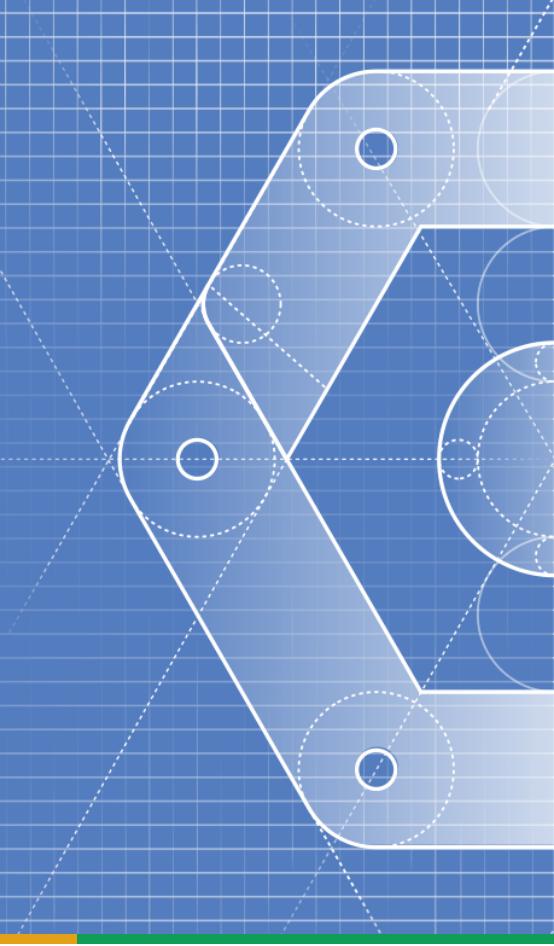
Google Cloud Dataflow

Cosmin Arad, Senior Software Engineer

carad@google.com

August 7, 2015

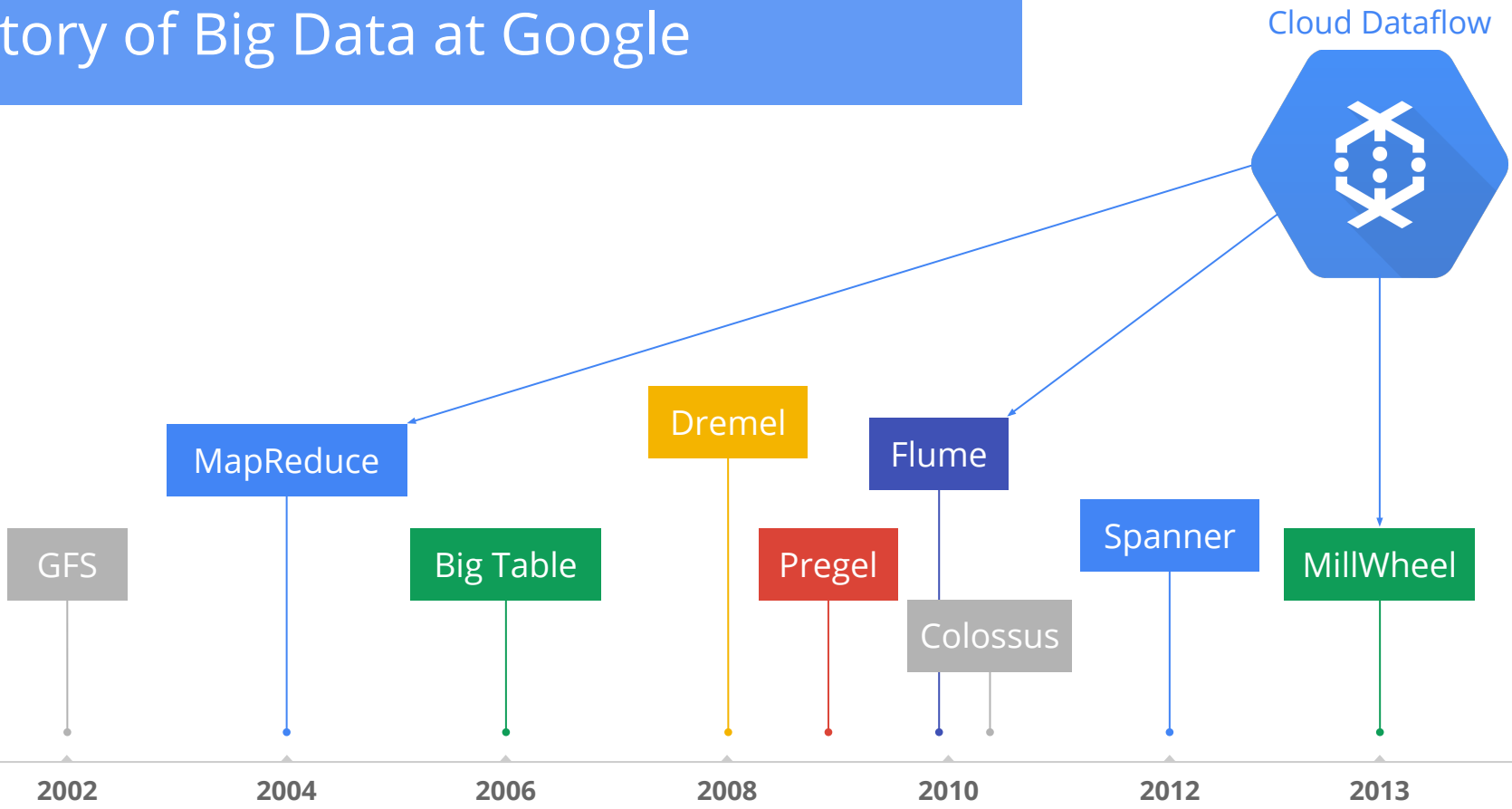


Google Cloud Platform

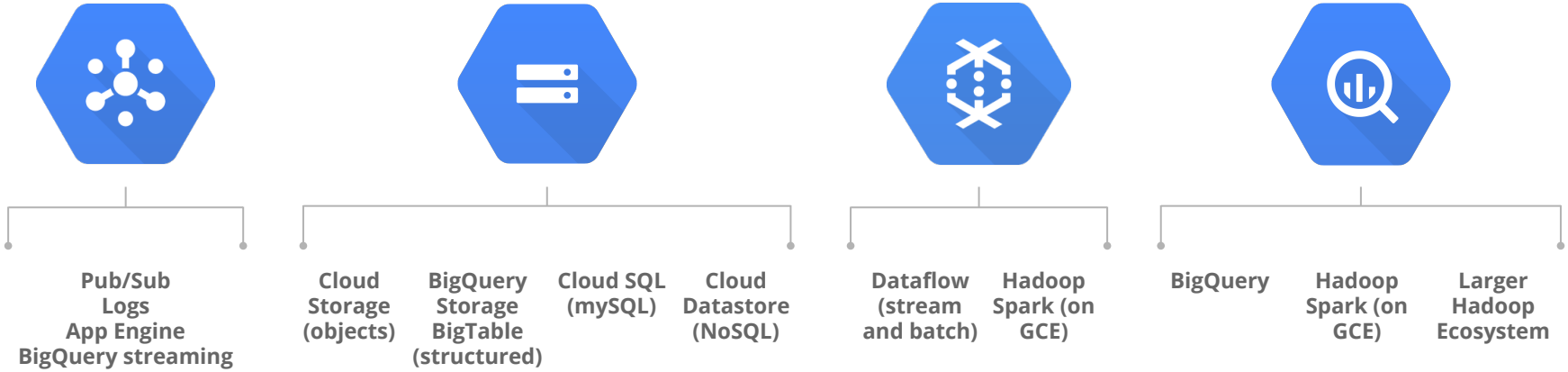
Agenda

1. Dataflow Overview
2. Dataflow SDK Concepts *(Programming Model)*
3. Cloud Dataflow Service
4. Demo: Counting Words!
5. Questions and Discussion

History of Big Data at Google



Big Data on Google Cloud Platform

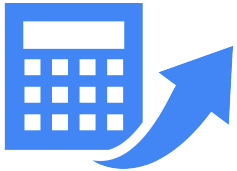


What is Cloud Dataflow?

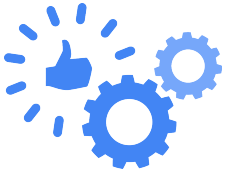
Cloud Dataflow is a collection of **SDK**s for *building* parallelized data processing pipelines

Cloud Dataflow is a managed **service** for *executing* parallelized data processing pipelines

Where might you use Cloud Dataflow?



ETL



Analysis

Orchestration

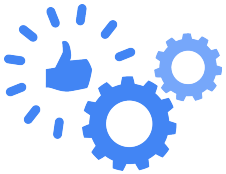
Where might you use Cloud Dataflow?

Movement

Filtering

Enrichment

Shaping



Reduction

Batch computation

Continuous computation

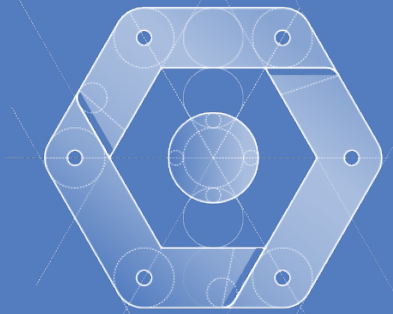
Composition

External orchestration

Simulation

Dataflow SDK Concepts

(Programming Model)



Google Cloud Platform

The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak,
Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills,
Frances Perry, Eric Schmidt, Sam Whittle
Google

{takidau, robertwb, chambers, chernyak, rfernand,
relax, sgmc, millsd, fjp, cloude, samuelw}@google.com

ABSTRACT

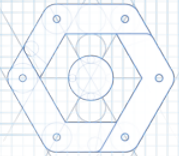
Unbounded, unordered, global-scale datasets are increasingly common in day-to-day business (e.g. Web logs, mobile usage statistics, and sensor networks). At the same time, consumers of these datasets have evolved sophisticated requirements, such as event-time ordering and windowing by features of the data themselves, in addition to an insatiable hunger for faster answers. Meanwhile, practicality dictates that one can never fully optimize along all dimensions of correctness, latency, and cost for these types of input. As a result, data processing practitioners are left with the quandary of how to reconcile the tensions between these seemingly competing propositions, often resulting in disparate implementations and systems.

