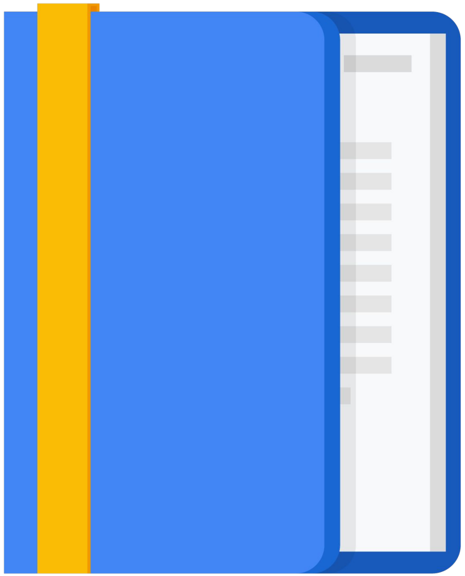
Serverless Data Processing with Dataflow

Agenda

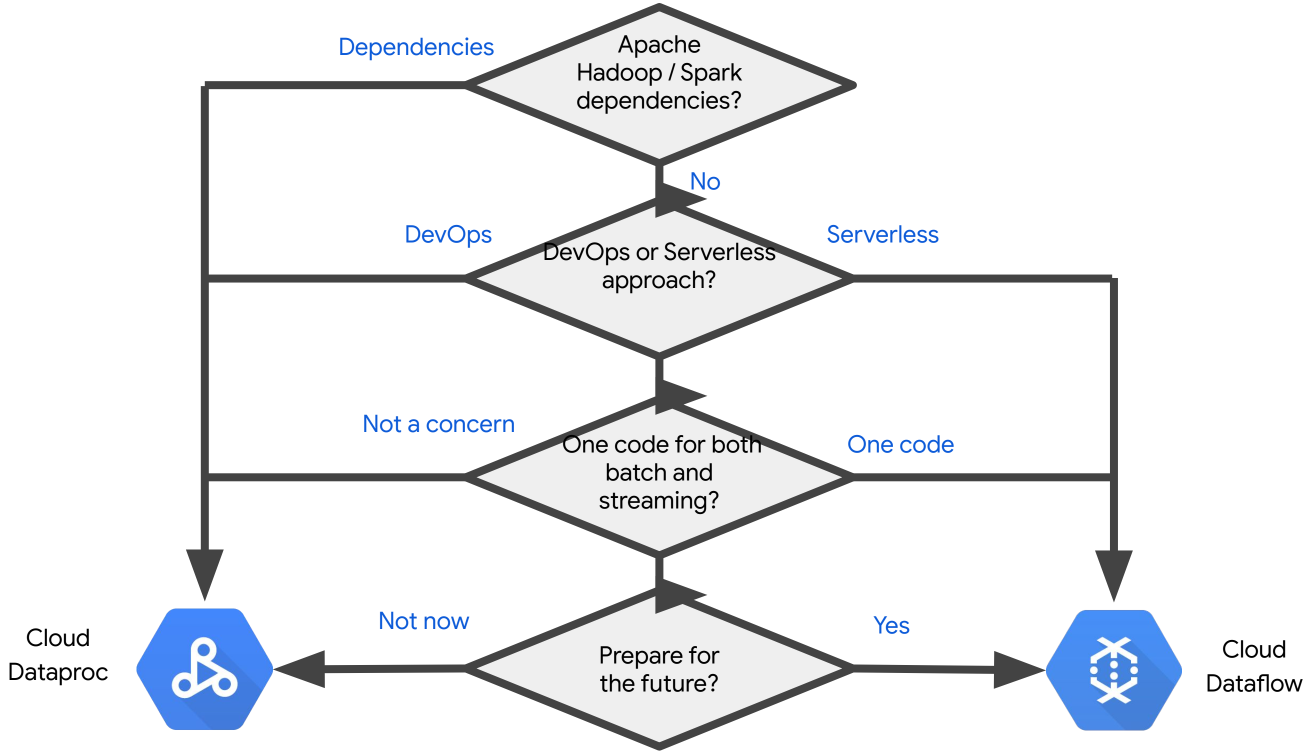
- Cloud Dataflow
- Why customers value Dataflow
- Dataflow Pipelines
- Dataflow Templates
- Dataflow SQL



Google Cloud processing options (1)

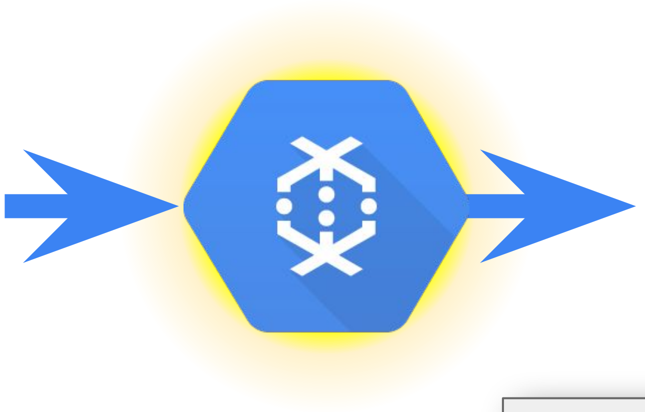
	Cloud Dataflow	Cloud Dataproc
Recommended for:	New data processing pipelines, unified batch and streaming	Existing Hadoop/Spark applications, machine learning/data science ecosystem, large-batch jobs, preemptible VMs
Fully-managed:	Yes	No
Auto-scaling:	Yes, transform-by-transform (adaptive)	Yes, based on cluster utilization (reactive)
Expertise:	Apache Beam	Hadoop, Hive, Pig, Apache Big Data ecosystem, Spark, Flink, Presto, Druid

Choosing between Cloud Dataflow and Cloud Dataproc



Google Cloud

Cloud Dataflow



Cloud
Dataflow

Qualities that Cloud Dataflow contributes to Data Engineering solutions:

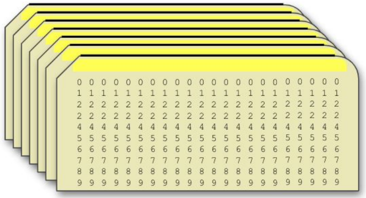
Scalability
Low latency

Batch programming and data processing used to be two very separate and different things

