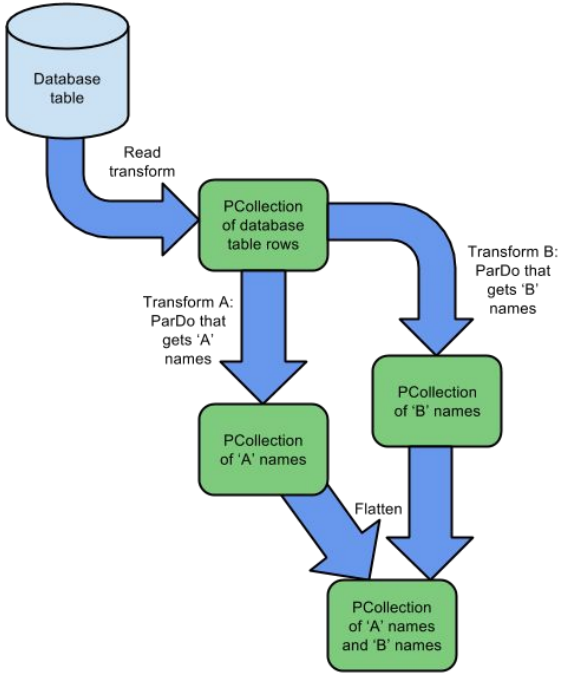
Pipelines: Merging PCollections

- Merge some/all branched PCollections together.
- **Flatten:** Merge PCollections of same type.
- **Join:** CoGroupByKey transform for relational join. PCollections must be key-value pairs and must use same key type.

```
...

PCollection<String> mergedCollectionWithFlatten = collectionList
    .apply(Flatten.<String>pCollections());

...
```



Pipelines: Multiple sources

- Read its input from one or more sources.
- Join the inputs together if data is related.

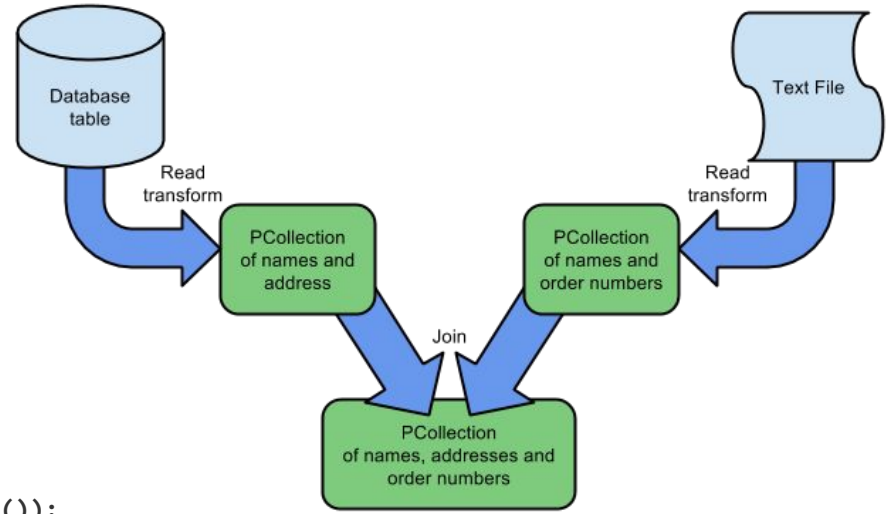
For e.g.:

```
PCollection<KV<String, String>> userAddress =
pipeline.apply(JdbcIO.<KV<String, String>>read()...);

PCollection<KV<String, String>> userOrder =
pipeline.apply(KafkaIO.<String, String>read()...);

...

// Merge collection values into a CoGbkResult collection.
PCollection<KV<String, CoGbkResult>> joinedCollection =
  KeyedPCollectionTuple.of(addressTag, userAddress)
                       .and(orderTag, userOrder)
                       .apply(CoGroupByKey.<String>create());
```



Design Concepts: PCollection

- PCollection is a Distributed collection of data. It is the data as it moves through a Pipeline.
- Usually with a timestamp to each element.
- Can be ‘bounded’ - comes from a fixed source like a file.
- Can be ‘unbounded’ - comes from a continuously updating source.

PCollection of type PCollection<String> of strings from a file:

```
Pipeline p = Pipeline.create();
PCollection<String> pc = p.apply(TextIO.Read.from("/home/me/mybigtextfile.txt"))
```

Design Concepts: Transforms

- Represents a data processing operation, or a step, in your pipeline
- Takes one or more PCollection objects as input, performs a processing function that you provide on the elements.

E.g., Flatten: To combine PCollections of the same type into a single PCollection.

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

Design Concepts: Sources and Sinks

- APIs to represent reading and writing data, respectively.
- Source encapsulates the code necessary to read data into your Beam pipeline from a file or streaming source.
- Sink encapsulates the code necessary to write the elements of a PCollection to an external data sink.

Runners

Beam Runner runs a Beam pipeline on a specific (often distributed) data processing system.

Pipelines can be portable across different runners.

The Capability Matrix provides a detailed comparison of runner functionality.