1. INTRODUCTION

Modern data processing is a complex and exciting field. From the scale enabled by MapReduce [16] and its successors (e.g Hadoop [4], Pig [18], Hive [29], Spark [33]), to the vast body of work on streaming within the SQL community (e.g. query systems [1, 14, 15], windowing [22], data streams [24], time domains [28], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MillWheel, and Storm [5], modern consumers of data wield remarkable amounts of power in shaping and taming massive-scale disorder into organized structures with far greater value. Yet, existing models and systems still fall short in a number of common use cases.

Consider an initial example: a streaming video provider

Dataflow SDK(s)



- Easily construct parallelized data processing pipelines using an intuitive set of programming abstractions
  - Do what the user expects.
  - No knobs whenever possible.
  - Build for extensibility.
  - Unified batch & streaming semantics.
- Google supported and open sourced
  - **Java** 7 (public) @ github.com/GoogleCloudPlatform/DataflowJavaSDK
  - **Python** 2 (in progress)
- Community sourced
  - **Scala** @ github.com/darkjh/scalaflow
  - **Scala** @ github.com/jhlch/scala-dataflow-dsl

Dataflow Java SDK Release Process

Google

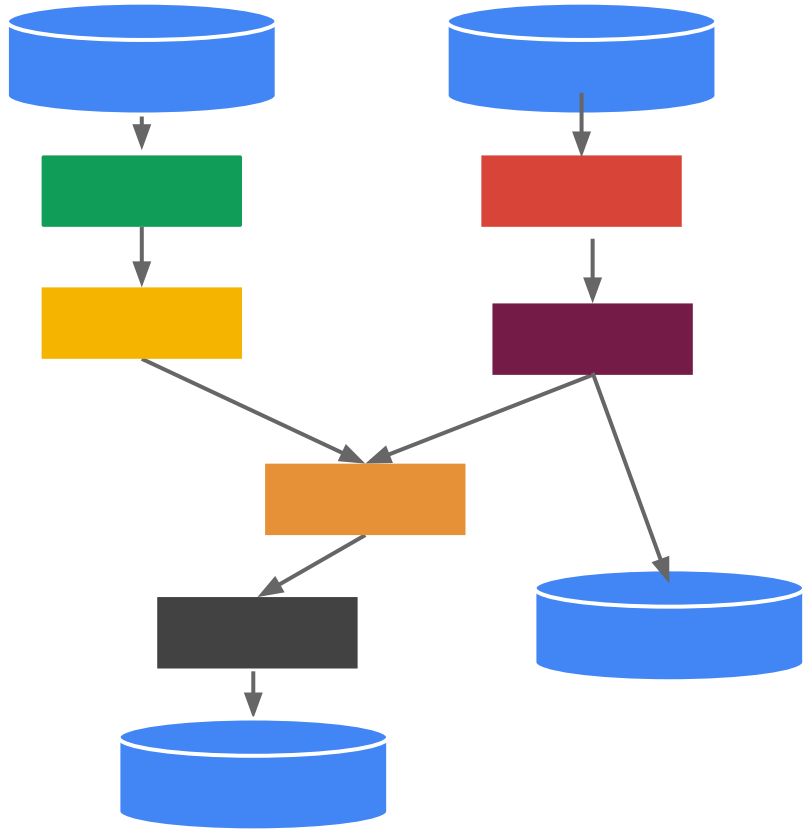
weekly

GitHub

monthly

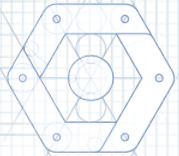
Maven

Pipeline



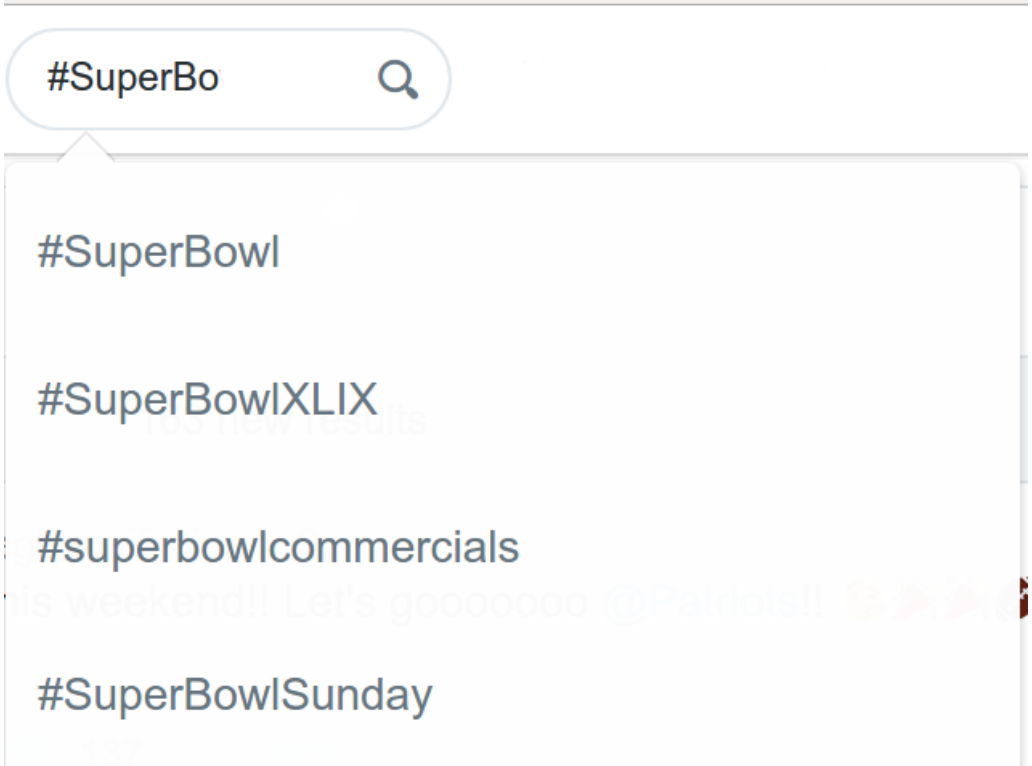
- A directed graph of data processing transformations
- Optimized and executed as a unit
- May include multiple inputs and multiple outputs
- May encompass many logical MapReduce or Millwheel operations
- PCollections conceptually flow through the pipeline

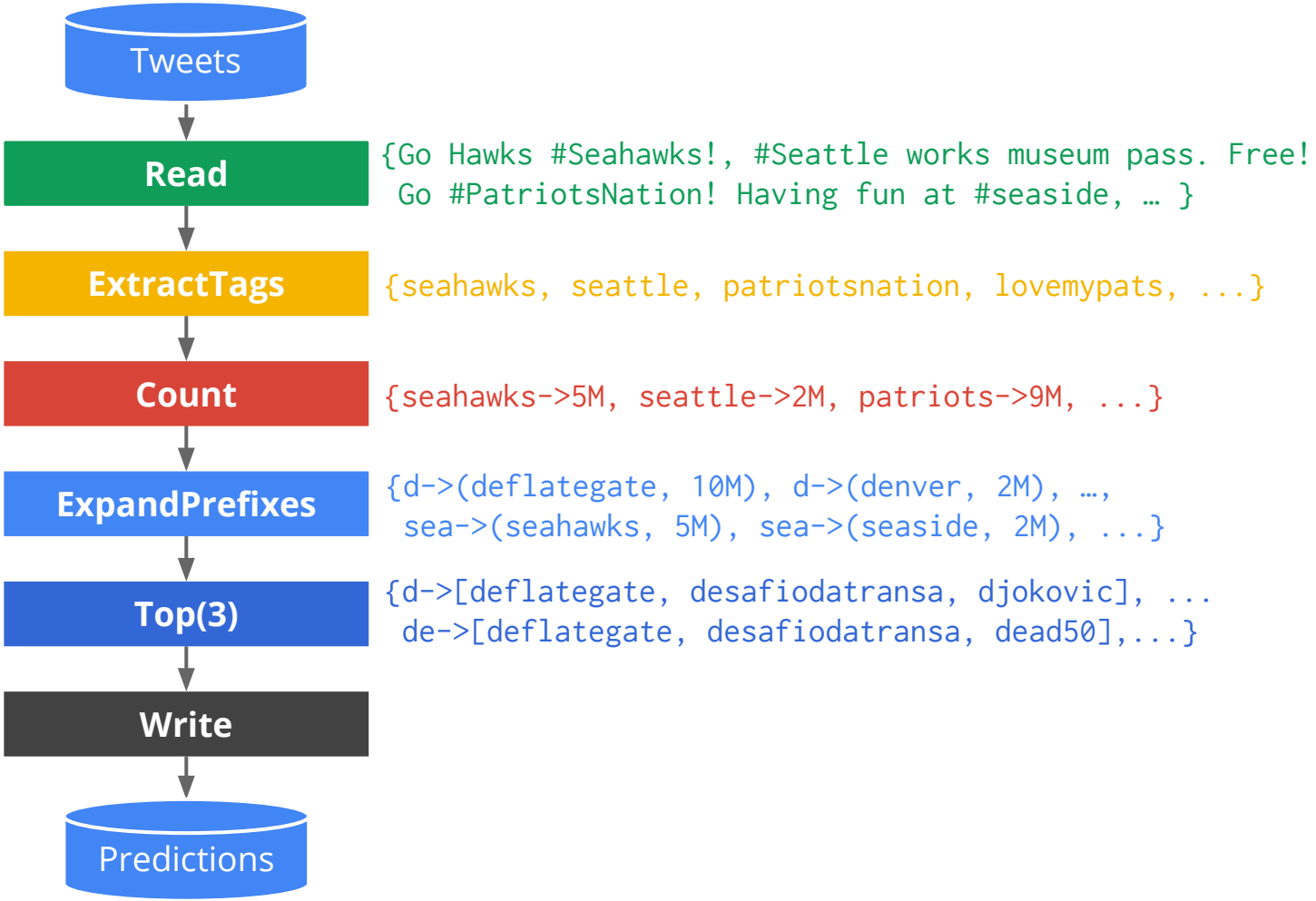
Runners

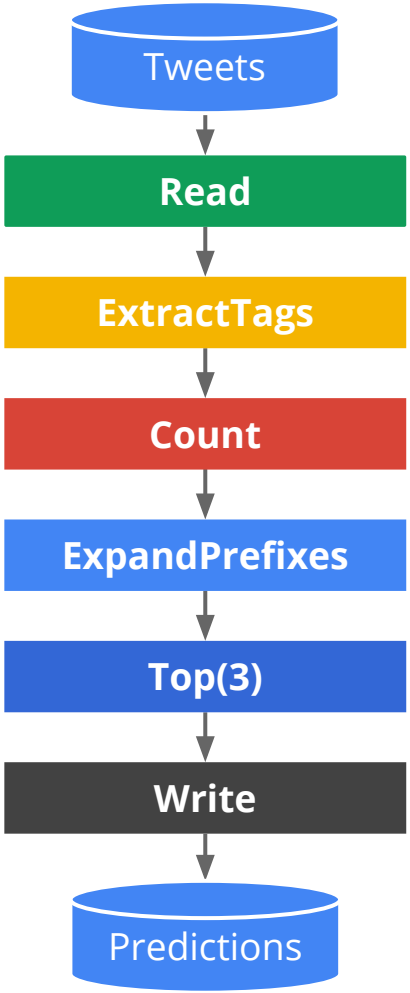


- Specify how a pipeline should run
- Direct Runner
  - For local, in-memory execution. Great for developing and unit tests
- Cloud Dataflow Service
  - **batch** mode: GCE instances poll for work items to execute.
  - **streaming** mode: GCE instances are set up in a semi-permanent topology
- Community sourced
  - **Spark** from Cloudera @ github.com/cloudera/spark-dataflow
  - **Flink** from dataArtisans @ github.com/dataArtisans/flink-dataflow