Batch Programming

1940's

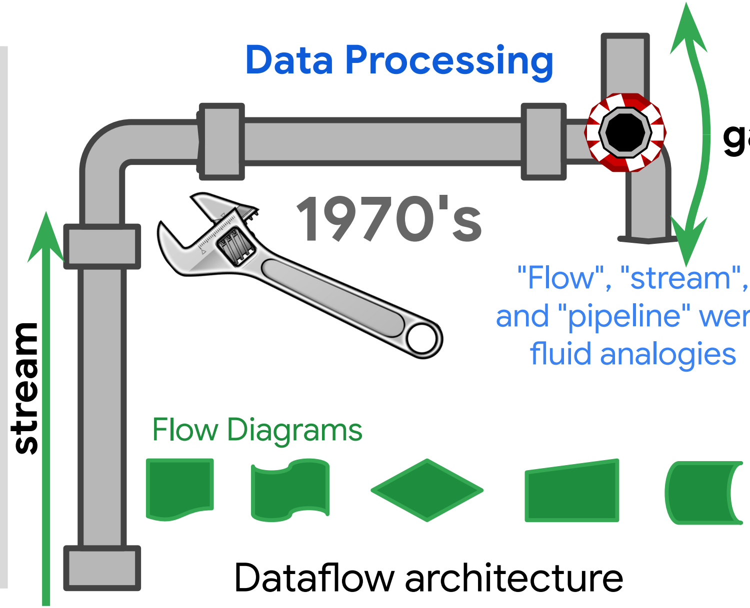
"Batch" originally referred to a batch or box of punch-cards



code	**data**

Von Neumann architecture

Data Processing



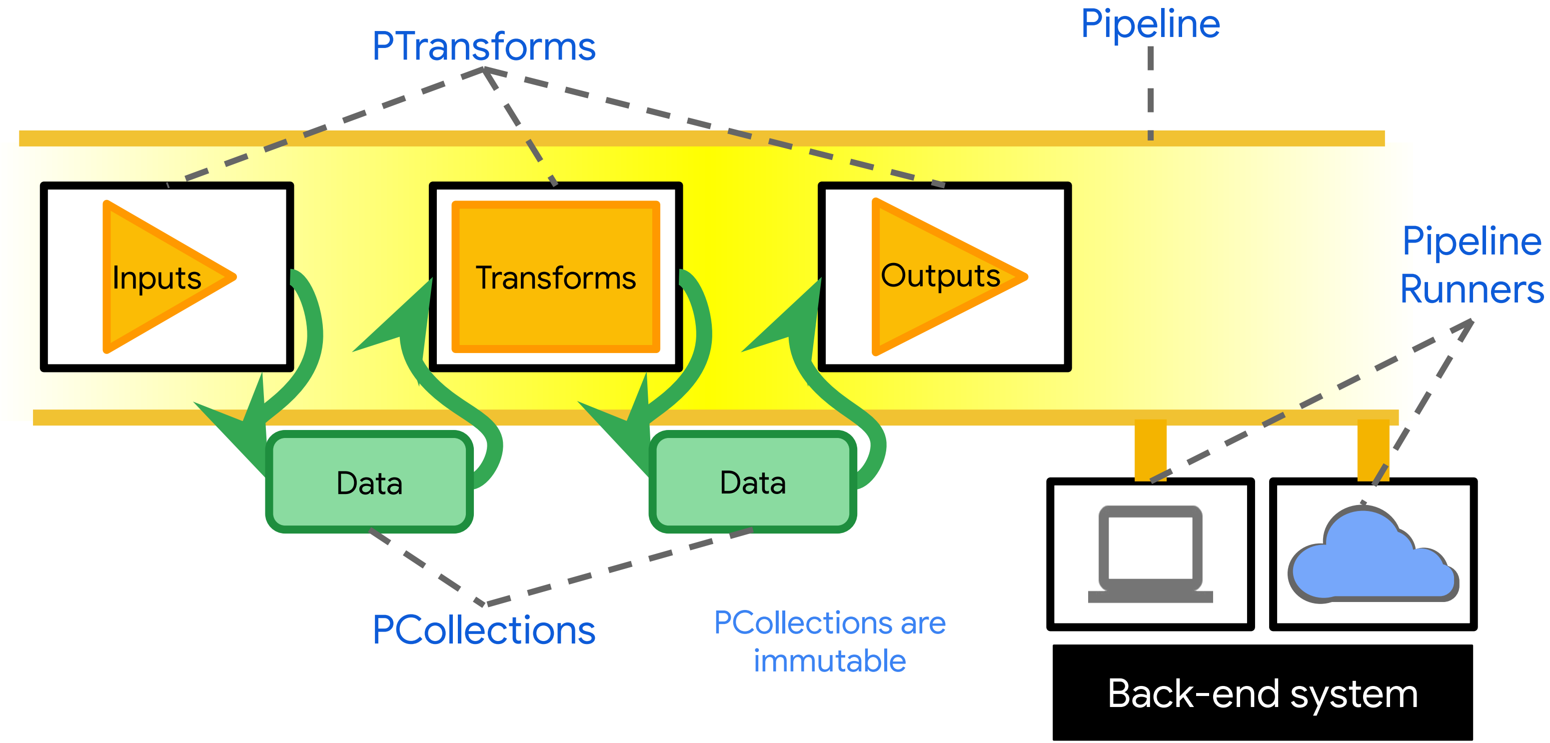
"Flow", "stream", and "pipeline" were fluid analogies