- DirectRunner (Local Machine)
- ApexRunner (Apache Apex)
- FlinkRunner (Apache Flink)
- SparkRunner (Apache Spark)
- DataflowRunner (Google Cloud Dataflow)
- GearpumpRunner (Apache Gearpump)

Direct Runner

- Suitable for running a Pipeline on local machine on small scale, example, and test data.
- Should be used to validate that pipelines adhere to the Apache Beam model as closely as possible.

Advantages:

1. Using it for Testing and Development ensures that pipelines are robust across different Beam runners.
2. Debugging failed runs is much faster with Direct Runner.
3. Faster and Simpler to perform local unit testing on pipeline code.

Add the below to pom.xml:

```
<artifactId>beam-runners-direct-java</artifactId>
```

Apex Runner

- Executes Apache Beam pipelines using Apache Apex as an underlying Engine.
- The runner has broad support for the Beam model and supports streaming and batch pipelines.
- We can put the data for processing into HDFS.
- Output monitoring: Using the Apex CLI or YARN web UI.

Prerequisites:

- Set up your own Hadoop cluster.
- Can install Apex for monitoring and troubleshooting.

<artifactId>beam-runners-apex</artifactId>

Flink Runner

- Execute a jar file containing your job, on a regular Flink cluster.
- Suitable for large scale, continuous jobs.
- A streaming-first runtime that supports both batch processing and data streaming programs.
- Very high throughput and low event latency at the same time.
- Fault-tolerance with exactly-once processing guarantees.
- Custom memory management.
- Integration with YARN and other components of Hadoop.
- Monitor the job running in http://localhost:8081.

Prerequisites:

- For local execution, no need of any setup.
- For cluster: Set up a Flink Cluster.

<artifactId>beam-runners-flink_2.10</artifactId>

Spark Runner

- Execute Spark pipelines just like a native Spark application.
- Batch and streaming (and combined) pipelines.
- The same fault-tolerance guarantees as provided by RDDs and DStreams.
- The same security features Spark provides.
- Built-in metrics reporting.

- Use spark-submit script when submitting a Spark application to cluster.
- Monitor the job running in http://localhost:4040.

Prerequisites:

- Any Spark version greater than 1.6.0.

<artifactId>beam-runners-spark</artifactId>

Google Cloud Dataflow Runner

- Uses the Cloud Dataflow managed service.
- Runner uploads executable code and dependencies to Cloud Storage bucket, creates a Cloud Dataflow job and executes it.
- Suitable for large scale, continuous jobs.
- Dynamic work rebalancing.
- Autoscaling workers throughout the lifetime of the job.

- Monitor the job using Dataflow Monitoring Interface or Dataflow CLI.

Prerequisites:

- Create a Google Cloud Platform Console project.
- Enable billing for your project.
- Enable required Google Cloud APIs.
- Install the Google Cloud SDK.
- Create a Cloud Storage bucket.

<artifactId>beam-runners-google-cloud-dataflow-java</artifactId>

Advanced Features

- **Portability**
  - You don’t need to worry about the actual runtime where your processes will be deployed and run.
  - Apache Beam provides runners. The runners are responsible for translating your pipelines to the target runtime.
  - You use the same code with different runners.
- **Extensible model and SDK**
  - Most of the Apache Beam parts can be extended to meet your varying environmental needs. Eg. create your own Runner

Advanced Features

- **Unifying batch and streaming**
  - Many systems can handle both batch and streaming, but they often do so via separate APIs.
  - Apache Beam provides the same unified model for batch and stream processing.
  - It's incredibly easy to adjust
- **APIs that raise the level of abstraction**
  - Beam's APIs focus on capturing properties of your data and your logic.

Upcoming / Required Features - Beam Version 2

Some recommendations or feature requests for second edition are as follows:

- ➔ New runners for Apache Hadoop MapReduce, Apache Karaf, and more.
- ➔ Time-based/historical views over datasets
  - ◆ Abstract time-based views over the dataset, such that streaming data is joined
    view, and re-processing of old data gets easy.
- ➔ “drag-and-drop” web interface.
- ➔ More libraries.
- ➔ Partial pipeline that is reusable.

Conclusion

- We firmly believe that the Beam model is the correct programming model for streaming and batch data processing.
- We encourage users to adopt this model for their future data applications.

References:

- Apache Beam Website - https://beam.apache.org/
- Apache Beam @Github - https://github.com/apache/beam
- Google Cloud Platform - https://cloud.google.com
- Introduction to Apache Beam & No shard left behind : APIs for massive parallel efficiency by Dan Halperin: Slides
- Dataflow/Apache Beam - A Unified Model for Batch and Streaming Data Processing by Eugene Kirpichov, Google: Slides
- A Quick Dive into Cloud Data Streaming Technology by Dennis Gannon: Slides
- Apache Beam: Integrating the Big Data Ecosystem Up, Down, and Sideways by Davor Bonaci and Jean-Baptiste Onofré: Slides
- Youtube videos:
    - Hadoop Summit: Apache Beam A Unified Model for Batch and Streaming Data Processing
    - Apache Beam: Portable and Parallel Data Processing (Google Cloud Next '17)
    - Fundamentals of Stream Processing with Apache Beam

Questions?