Example: #HashTag Autocompletion




Tweets
Read
{Go Hawks #Seahawks!, #Seattle works museum pass. Free!
Go #PatriotsNation! Having fun at #seaside, … }
ExtractTags
{seahawks, seattle, patriotsnation, lovemypats, ...}
Count
{seahawks->5M, seattle->2M, patriots->9M, ...}
ExpandPrefixes
{d->(deflategate, 10M), d->(denver, 2M), …,
sea->(seahawks, 5M), sea->(seaside, 2M), ...}
Top(3)
{d->[deflategate, desafiodatransa, djokovic], ...
de->[deflategate, desafiodatransa, dead50],...}
Write
Predictions




```
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();
```

Dataflow Basics

Pipeline

- Directed graph of steps operating on data

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




p.run();
```

Dataflow Basics

Pipeline
- Directed graph of steps operating on data

Data
- PCollection
  - Immutable collection of same-typed elements that can be encoded
- PCollectionTuple, PCollectionList

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

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

  .apply(TextIO.Write.to("gs://…"));

p.run();
```

Dataflow Basics

Pipeline
- Directed graph of steps operating on data

Data
- PCollection
  - Immutable collection of same-typed elements that can be encoded
- PCollectionTuple, PCollectionList

Transformation
- Step that operates on data
- Core transforms
  - ParDo, GroupByKey, Combine, Flatten
- Composite and custom transforms

```
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();
```

Dataflow Basics

```
class ExpandPrefixes … {
    ...
  public void processElement(ProcessContext c) {
    String word = c.element().getKey();
    for (int i = 1; i <= word.length(); i++) {
      String prefix = word.substring(0, i);
      c.output(KV.of(prefix, c.element()));
    }
  }
}
```

```
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();
```

PCollections

- A collection of data of type T in a pipeline
- Maybe be either ***bounded*** or ***unbounded*** in size
- Created by using a PTransform to:
  - Build from a java.util.Collection
  - Read from a backing data store
  - Transform an existing PCollection
- Often contain key-value pairs using KV<K, V>

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

```
{...,
 “NFC Champions #GreenBay”,
 “Green Bay #superbowl!”,
 ...
 “#GoHawks”,
  ...}
```

Inputs & Outputs

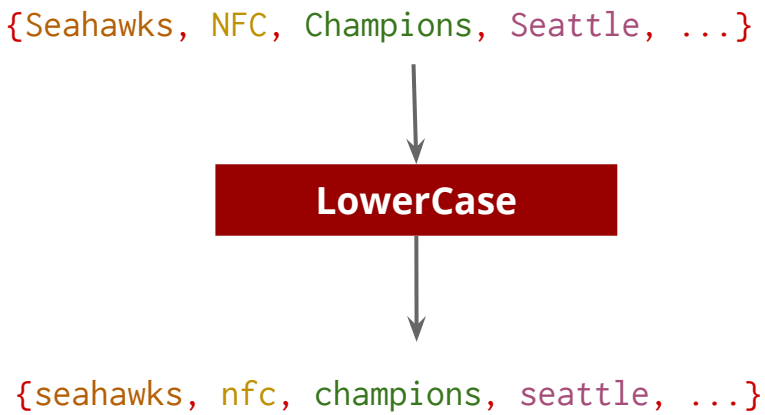
- Read from standard Google Cloud Platform data sources
  - GCS, Pub/Sub, BigQuery, Datastore, ...
- Write your own custom source by teaching Dataflow how to read it in parallel
- Write to standard Google Cloud Platform data sinks
  - GCS, BigQuery, Pub/Sub, Datastore, ...
- Can use a combination of text, JSON, XML, Avro formatted data

Coders

- A Coder<T> explains how an element of type T can be written to disk or communicated between machines
- Every PCollection<T> needs a valid coder in case the service decides to communicate those values between machines.
- Encoded values are used to compare keys -- need to be deterministic.
- Avro Coder inference can infer a coder for many basic Java objects.

ParDo (“Parallel Do”)

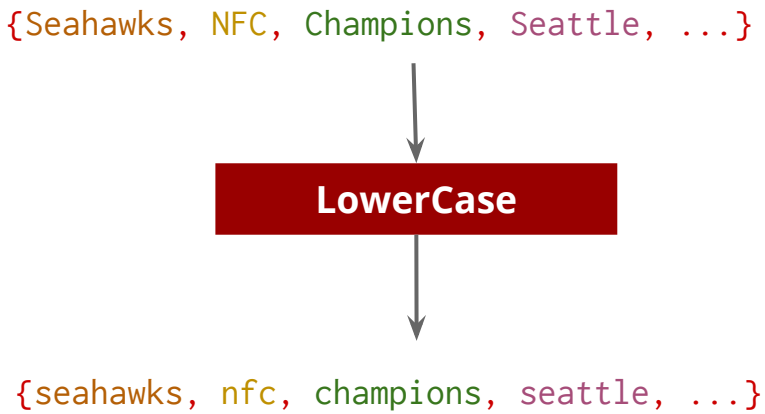
- Processes each element of a PCollection independently using a user-provided DoFn



ParDo (“Parallel Do”)

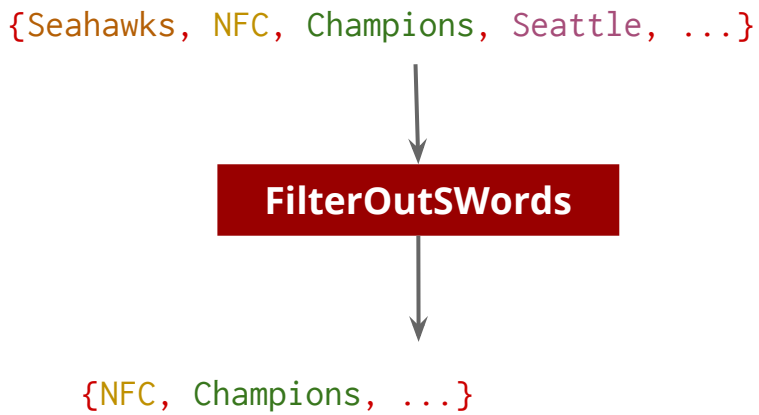
- Processes each element of a PCollection independently using a user-provided DoFn

```
PCollection<String> tweets = …;
tweets.apply(ParDo.of(
  new DoFn<String, String>() {
    @Override
    public void processElement(
        ProcessContext c) {
      c.output(c.element().toLowerCase());
    }));
```



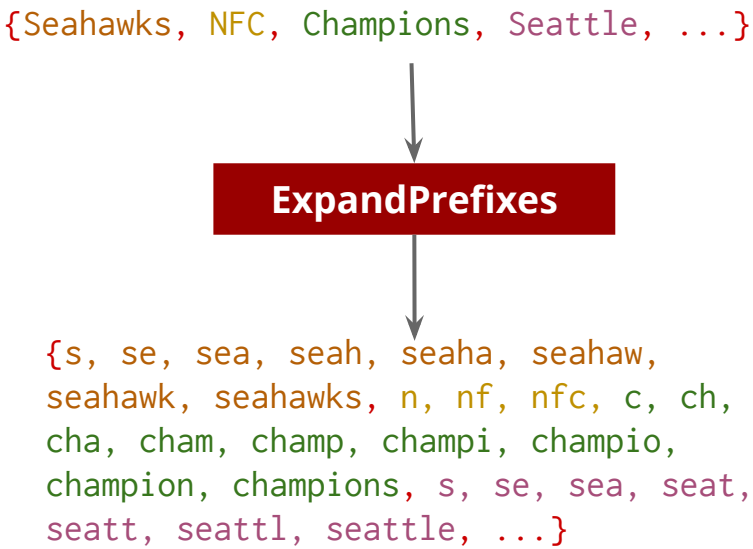
ParDo (“Parallel Do”)

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



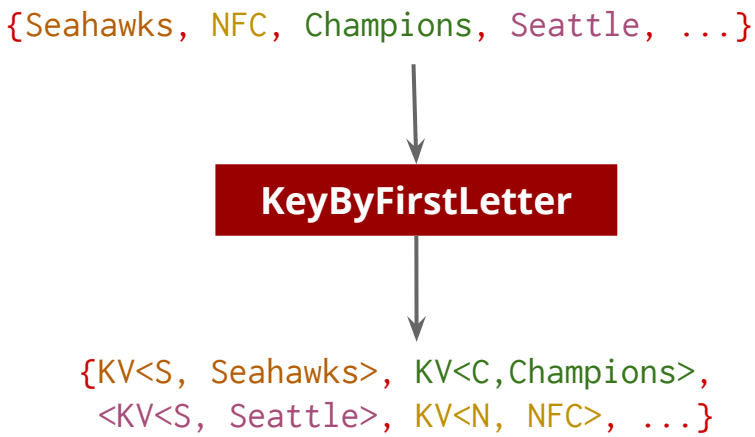
ParDo (“Parallel Do”)