Flow Diagrams

Dataflow architecture

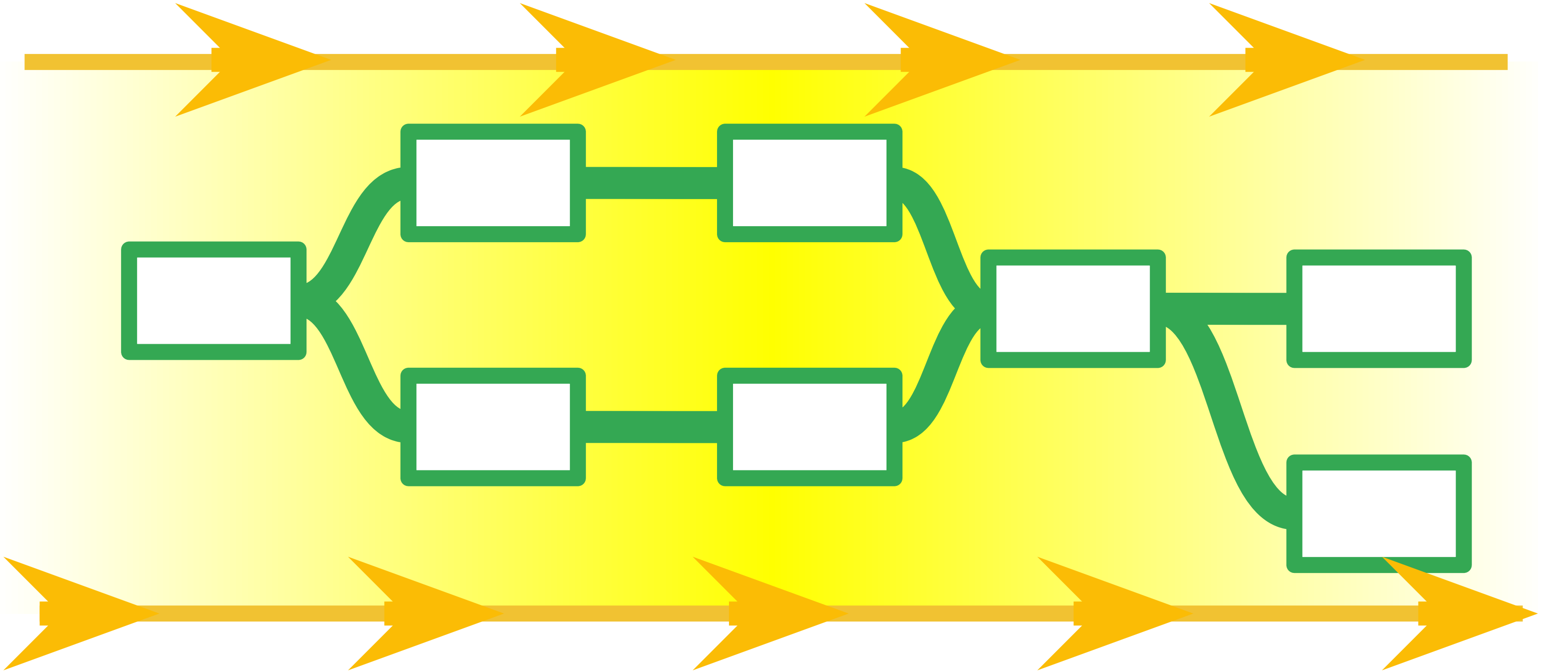
Different tools, different platforms, different concepts, different methods.

Apache BEAM = Batch + strEAM



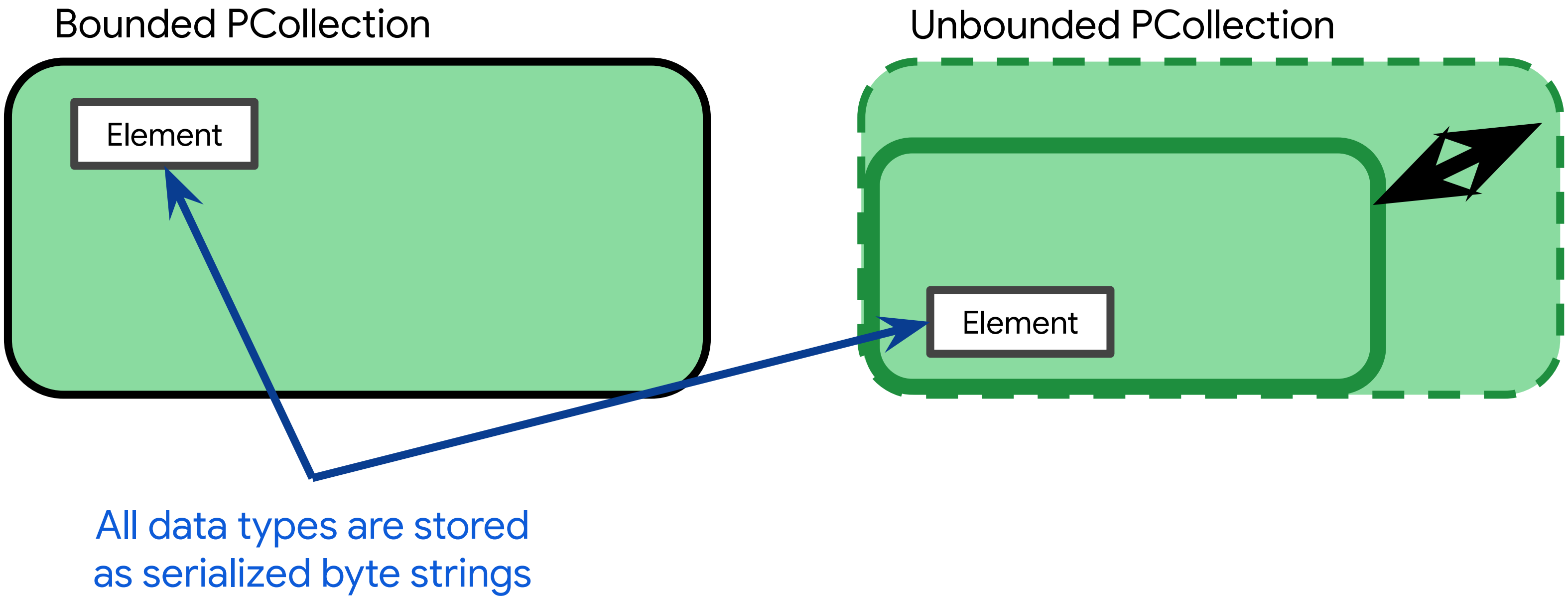
Google Cloud

A Cloud Dataflow pipeline is a directed graph of steps



Google Cloud

A PCollection represents batch or stream data



Note: Bounded means the data has a fixed size not that the PCollection size is limited. A PCollection can be any size and be distributed across many workers.

