Scalable Stream Processing
MillWheel and Cloud Dataflow

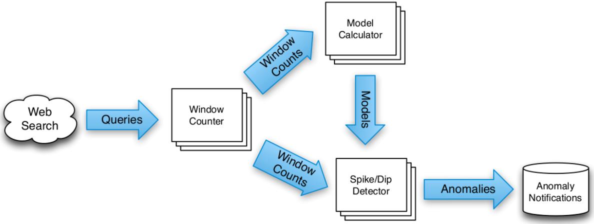
Amir H. Payberah
amir@sics.se

KTH Royal Institute of Technology

MillWheel

Motivation

- Google's Zeitgeist pipeline: tracking trends in web queries
- Ingests a continuous input of search queries and performs anomaly detection.
- Builds a historical model of each query, so that expected changes in traffic.



Requirements

- Persistent storage: shortterm and longterm
- Low watermarks: distinguish late records
- Duplicate prevention

MillWheel Dataflow

- A graph of user-defined transformations (computations) on input data that produces output data.

- Computation actions include:
  - Contacting external systems
  - Manipulating other MillWheel primitives
  - Outputting data

Data Model (1/3)

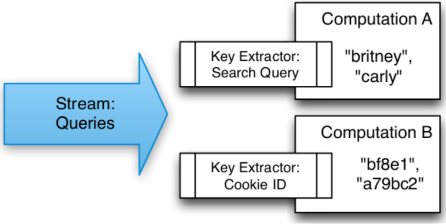
- Stream: the delivery mechanism between computations.
- Inputs and outputs are represented by (key, value, timestamp) triples.

Data Model (1/3)

- Stream: the delivery mechanism between computations.
- Inputs and outputs are represented by (key, value, timestamp) triples.
- Key: a metadata field with semantic meaning in the system.
- Value: an arbitrary byte string, corresponding to the entire record.
- Timestamp: typically wall clock time when the event occurred.

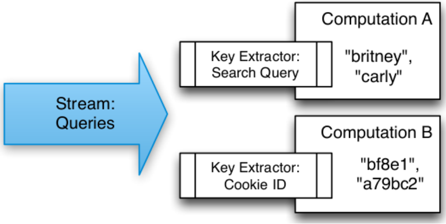
Data Model (2/3)

- Keys are abstraction for record aggregation and comparison.
- Key extraction function: specified by the stream consumer to assign keys to records.



Data Model (2/3)

- Keys are abstraction for record aggregation and comparison.
- Key extraction function: specified by the stream consumer to assign keys to records.
- Computation can only access state for the specific key.



Data Model (3/3)

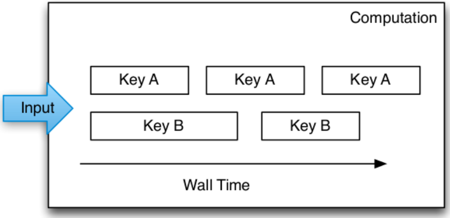
- A computation subscribes to zero or more input streams and publishes one or more output streams.

Data Model (3/3)

- A computation subscribes to zero or more input streams and publishes one or more output streams.
- Multiple computations can extract different keys from the same stream.

Computation (1/3)

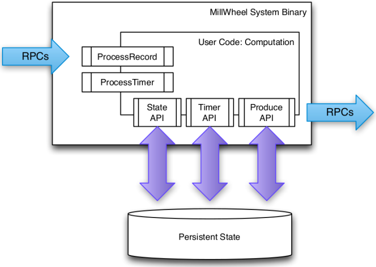
- Application logic lives in computations.
- Users can add and remove computations from a topology dynamically.
- Runs in the context of a single key.
- Parallel per-key processing



Computation (2/3)

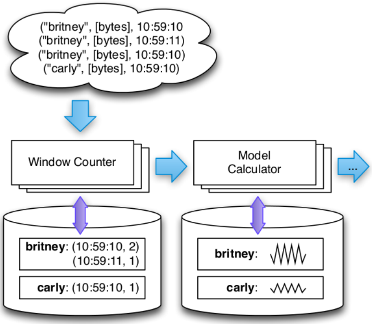
```
class Computation {
  // Hooks called by the system.
  void ProcessRecord(Record data);
  void ProcessTimer(Timer timer);
  ...
};
```