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



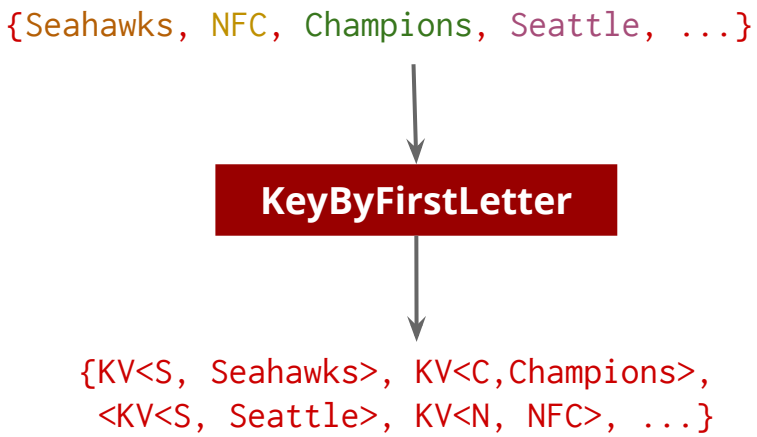
ParDo (“Parallel Do”)

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



ParDo (“Parallel Do”)

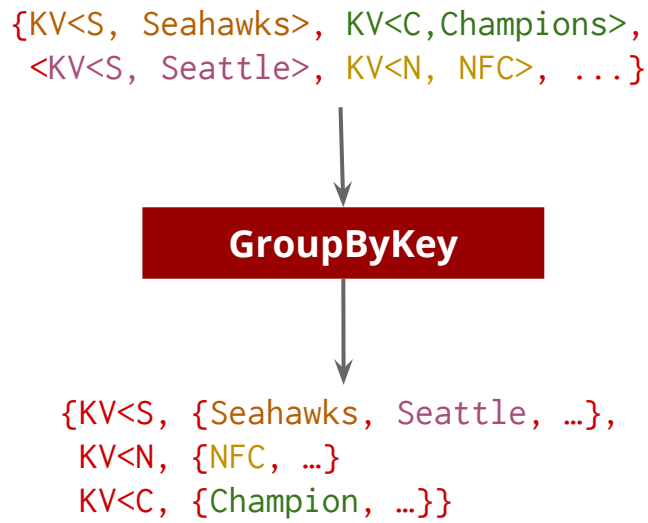
- Processes each element of a PCollection independently using a user-provided DoFn
- Elements are processed in arbitrary ‘bundles’ e.g. “shards”
  - startBundle(), processElement()*, finishBundle()
  - supports arbitrary amounts of parallelization
- Corresponds to both the Map and Reduce phases in Hadoop i.e. ParDo->GBK->ParDo



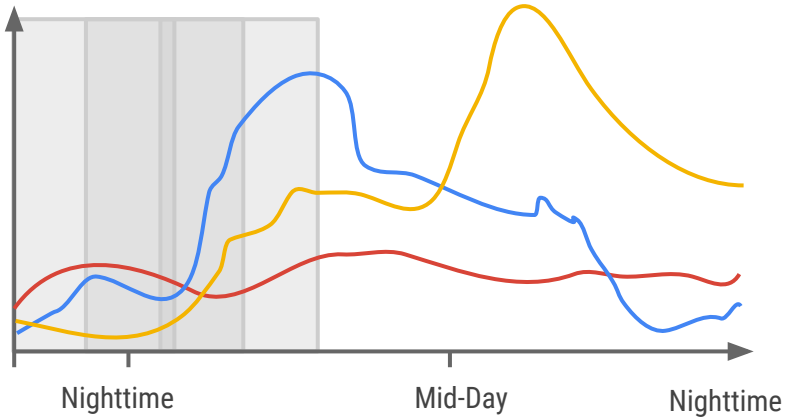
GroupByKey

- Takes a PCollection of key-value pairs and gathers up all values with the same key
- Corresponds to the shuffle phase in Hadoop

How do you do a GroupByKey on an unbounded PCollection?

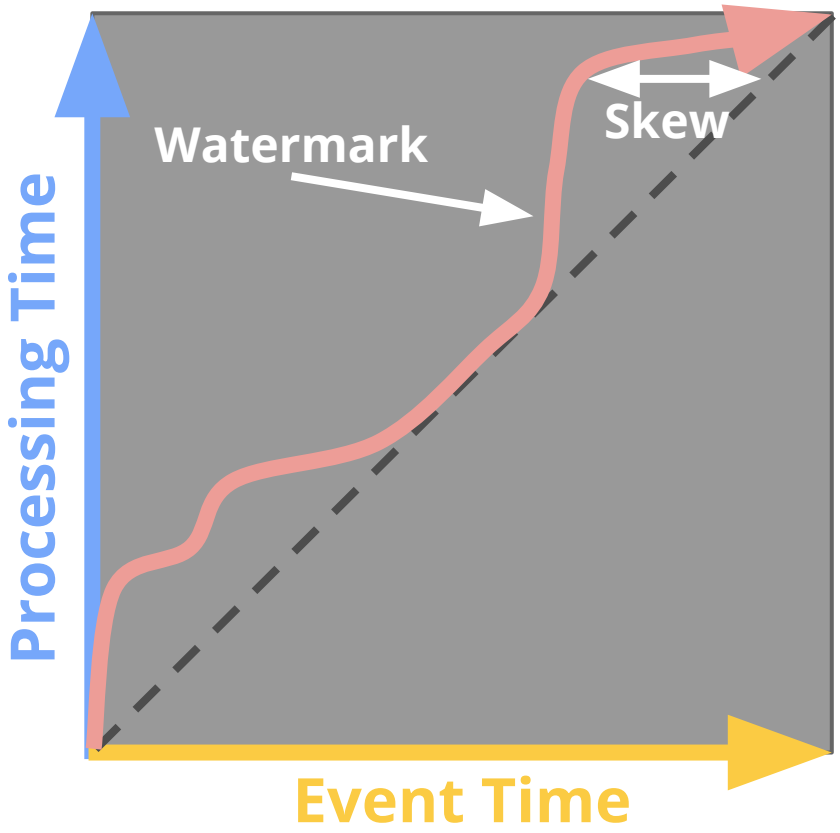


Windows



- Logically divide up or groups the elements of a PCollection into finite windows
  - Fixed Windows: hourly, daily, ...
  - Sliding Windows
  - Sessions
- Required for GroupByKey-based transforms on an unbounded PCollection, but can also be used for bounded PCollections
- Window.into() can be called at any point in the pipeline and will be applied when needed
- Can be tied to arrival time or custom event time

Event Time Skew



Triggers

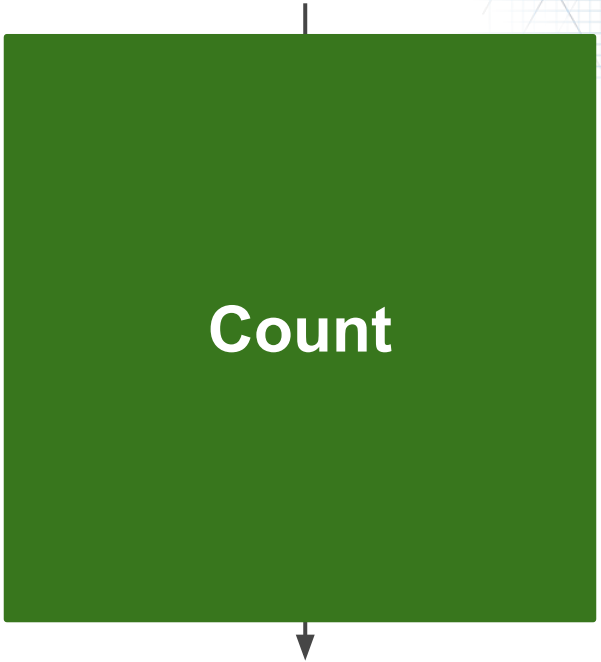
- Determine when to emit elements into an aggregated Window.
- Provide flexibility for dealing with time skew and data lag.
- Example use: Deal with late-arriving data. (Someone was in the woods playing Candy Crush offline.)
- Example use: Get early results, before all the data in a given window has arrived. (Want to know # users per hour, with updates every 5 minutes.)

Late & Speculative Results

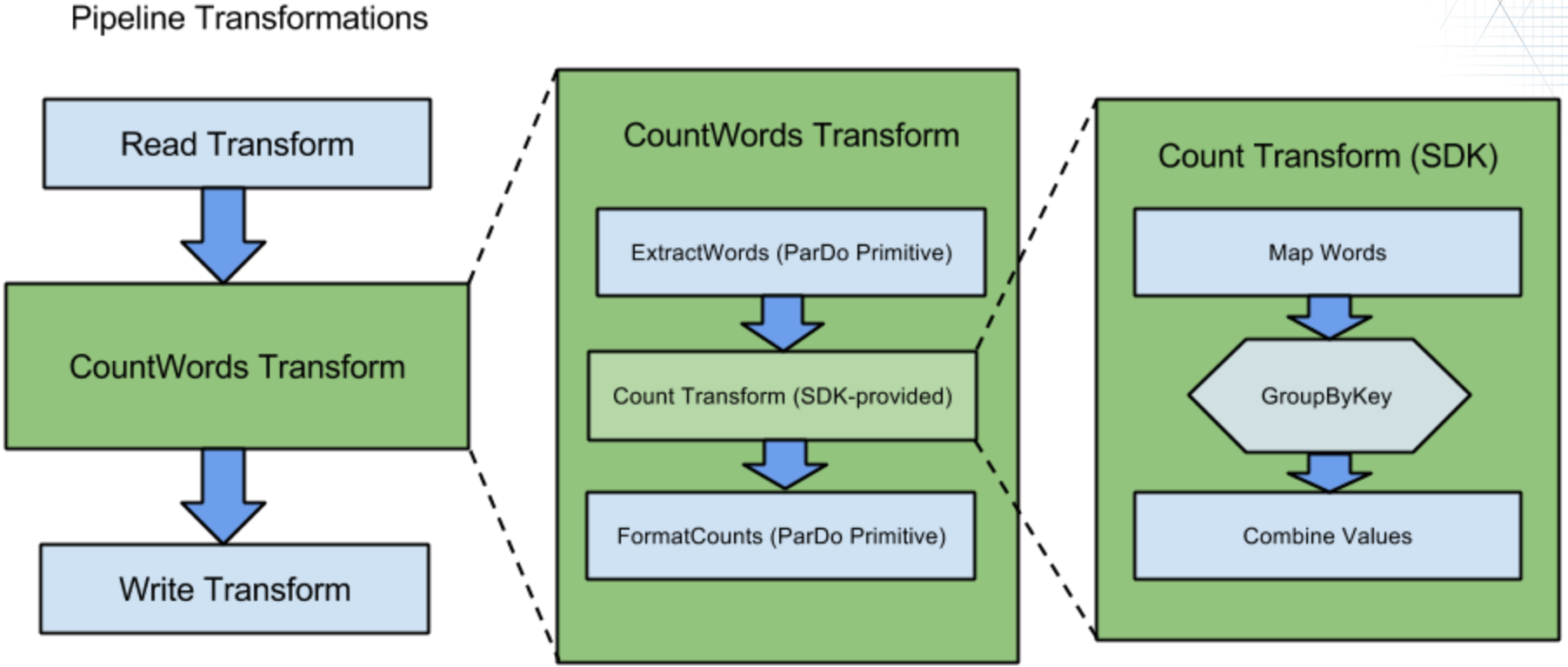
```
PCollection<KV<String, Long>> sums = Pipeline
  .begin()
  .read("userRequests")
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2))
          .triggering(
            AfterEach.inOrder(
              Repeatedly.forever(
                AfterProcessingTime.pastFirstElementInPane()
                  .alignedTo(Duration.standardMinutes(1)))
                .orFinally(AfterWatermark.pastEndOfWindow()),
              Repeatedly.forever(
                AfterPane.elementCountAtLeast(1)))
                .orFinally(AfterWatermark.pastEndOfWindow()
                  .plusDelayOf(Duration.standardDays(7))))
          .accumulatingFiredPanes())
  .apply(new Sum());
```