Agenda

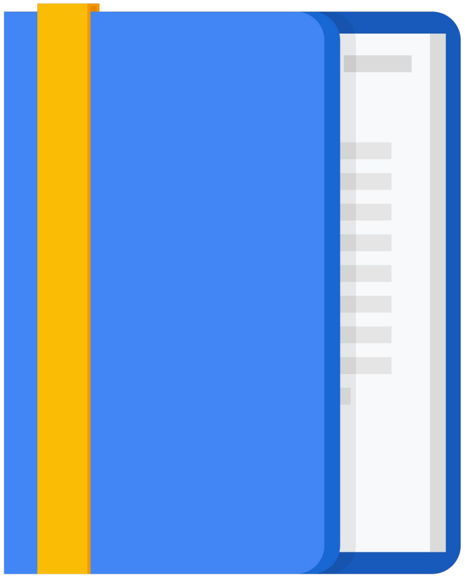
Cloud Dataflow

Why customers value Dataflow

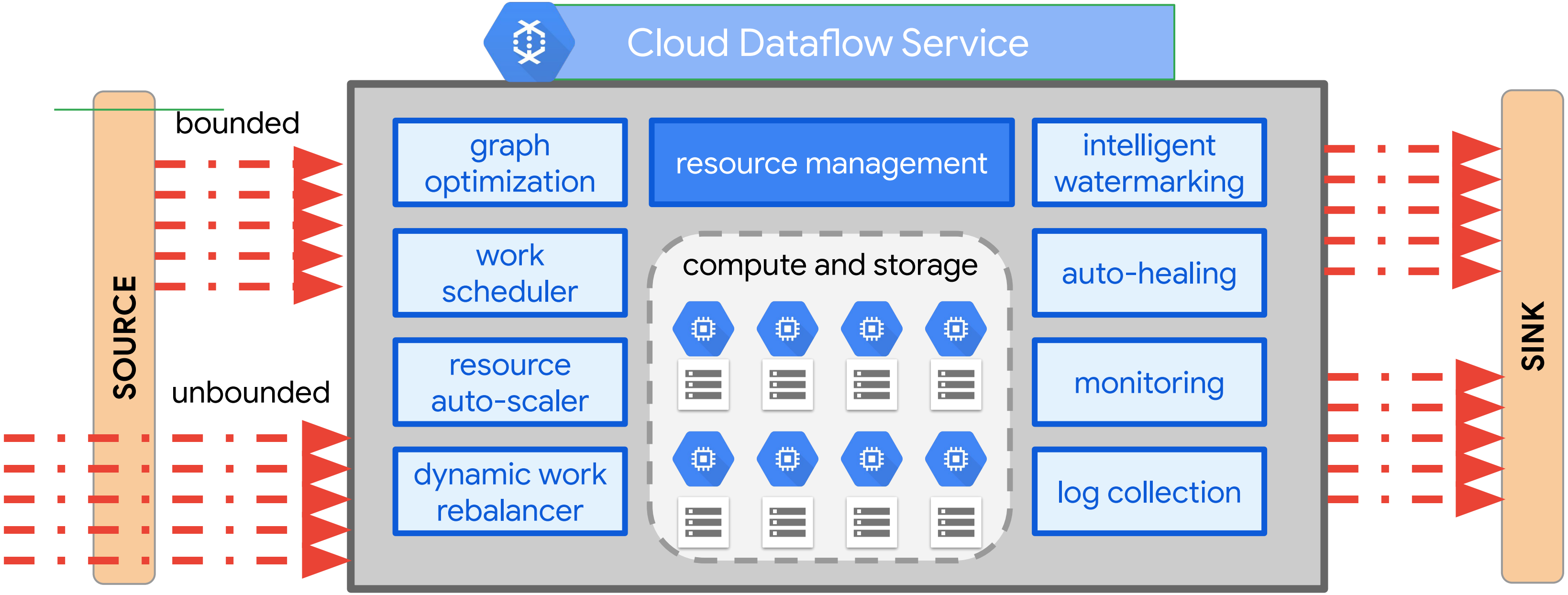
Dataflow Pipelines

Dataflow Templates

Dataflow SQL



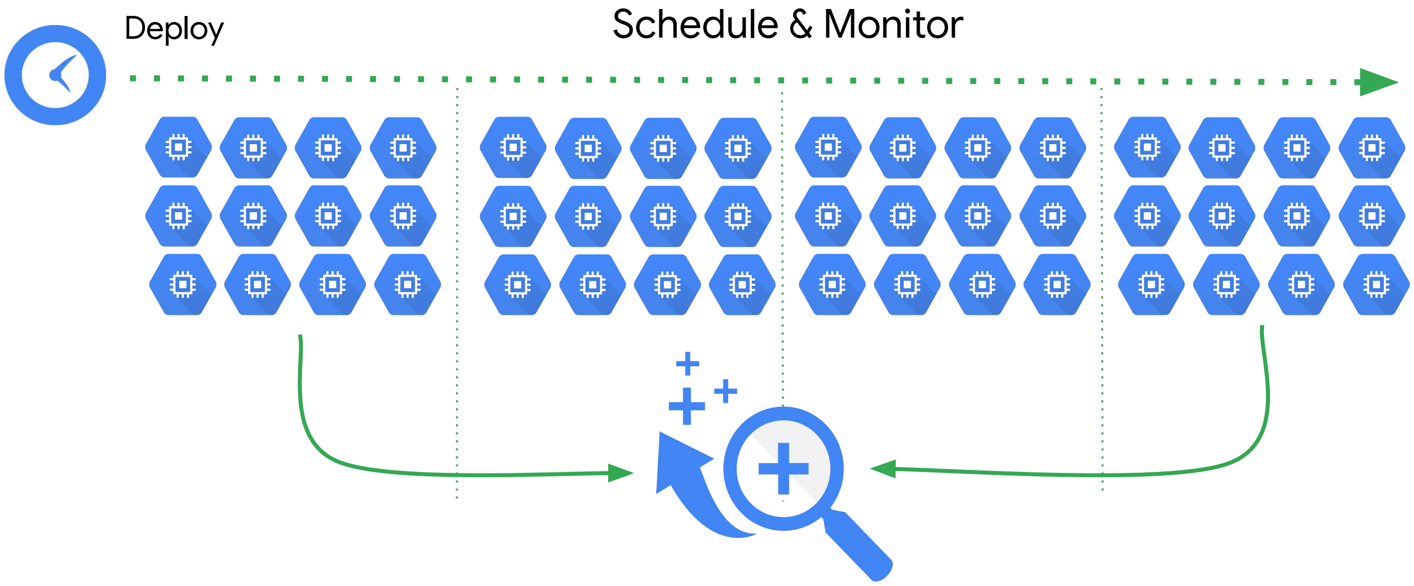
How does Cloud Dataflow work?