- ProcessRecord
  - Triggered when receiving a record
- ProcessTimer
  - Triggered at a specific value or low watermark value
  - Optional



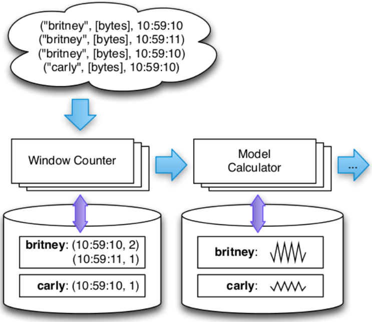
Computation (3/3)

```
// Upon receipt of a record, update the running total for its timestamp bucket,
// and set a timer to fire when we have received all of the data for that bucket.
void Windower::ProcessRecord(Record input) {
  WindowState state(MutablePersistentState());
  state.UpdateBucketCount(input.timestamp());
  string id = WindowID(input.timestamp())
  SetTimer(id, WindowBoundary(input.timestamp()));
}
```



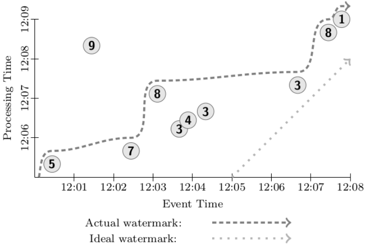
Persistent State

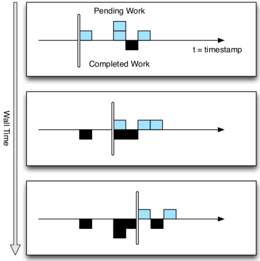
- Managed on per-key basis
- Stored in Bigtable or Spanner
- Common use: aggregation, buffered data for joins, ...



Low Watermarks (1/3)

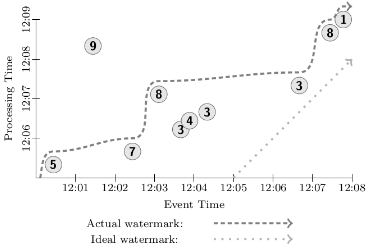
- **Low watermark**: provides a bound on the timestamps of future records arriving at that computation.

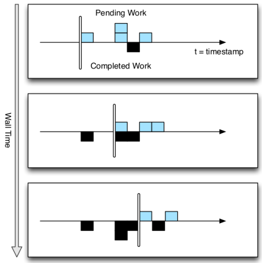




Low Watermarks (1/3)

- Low watermark: provides a bound on the timestamps of future records arriving at that computation.
- Late records: records behind the low watermark.
  - Process them according to application, e.g., discard or correct the result.





Low Watermarks (3/3)

- min(oldest work of A, low watermark of C: C outputs to A)

Low Watermarks (3/3)

- min(oldest work of A, low watermark of C: C outputs to A)

- Low watermark values are seeded by injectors that send data into MillWheel from external systems.

Low Watermarks (3/3)

- Example: a file injector reports a low watermark value that corresponds to the oldest unfinished file.

```
// Upon finishing a file or receiving a new one, we update the low watermark
// to be the minimum creation time.

void OnFileEvent() {
  int64 watermark = kint64max;

  for (file : files) {
    if (!file.AtEOF())
      watermark = min(watermark, file.GetCreationTime());
  }

  if (watermark != kint64max)
    UpdateInjectorWatermark(watermark);
}
```

Fault Tolerance

- Delivery guarantees
- State manipulation

Delivery Guarantees - Exactly-One Delivery

- Upon receipt of an input record in a computation:
  - The duplicated records are discarded.
  - User code is run for the input record.
  - Pending changes are committed to the backing store.
  - Senders are ACKed.
  - Pending downstream productions are sent.