Composite PTransforms

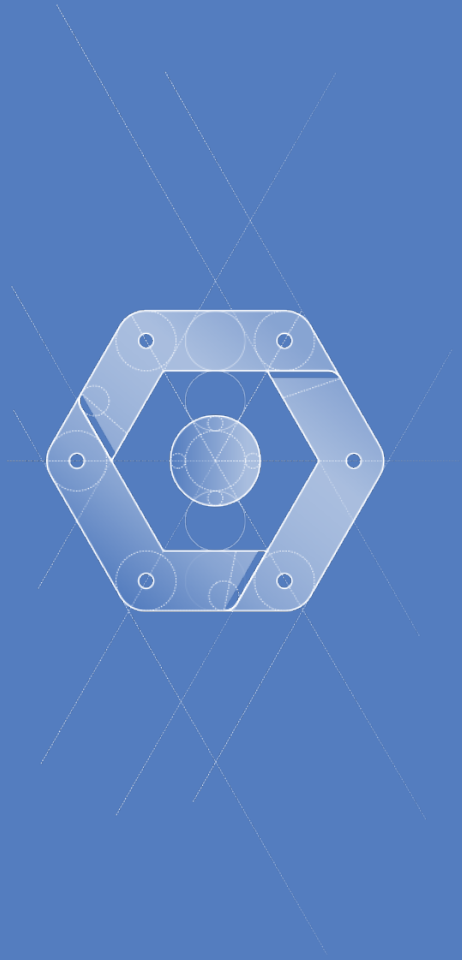
- Define new PTransforms by building up subgraphs of existing transforms
- Some utilities are included in the SDK
  - Count, RemoveDuplicates, Join, Min, Max, Sum, ...
- You can define your own:
  - modularity, code reuse
  - better monitoring experience



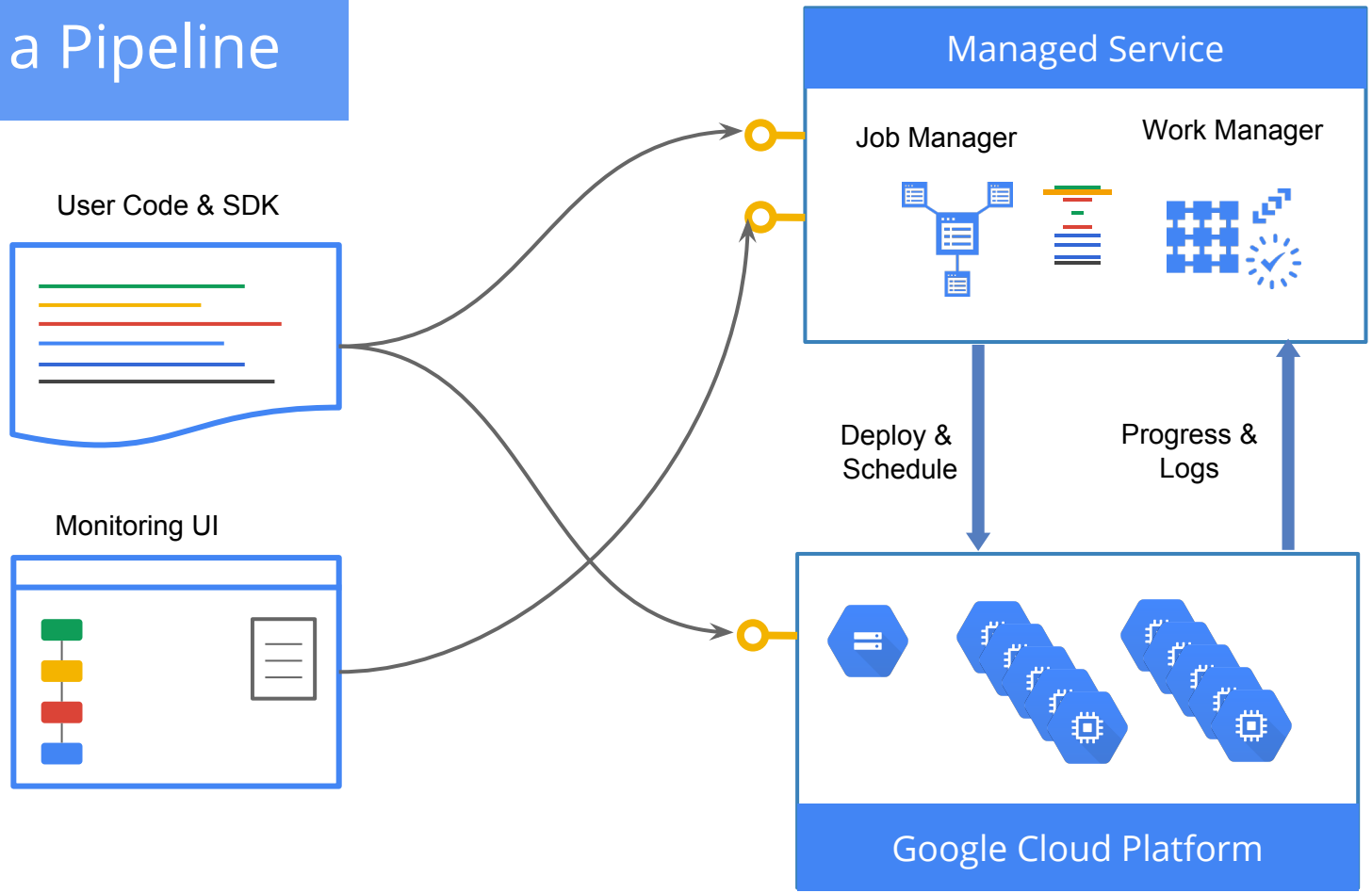
Composite PTransforms



Cloud Dataflow Service



Life of a Pipeline



Cloud Dataflow Service Fundamentals

- **Pipeline optimization**: Modular code, efficient execution
- **Smart Workers**: Lifecycle management, Auto-Scaling, and Dynamic Work Rebalancing
- **Easy Monitoring**: Dataflow UI, Restful API and CLI, Integration with Cloud Logging, Cloud Debugger, etc.

Graph Optimization

ParDo fusion

- Producer Consumer
- Sibling
- Intelligent fusion boundaries

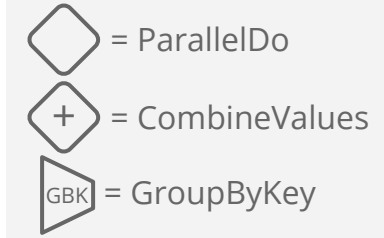
Combiner lifting e.g. partial aggregations before reduction

Flatten unzipping

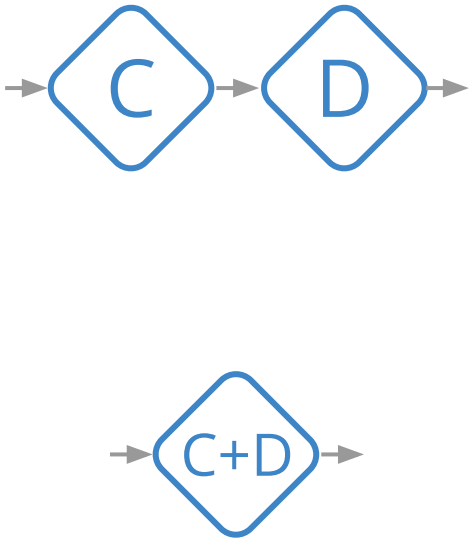
Reshard placement

...