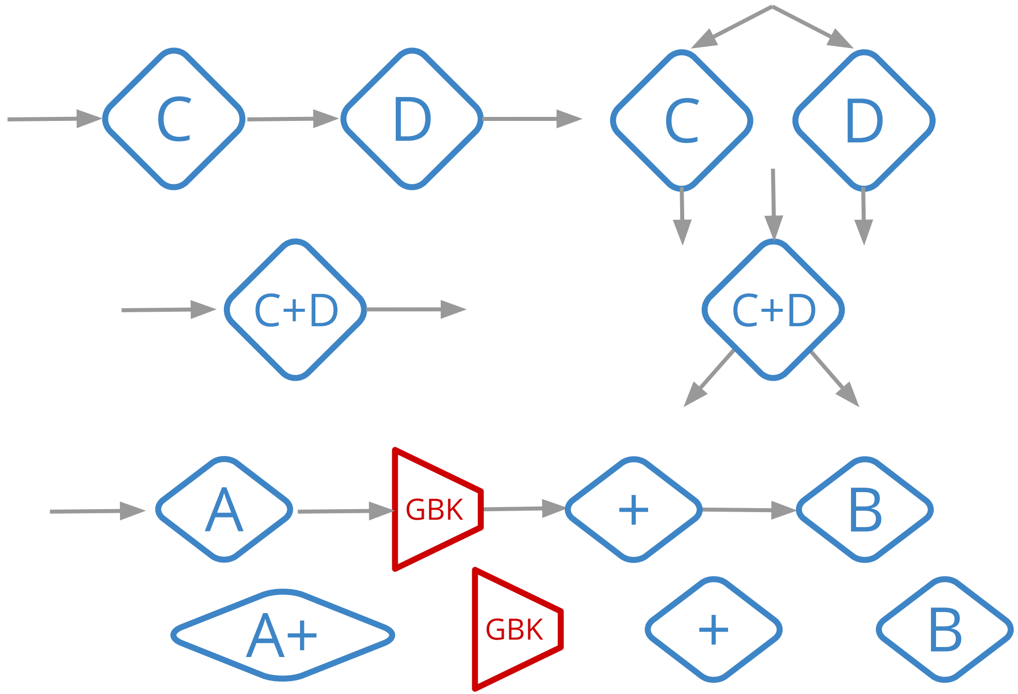
Cloud Dataflow constantly rebalances the work.