Delivery Guarantees - Strong Productions

- Inputs are not necessarily ordered or deterministic: emitted records are checkpointed before delivery.
  - If an ACK is not received, the record can be re-sent.
  - Duplicates are discarded by MillWheel at the recipient.

Delivery Guarantees - Strong Productions

- Inputs are not necessarily ordered or deterministic: emitted records are checkpointed before delivery.
  - If an ACK is not received, the record can be re-sent.
  - Duplicates are discarded by MillWheel at the recipient.

- The checkpoints allow fault-tolerance.
  - If a processor crashes and is restarted somewhere else any intermediate computations can be recovered.

Delivery Guarantees - Strong Productions

- Inputs are not necessarily ordered or deterministic: emitted records are checkpointed before delivery.
  - If an ACK is not received, the record can be re-sent.
  - Duplicates are discarded by MillWheel at the recipient.

- The checkpoints allow fault-tolerance.
  - If a processor crashes and is restarted somewhere else any intermediate computations can be recovered.

- When a delivery is ACKed the checkpoints can be garbage collected.

Delivery Guarantees - Strong Productions

- Inputs are not necessarily ordered or deterministic: emitted records are checkpointed before delivery.
  - If an ACK is not received, the record can be re-sent.
  - Duplicates are discarded by MillWheel at the recipient.

- The checkpoints allow fault-tolerance.
  - If a processor crashes and is restarted somewhere else any intermediate computations can be recovered.

- When a delivery is ACKed the checkpoints can be garbage collected.

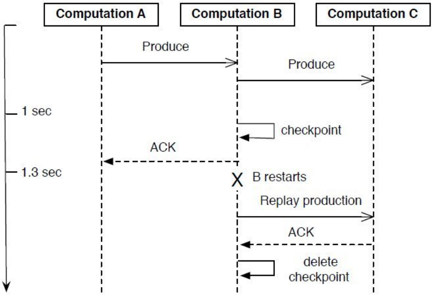
- The Checkpoint→Delivery→ACK→GC sequence is called a strong production.

Delivery Guarantees - Weak Productions (1/2)

- Some computation may be idempotent, regardless of the presence of strong production and exactly-once delivery.
- Disable the exactly-once and/or strong production guarantee for applications that do not need it.
- Weak production is when Millwheel users can allow events to be sent before the checkpoint is committed to persistent storage.

Delivery Guarantees - Weak Productions (2/2)

- Weak production checkpointing prevents straggler productions from occupying undue resources in the sender (Computation A) by saving a checkpoint for receiver (Computation B).



State Manipulation (1/2)

- Hard state: persisted to the backing store.
- Soft state: in-memory caches or aggregates.

State Manipulation (2/2)

- To ensure consistency in hard state, only one bulk write is permitted per event.
    - Wrap all per-key updates in a single atomic operation.

State Manipulation (2/2)

- To ensure consistency in hard state, only one bulk write is permitted per event.
  - Wrap all per-key updates in a single atomic operation.

- To avoid zombie writers (where work has been moved elsewhere through failure detection or through load balancing), every writer has a lease or sequencer that ensures only they may write.

State Manipulation (2/2)

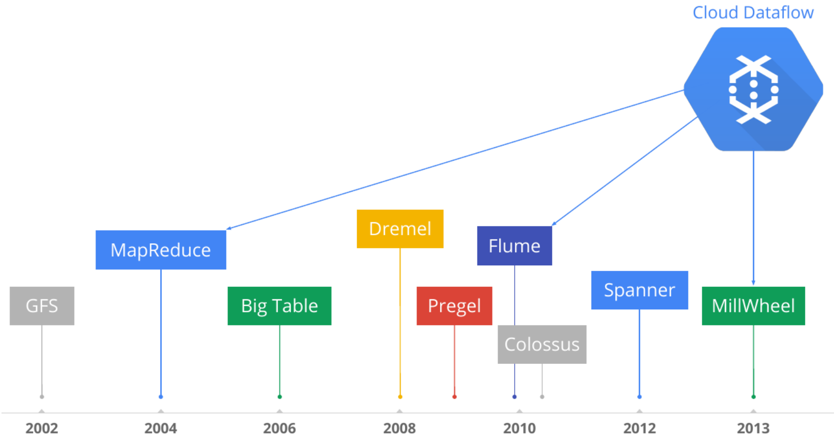
- To ensure consistency in hard state, only one bulk write is permitted per event.
  - Wrap all per-key updates in a single atomic operation.
- To avoid zombie writers (where work has been moved elsewhere through failure detection or through load balancing), every writer has a lease or sequencer that ensures only they may write.
- The single-writer for a key at a particular point in time is critical to the maintenance of soft state.