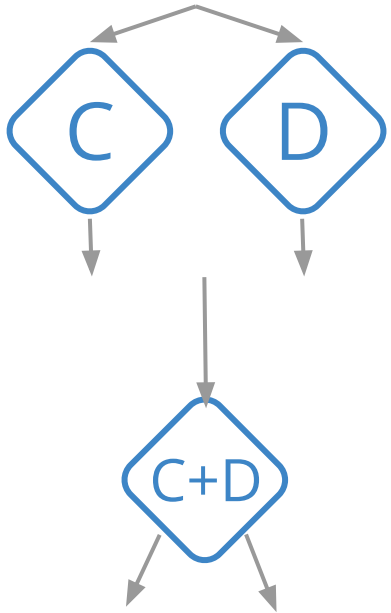
Optimizer: ParallelDo Fusion



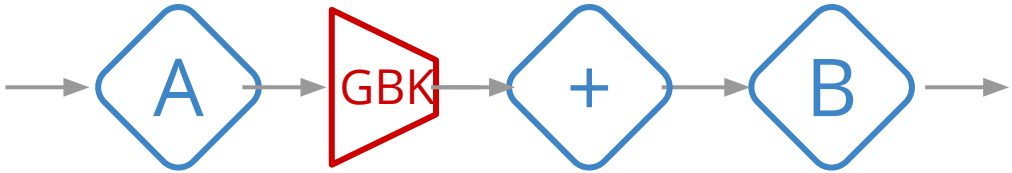
consumer-producer

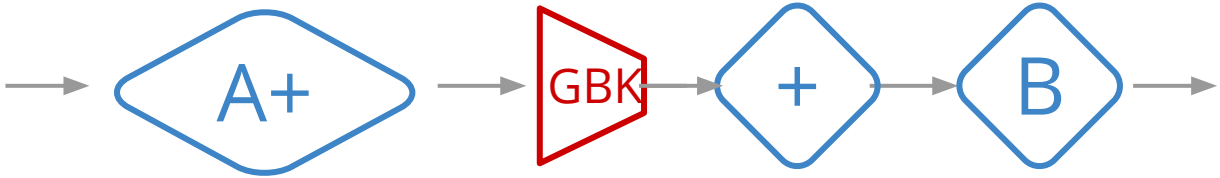


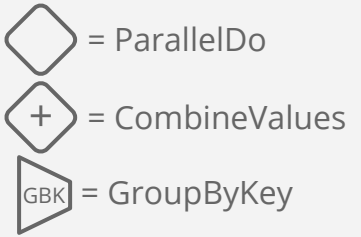
sibling

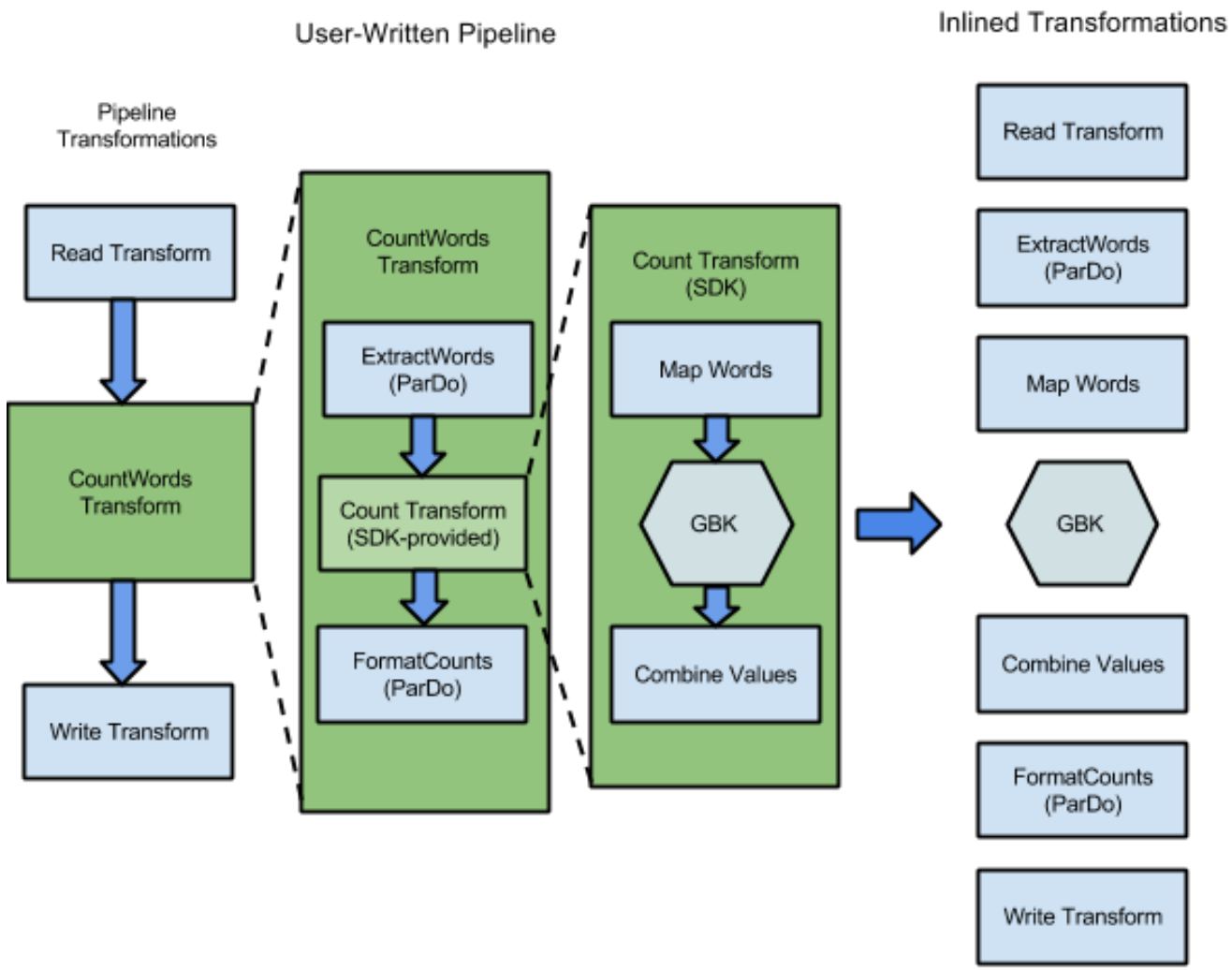


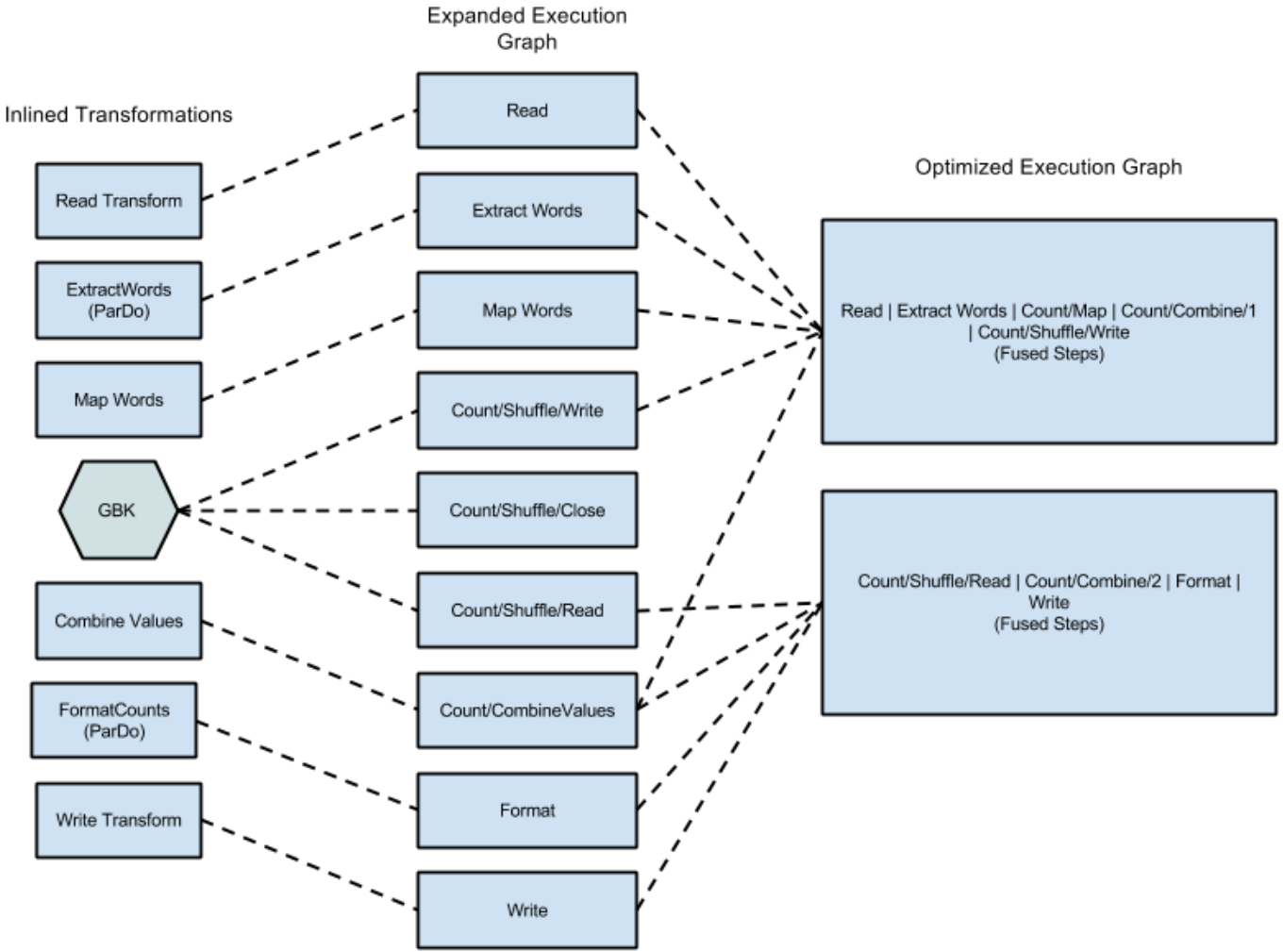
Optimizer: Combiner Lifting



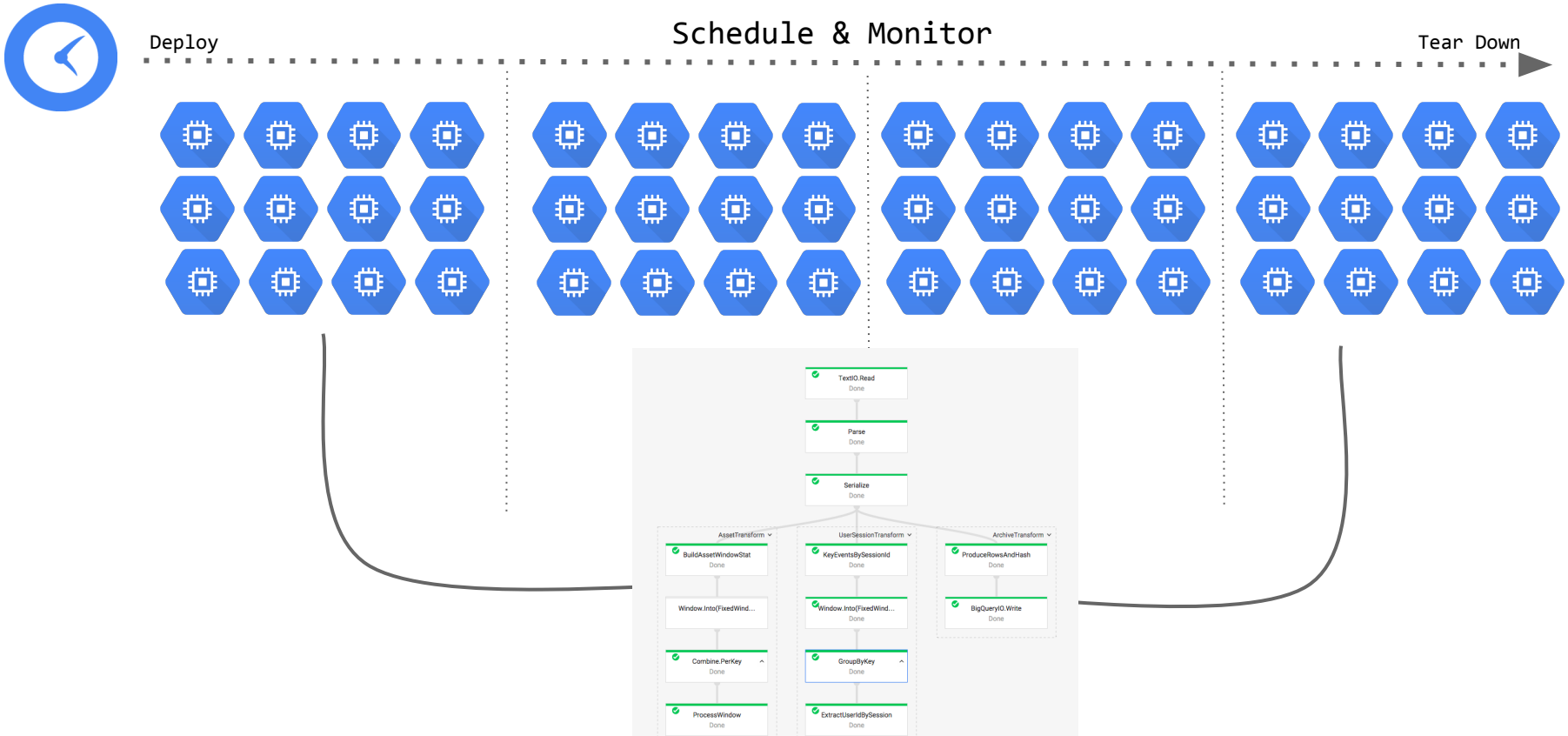




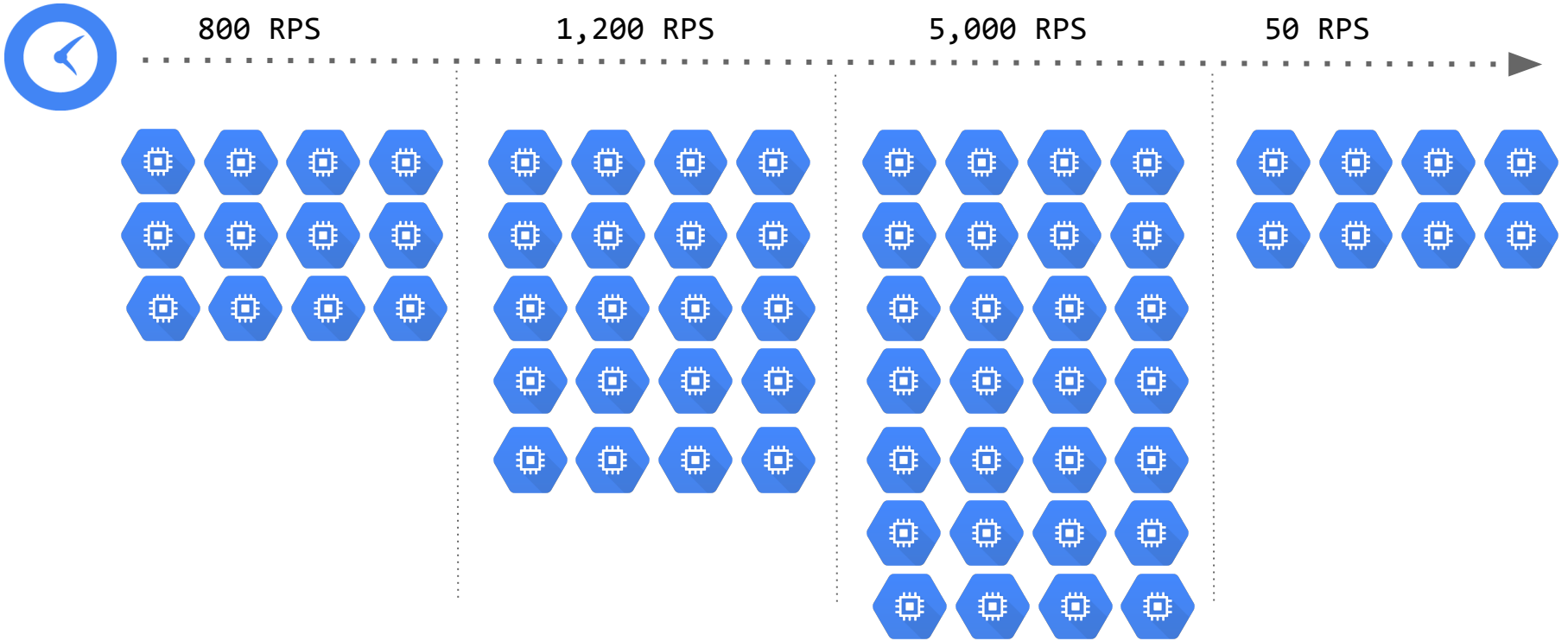

User-Written Pipeline
Inlined Transformations
Pipeline Transformations
Read Transform
CountWords Transform
Write Transform
CountWords Transform
ExtractWords (ParDo)
Count Transform (SDK-provided)
FormatCounts (ParDo)
Count Transform (SDK)
Map Words
GBK
Combine Values
Read Transform
ExtractWords (ParDo)