Google Cloud

Why customers value Cloud Dataflow: Fully-managed and auto-configured

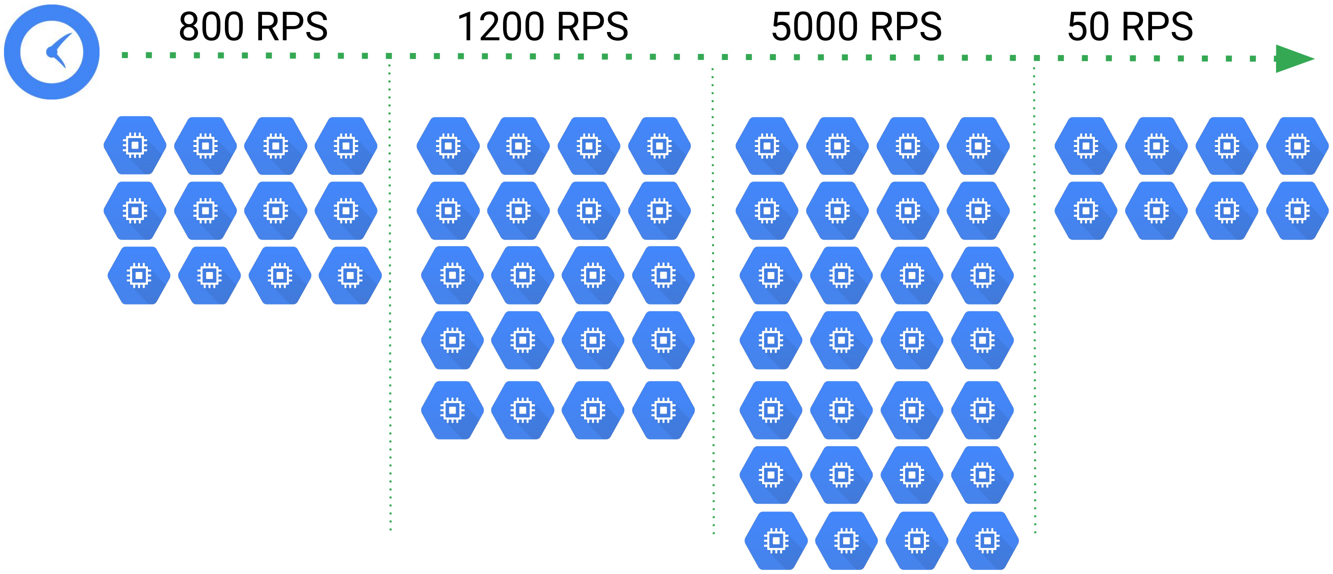


Why customers value Cloud Dataflow: Graph is optimized for best execution path

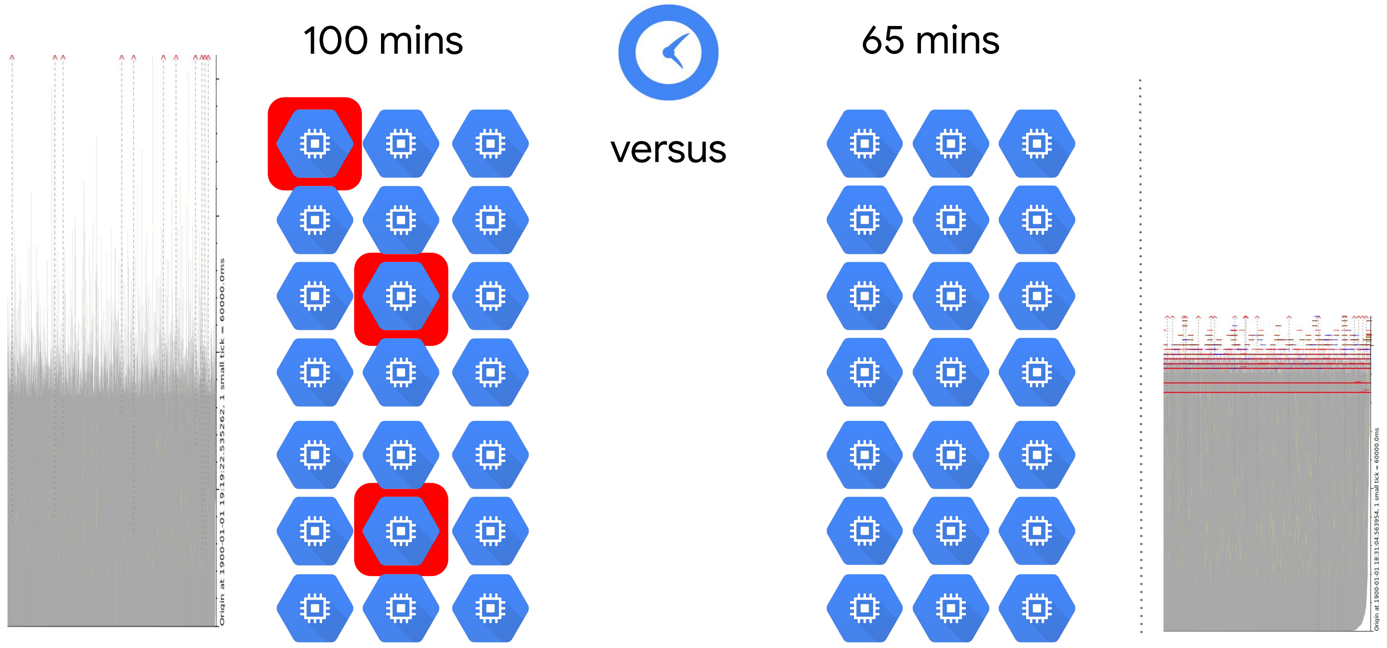


Google Cloud

Why customers value Cloud Dataflow: Autoscaling mid-job



Why customers value Cloud Dataflow: Dynamic work rebalancing mid-job



Why customers value Cloud Dataflow: Strong streaming semantics

- Exactly once aggregations
- Rich time tracking
- Good integration with other GCP services

Agenda

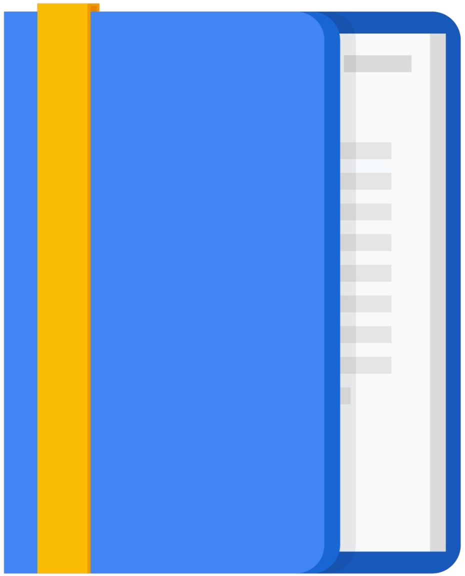
Cloud Dataflow

Why customers value Dataflow

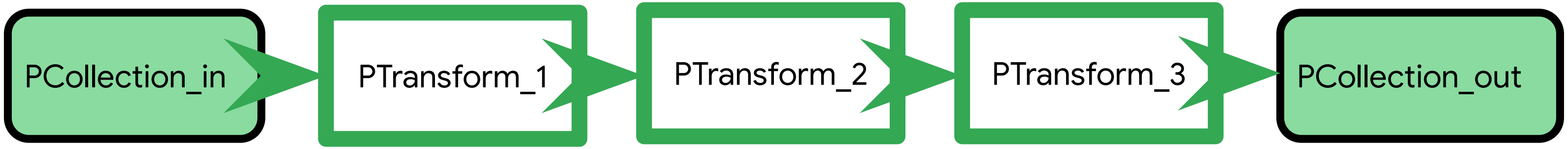
Dataflow Pipelines

Dataflow Templates

Dataflow SQL



How to construct a simple pipeline




```
PCollection_out = (PCollection_in | PTransform_1
                  | PTransform_2
                  | PTransform_3 )
```

Python

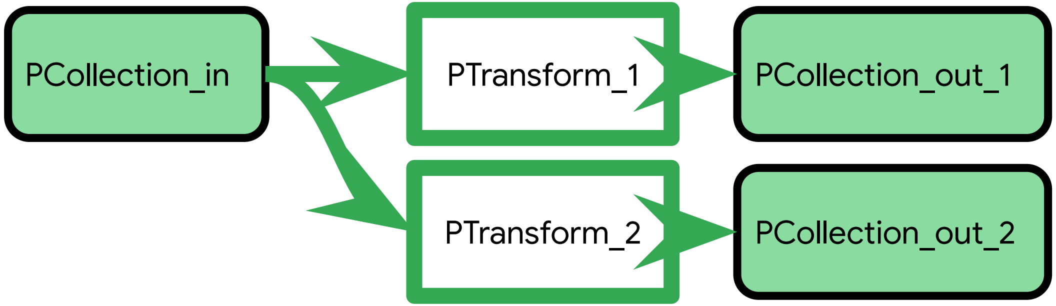
Python overloads the pipe operator

Java

```
PCollection_out = PCollection_in.apply(PTransform_1)
.apply(PTransform_2)
.apply(PTransform_3)
```

Java uses the .apply method

How to construct a branching pipeline



Python

```
PCollection_out_1 = PCollection_in | PTransform_1
PCollection_out_2 = PCollection_in | PTransform_2
```

Java

```
PCollection_out_1 = PCollection_in.apply(PTransform_1)
PCollection_out_2 = PCollection_in.apply(PTransform_2)
```

A Pipeline is a directed graph of steps

Python

```
import apache_beam as beam

if __name__ == '__main__':

   with beam.Pipeline(argv=sys.argv) as p:

        (p
           | beam.io.ReadFromText('gs://...')
           | beam.FlatMap(lambda line:
     count_words(line))
           | beam.io.WriteToText('gs://...')
        )

   # end of with-clause: runs, stops the pipeline
```

Create a pipeline parameterized by command line flags

Read input

Apply transform

Write output

Google Cloud

Run a pipeline on Cloud Dataflow

Python

```
import apache_beam as beam

options = {'project': <project>,
           'runner': 'DataflowRunner',
           'region': <region>,
           'setup_file': <setup.py file>}
pipeline_options =
beam.pipeline.PipelineOptions(flags=[], **options)
pipeline = beam.Pipeline(options = pipeline_options)
```

Where to run

This creates the pipeline

Pipeline Execution using DataflowRunner

Run local

```
python ./grep.py
```

Run on cloud

```
python ./grep.py \
      --project=$PROJECT \
      --job_name=myjob \
      --staging_location=gs://$BUCKET/staging/ \
      --temp_location=gs://$BUCKET/tmp/ \
      --runner=DataflowRunner
```

Designing Pipelines

- **Input and Output**
- PTransforms

Read data from local file system, Cloud Storage, Cloud Pub/Sub, BigQuery, ...

```
with beam.Pipeline(options=pipeline_options) as p:
```

Read from Cloud Storage (returns a string)

```
lines = p | beam.io.ReadFromText("gs://.../input-*.csv.gz")
```

Read from Cloud Pub/Sub (returns a string)

```
lines = p | beam.io.ReadStringsFromPubSub(topic=known_args.input_topic)
```