Google Cloud Dataflow

Google Cloud Dataflow (1/4)

- Google managed service for batch and stream data processing.
- A programming model and execution framework.

Google Cloud Dataflow (2/4)



Google Cloud Dataflow (3/4)

- MapReduce: batch processing
- FlumeJava: dataflow programming model
- MillWheel: handling streaming data

Google Cloud Dataflow (4/4)

- Open source Cloud Dataflow SDK
- Express your data processing pipeline using FlumeJava.

Google Cloud Dataflow (4/4)

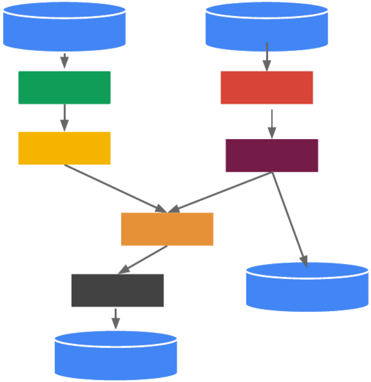
- Open source Cloud Dataflow SDK
- Express your data processing pipeline using FlumeJava.
- If you run your Cloud Dataflow program in batch mode, it is converted to MapReduce operations and run on Google's MapReduce framework.

Google Cloud Dataflow (4/4)

- Open source Cloud Dataflow SDK
- Express your data processing pipeline using FlumeJava.
- If you run your Cloud Dataflow program in batch mode, it is converted to MapReduce operations and run on Google's MapReduce framework.
- If you run the same program in streaming mode, it is executed on the MillWheel stream processing engine.

Programming Model

- Pipeline, a directed graph of data processing transformations
- Optimized and executed as a unit
- May include multiple inputs and multiple outputs
- May encompass many logical MapReduce or Millwheel operations
- PCollections conceptually flow through the pipeline

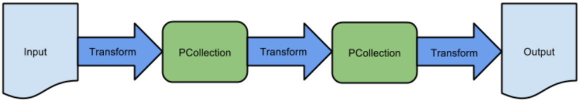


Dataflow Main Components

- Pipelines
- PCollections
- Transforms
- I/O sources and sinks

Pipelines (1/2)

- A pipeline represents a data processing job
- Directed graph of steps operating on data
- A pipeline consists of two parts:
  - Data (PCollection)
  - Transforms applied to that data

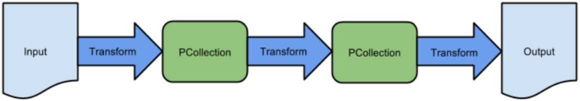


Pipelines (2/2)

```
public static void main(String[] args) {

  // Create a pipeline parameterized by commandline flags.
  Pipeline p = Pipeline.create(PipelineOptionsFactory.fromArgs(arg));

  p.apply(TextIO.Read.from("gs://..."))   // Read input.
   .apply(new CountWords())               // Do some processing.
   .apply(TextIO.Write.to("gs://..."));   // Write output.

  // Run the pipeline.
  p.run();
}
```

PCollections - Overview(1/2)

- A specialized class to represent data in a pipeline.
- A parallel collection of records
- Immutable
- No random access
- Must specify bounded or unbounded



PCollections - Overview (2/2)