Map Words
GBK
Combine Values
FormatCounts (ParDo)
Write Transform


Expanded Execution Graph
Inlined Transformations
Read Transform
ExtractWords (ParDo)
Map Words
GBK
Combine Values
FormatCounts (ParDo)
Write Transform
Read
Extract Words
Map Words
Count/Shuffle/Write
Count/Shuffle/Close
Count/Shuffle/Read
Count/CombineValues
Format
Write
Optimized Execution Graph
Read | Extract Words | Count/Map | Count/Combine/1 | Count/Shuffle/Write (Fused Steps)
Count/Shuffle/Read | Count/Combine/2 | Format | Write (Fused Steps)

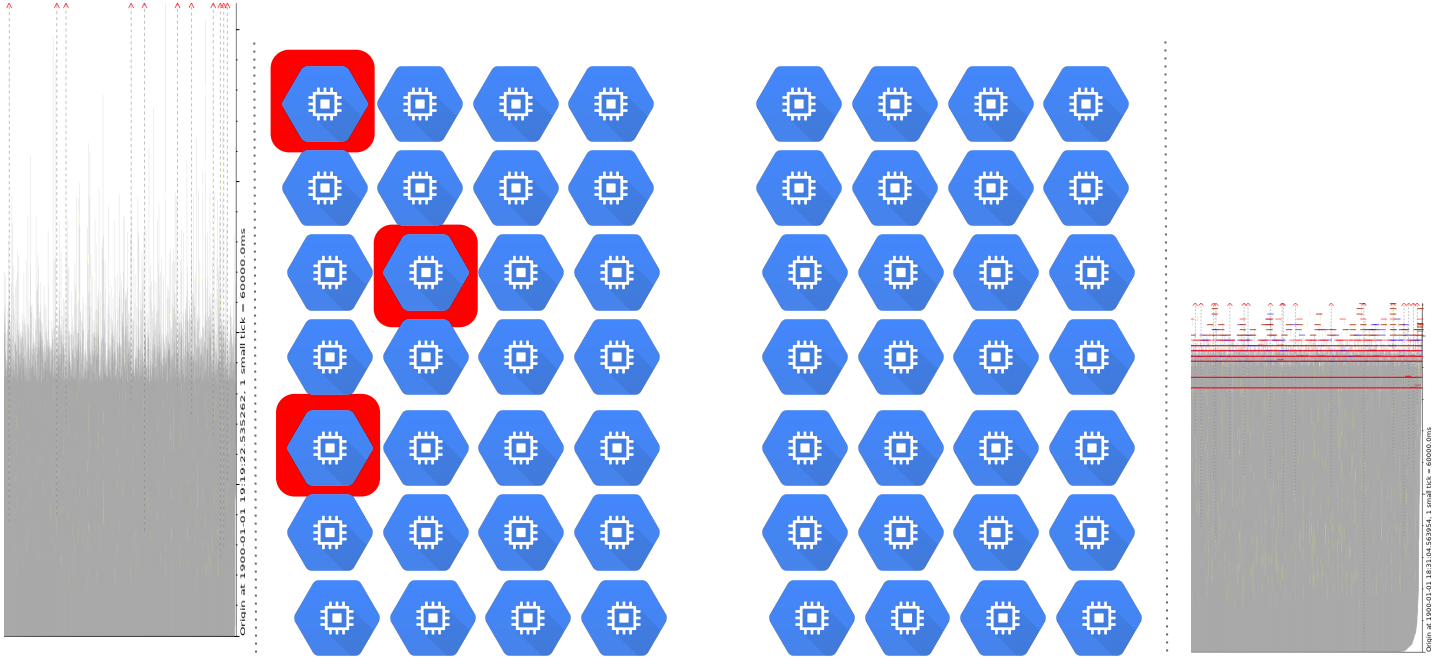
Worker Lifecycle Management



Worker Pool Auto-Scaling



Dynamic Work Rebalancing



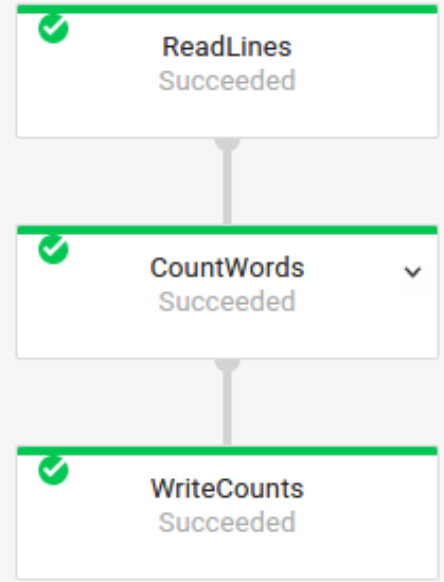
Monitoring

Pipeline Code:

Java

```
// Read the lines of the input text.
p.apply(TextIO.Read.named("ReadLines").from(options.getInput()))
    // Count the words.
    .apply(new CountWords())
    // Write the formatted word counts to output.
    .apply(TextIO.Write.named("WriteCounts")
        .to(options.getOutput())
        .withNumShards(options.getNumShards()));
```

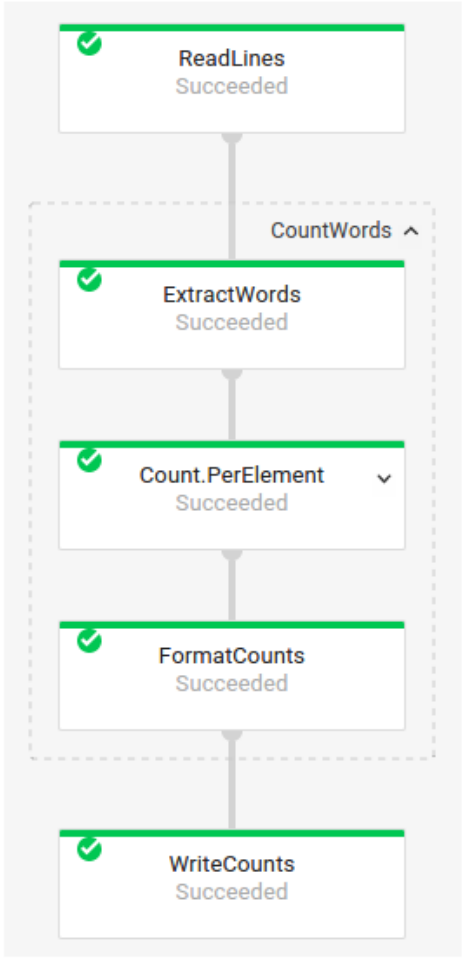
Execution Graph:



Java

```java
// The CountWords Composite Transform
// inside the WordCount pipeline.

public static class CountWords
  extends PTransform<PCollection<String>, PCollection<String>> {

  @Override
  public PCollection<String> apply(PCollection<String> lines) {

    // Convert lines of text into individual words.
    PCollection<String> words = lines.apply(
      ParDo.of(new ExtractWordsFn()));

    // Count the number of times each word occurs.
    PCollection<KV<String, Long>> wordCounts =
      words.apply(Count.<String>perElement());

    // Format each word and count into a printable string.
    PCollection<String> results = wordCounts.apply(
      ParDo.of(new FormatCountsFn()));

    return results;
  }
}
```

Execution Graph:



Fully-managed Service

Pipeline management

- Validation
- Pipeline execution graph optimizations
- Dynamic and adaptive sharding of computation stages
- Monitoring UI

Cloud resource management

- Spin worker VMs
- Set up logging
- Manages exports
- Teardown

Benefits of Dataflow on Google Cloud Platform

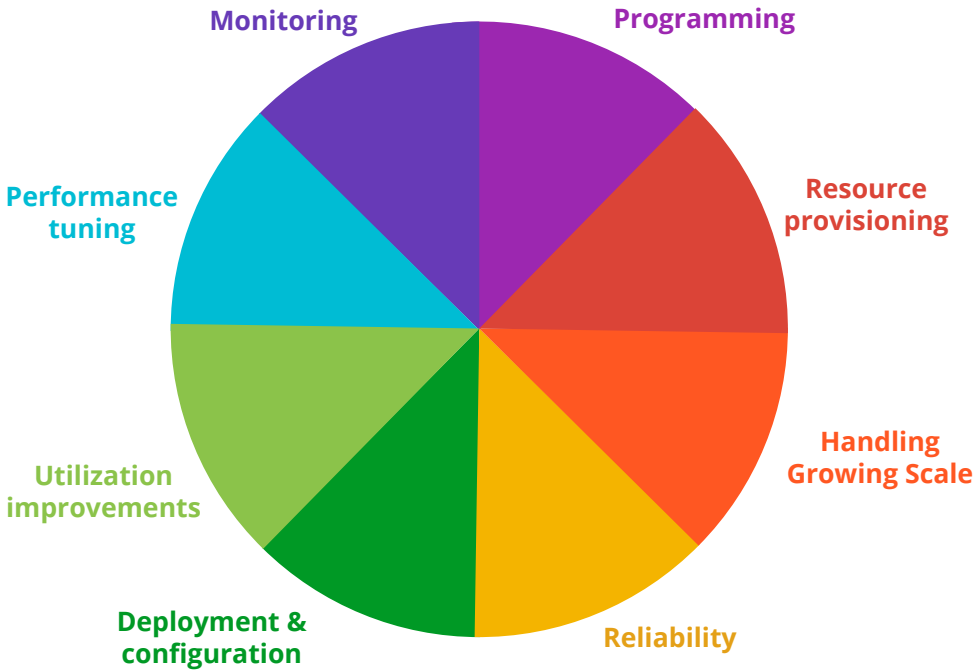
Ease of use

- No performance tuning required
  - Highly scalable, performant out of the box
  - Novel techniques to lower e2e execution time
- Intuitive programming model + Java SDK
  - No dichotomy between batch and streaming processing
- Integrated with GCP (VMs, GCS, BigQuery, Datastore, ...)

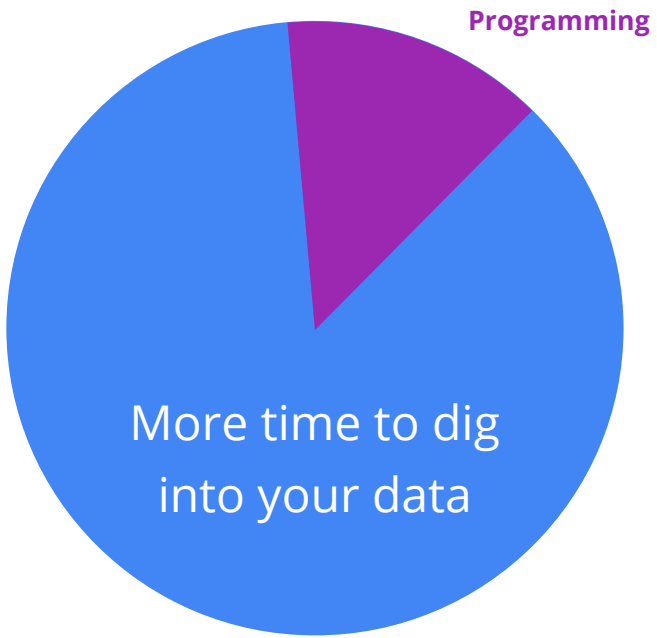
Total Cost of Ownership

- Benefits from GCE's cost model

Optimizing your time: *no-ops, no-knobs, zero-config*

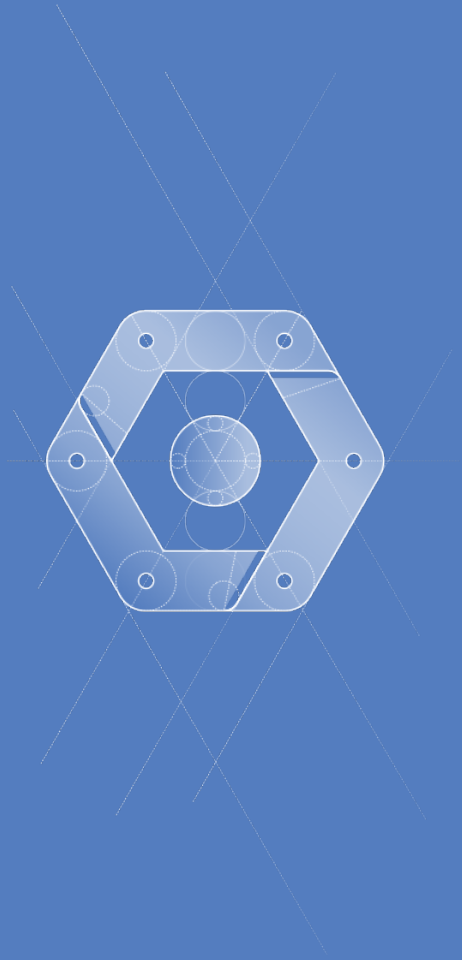


Typical Data Processing



Data Processing with Google Cloud Dataflow

Demo: Counting Words!

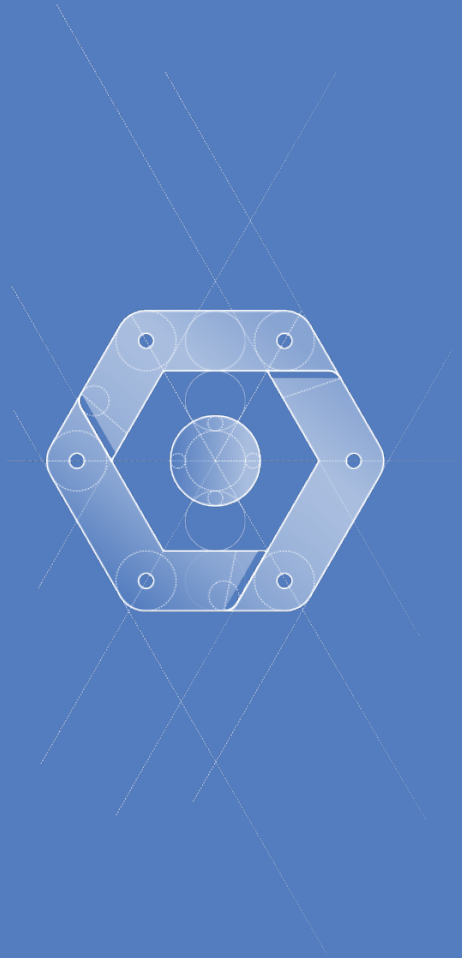


Highlights from the live demo...

- WordCount Code: See the SDK Concepts in action

- Running on the Dataflow Service
  - Monitoring job progress in the Dataflow Monitoring UI
  - Looking at worker logs in Cloud Logging
  - Using the CLI

Questions and Discussion



Getting Started

- cloud.google.com/dataflow/getting-started
- github.com/GoogleCloudPlatform/DataflowJavaSDK
- stackoverflow.com/questions/tagged/google-cloud-dataflow