Read from BigQuery (returns rows)

```
query = "SELECT x, y, z FROM `project.dataset.tablename`"
BQ_source = beam.io.BigQuerySource(query = <query>, use_standard_sql=True)
BQ_data = pipeline | beam.io.Read(BQ_source)
```

Setup

Read

Write to a BigQuery table

Establish reference to BigQuery table

```
from apache_beam.io.gcp.internal.clients import bigquery

table_spec = bigquery.TableReference(
    projectId='clouddataflow-readonly',
    datasetId='samples',
    tableId='weather_stations')
```

Write to BigQuery table

```
p | beam.io.WriteToBigQuery(
    table_spec,
    schema=table_schema,
    write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE,
    create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED)
```

Create a PCollection from in-memory data

Python

```python
city_zip_list = [
    ('Lexington', '40513'),
    ('Nashville', '37027'),
    ('Lexington', '40502'),
    ('Seattle', '98125'),
    ('Mountain View', '94041'),
    ('Seattle', '98133'),
    ('Lexington', '40591'),
    ('Mountain View', '94085'),
]
citycodes = p | 'CreateCityCodes' >> beam.Create(city_zip_list)
```

This is the display name
of the pipeline step

PCollection

Designing Pipelines

- Input and Output
- **PTransforms**

Map and FlatMap

Use Map for 1:1 relationship between input and output

```
'WordLengths' >> beam.Map( lambda word: (word, len(word)) )
```

Map (fn) uses a callable fn to do a one-to-one transformation.

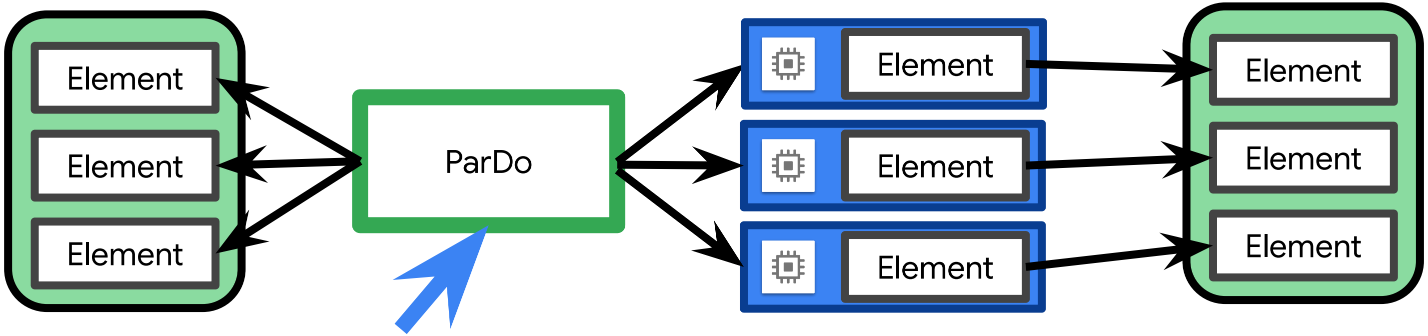
Use FlatMap for non 1:1 relationships, usually with a generator

```
def my_grep(line, term):
   if term in line:
      yield line

'Grep' >> beam.FlatMap( lambda line: my_grep(line, searchTerm) )
```

Generator

FlatMap is similar to Map, but fn returns an iterable of zero or more elements.
The iterables are flattened into one PCollection.

ParDo implements parallel processing



ParDo acts on one item at a time in the PCollection
Multiple instances of class on many machines
Should not contain any state

Uses:

Filtering a data set, choosing which elements to output.
Formatting or type-converting each element in a data set.
Extracting parts of each element in a data set.
Performing computations on each element in a data set.

ParDo requires code passed as a **DoFn** object

Python

```
words = ...

class ComputeWordLengthFn(beam.DoFn):
  def process(self, element):
    return [len(element)]

word_lengths = words | beam.ParDo(ComputeWordLengthFn())
```

DoFN

ParDo

The input is a PCollection of strings.

The DoFn to perform on each element in the input PCollection.

The output is a PCollection of integers.

Apply a ParDo to the PCollection "words" to compute lengths for each word.

ParDo method can emit multiple variables

```
results = (words | beam.ParDo(ProcessWords(), cutoff_length=2, marker='x')
   .with_outputs('above_cutoff_lengths', 'marked strings',
main='below_cutoff_strings'))

below   = results.below_cutoff_strings
above   = results.above_cutoff_lengths
marked = results['marked strings']
```

Lab

A Simple Dataflow Pipeline (Python/Java)

Objectives

- Open Dataflow project
- Pipeline filtering
- Execute the pipeline locally and on the cloud

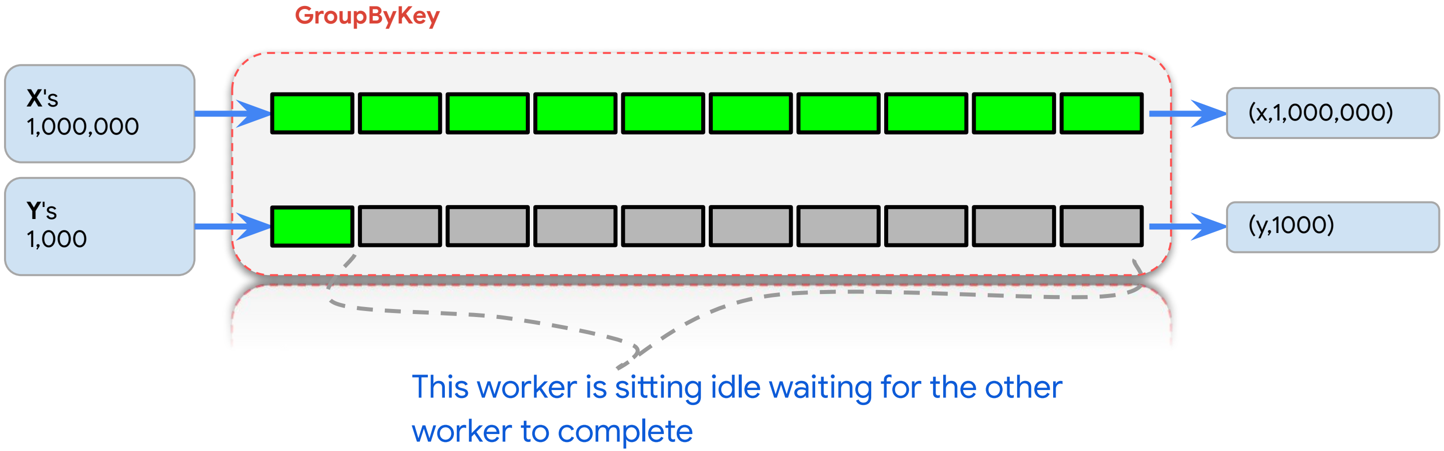
GroupByKey explicitly shuffles key-values pairs

```
cityAndZipcodes = p | beam.Map(lambda fields : (fields[0], fields[1]))

grouped = cityAndZipCodes | beam.GroupByKey()
```