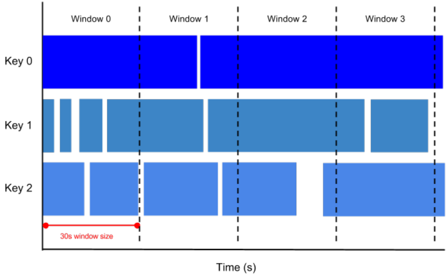
```java
// Create a Java Collection, in this case a List of Strings.
static final List<String> LINES = Arrays.asList(
  "To be, or not to be: that is the question: ",
  "Whether 'tis nobler in the mind to suffer ",
  "The slings and arrows of outrageous fortune, ",
  "Or to take arms against a sea of troubles, ");

PipelineOptions options = PipelineOptionsFactory.create();

Pipeline p = Pipeline.create(options);

// Create the PCollection
p.apply(Create.of(LINES)).setCoder(StringUtf8Coder.of())
```

PCollections - Windowing (1/6)

- Logically divide up or groups the elements of a PCollection into finite windows.
- Each element in a PCollection is assigned to one or more windows.
- Windowing functions:
  - Fixed time windows
  - Sliding time windows
  - Per-session windows

PCollections - Windowing (2/6)

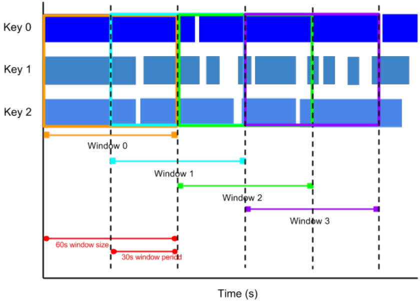
- Fixed time windows
- Represents the time interval in the data stream to define bundles of data, e.g., hourly




```
PCollection<String> items = ...;
PCollection<String> fixed_windowed_items = items.apply(
  Window.<String>into(FixedWindows.of(1, TimeUnit.MINUTES)));
```

PCollections - Windowing (3/6)

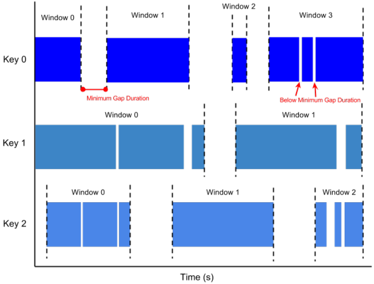
- Sliding time windows
- Uses time intervals in the data stream to define bundles of data, however the windows overlap.




```
PCollection<String> items = ...;
PCollection<String> sliding_windowed_items = items.apply(
    Window.<String>into(SlidingWindows
      .of(Duration.standardMinutes(30))
      .every(Duration.standardSeconds(5))));
```

PCollections - Windowing (4/6)

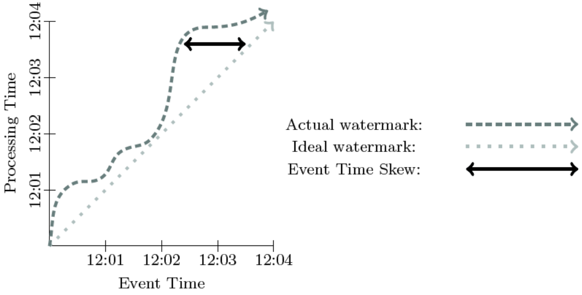
- Session windows
- Defines windows around areas of concentration in the data.
- Useful for data that is irregularly distributed with respect to time, e.g., user mouse activity
- Applies on a per-key basis




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

PCollections - Windowing (5/6)

- Time skew and late data
- Dataflow tracks a watermark: the system's notion of when all data in a certain window can be expected to have arrived in the pipeline.
- Data that arrives with a timestamp after the watermark is considered late data.



PCollections - Windowing (6/6)

- Allow late data by invoking the `withAllowedLateness` operation.

```
PCollection<String> items = ...;
  PCollection<String> fixed_windowed_items = items.apply(
    Window.<String>into(FixedWindows.of(1, TimeUnit.MINUTES))
          .withAllowedLateness(Duration.standardDays(2)));
```

PCollections - Triggers (1/3)

- Determine when to emit elements into an aggregated window.
- Provide flexibility for dealing with time skew and data lag.
  - Example: deal with late-arriving data.
  - Example: get early results, before all the data in a given window has arrived.
- Three main types of triggers:
  - Time-based triggers
  - Data-driven triggers
  - Composit triggers

PCollections - Triggers (2/3)

- Time-base triggers
- Operate on a time reference
  - Event time: as indicated by the timestamp on each data element
  - Processing time: the time when the data element is processed at any given stage in the pipeline

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

PCollections - Triggers (3/3)

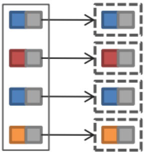
- Data-driven triggers
  - Operate by examining the data as it arrives in each window and firing when a data condition that you specify is met.
  - Example: emit results from a window when that window has received a certain number of data elements.
- Composit triggers
  - Combine multiple time-based or data-driven triggers in some logical way.
  - You can set a composite trigger to fire when all triggers are met (logical AND), when any trigger is met (logical OR), etc.

Transformations - Overview

- A processing operation that transforms data
- Each transform accepts one (or multiple) PCollections as input, performs an operation on the elements in the input PCollection(s), and produces one (or multiple) new PCollections as output.
- Core transforms: `ParDo`, `GroupByKey`, `Combine`, `Flatten`

Transformations - ParDo

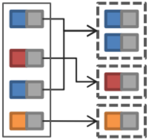
- Processes each element of a PCollection independently using a user-provided DoFn.



```
// The input PCollection of Strings.
PCollection<String> words = ...;

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

// Apply a ParDo to the PCollection "words" to compute lengths for each word.
PCollection<Integer> wordLengths = words.apply(
  ParDo.of(new ComputeWordLengthFn()));
```

Transformations - GroupByKey

- Takes a PCollection of key-value pairs and gathers up all values with the same key.



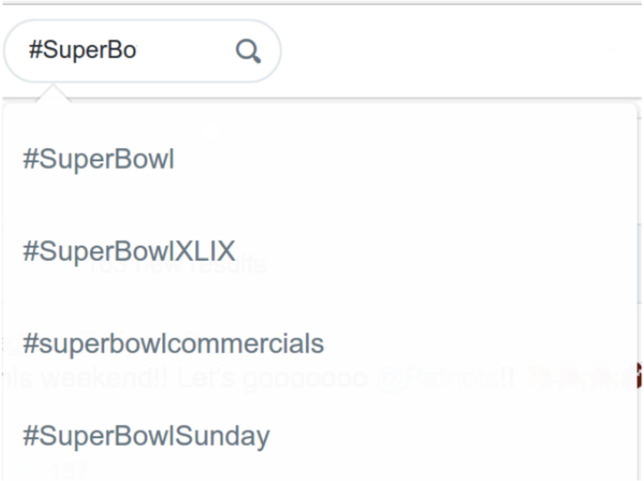
```
// A PCollection of key/value pairs: words and line numbers.
PCollection<KV<String, Integer>> wordsAndLines = ...;

// Apply a GroupByKey transform to the PCollection "wordsAndLines".
PCollection<KV<String, Iterable<Integer>>> groupedWords = wordsAndLines.apply(
  GroupByKey.<String, Integer>create());
```

Transformations - Join and CoGroubByKey

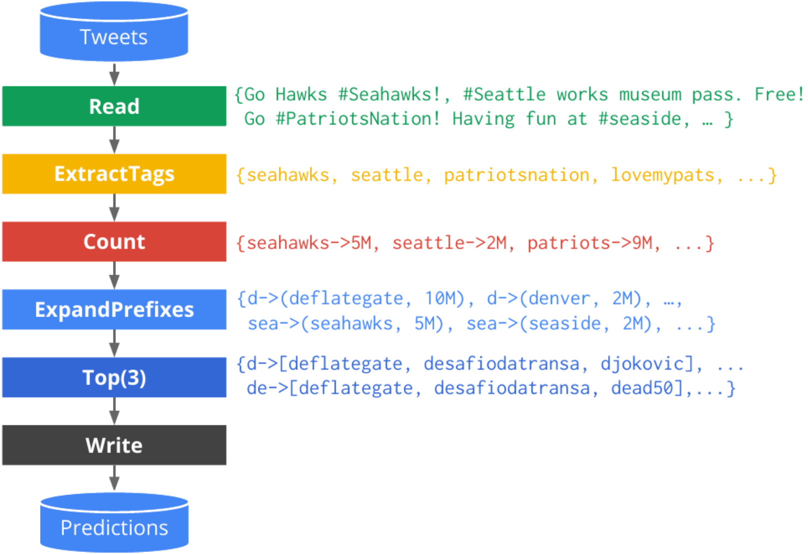
- Groups together the values from multiple PCollections of key-value pairs, where each PCollection in the input has the same key type.

```
// Each data set is represented by key-value pairs in separate PCollections.
// Both data sets share a common key type ("K").
PCollection<KV<K, V1>> pc1 = ...;
PCollection<KV<K, V2>> pc2 = ...;

// Create tuple tags for the value types in each collection.
final TupleTag<V1> tag1 = new TupleTag<V1>();
final TupleTag<V2> tag2 = new TupleTag<V2>();

// Merge collection values into a CoGbkResult collection.
PCollection<KV<K, CoGbkResult>> coGbkResultCollection =
  KeyedPCollectionTuple.of(tag1, pc1)
                       .and(tag2, pc2)
                       .apply(CoGroupByKey.<K>create());
```

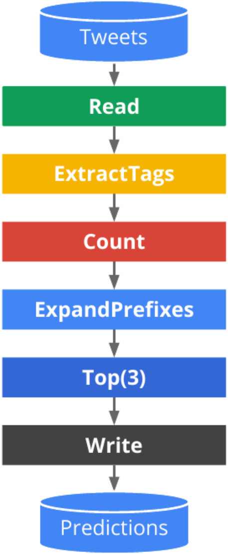
Example: HashTag Autocompletion (1/3)



Example: HashTag Autocompletion (2/3)

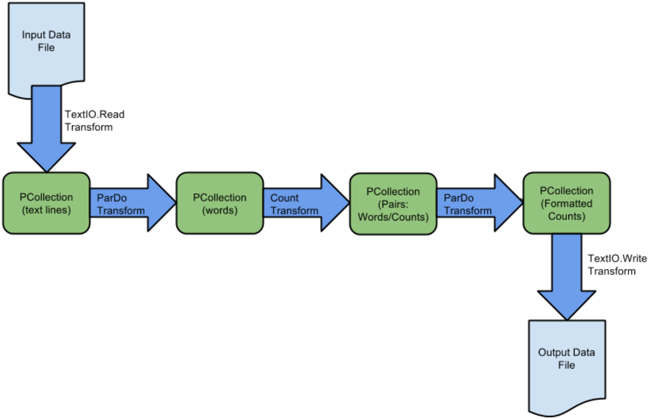


Example: HashTag Autocompletion (3/3)




```
Pipeline p = Pipeline.create();
p.begin()

  .apply(TextIO.Read.from("gs://…"))

  .apply(ParDo.of(new ExtractTags()))

  .apply(Count.perElement())

  .apply(ParDo.of(new ExpandPrefixes())

  .apply(Top.largestPerKey(3))

  .apply(TextIO.Write.to("gs://…"));

p.run();
```

Example: Word Count (1/2)



Example: Word Count (2/2)

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

p.apply(TextIO.Read.from("gs://..."))

  // Apply a ParDo transform to our PCollection of text lines.
  .apply(ParDo.of(new DoFn<String, String>() {
    public void processElement(ProcessContext c) { ... }}))

  // Apply the Count transform to our PCollection of individual words.
  .apply(Count.<String>perElement())

  // Formats our PCollection of word counts into a printable string
  .apply("FormatResults", MapElements...))

  // Apply a write transform
  .apply(TextIO.Write.to("gs://..."));

// Run the pipeline.
p.run();
```

Summary

Summary

- MillWheel
  - DAG of computations
  - Persistent state: per-key
  - Low watermark
  - Exactly-one delivery
- Google cloud dataflow
  - Pipeline
  - PCollection: windows and triggers
  - Transforms

Questions?