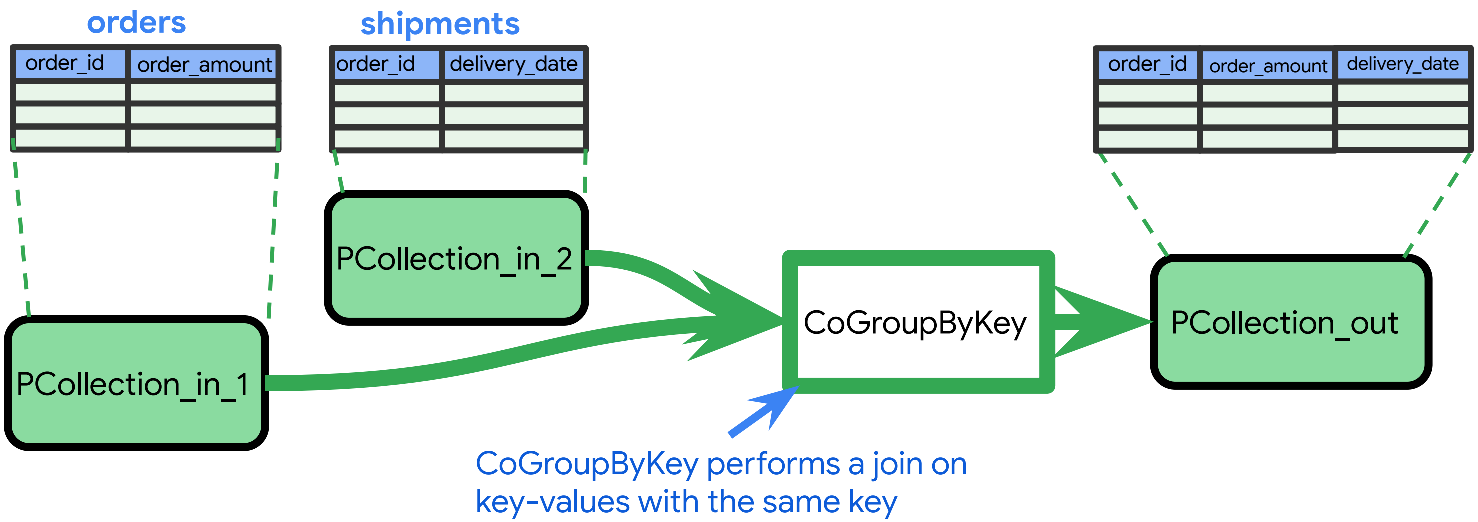
```
Lexington, 40513
Nashville, 37027
Lexington, 40502
Seattle, 98125
Mountain View, 94041
Seattle, 98133
Lexington, 40591
Mountain View, 94085
```

```
Lexington, [40513, 40502, 40592]
Nashville, [37027]
Seattle, [98125, 98133]
Mountain View, [94041, 94085]
```

Data skew makes grouping less efficient at scale



CoGroupByKey joins two or more key-value pairs




```
results = ({'orders': orders, 'shipments': shipments}
           | beam.CoGroupByKey())
```

Google Cloud

Combine (reduce) a PCollection

Applied to a PCollection of values

```
totalAmount = salesAmounts | CombineGlobally(sum)
```

Applied to a grouped Key-Value pair

```
totalSalesPerPerson = salesRecords | CombinePerKey(sum)
```

Each element of salesRecords is a tuple:
(salesPerson, salesAmount)

Pre-built combine functions for many common numeric combination operations such as sum, mean, min, and max

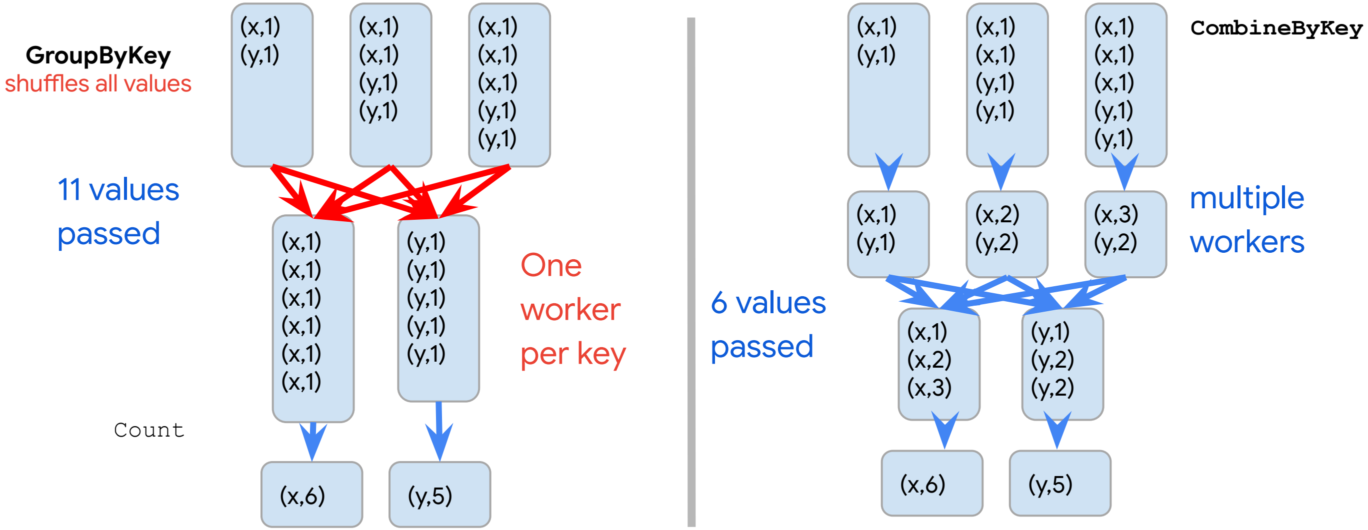
CombineFn works by overriding existing operations

You must provide four operations by overriding the corresponding methods

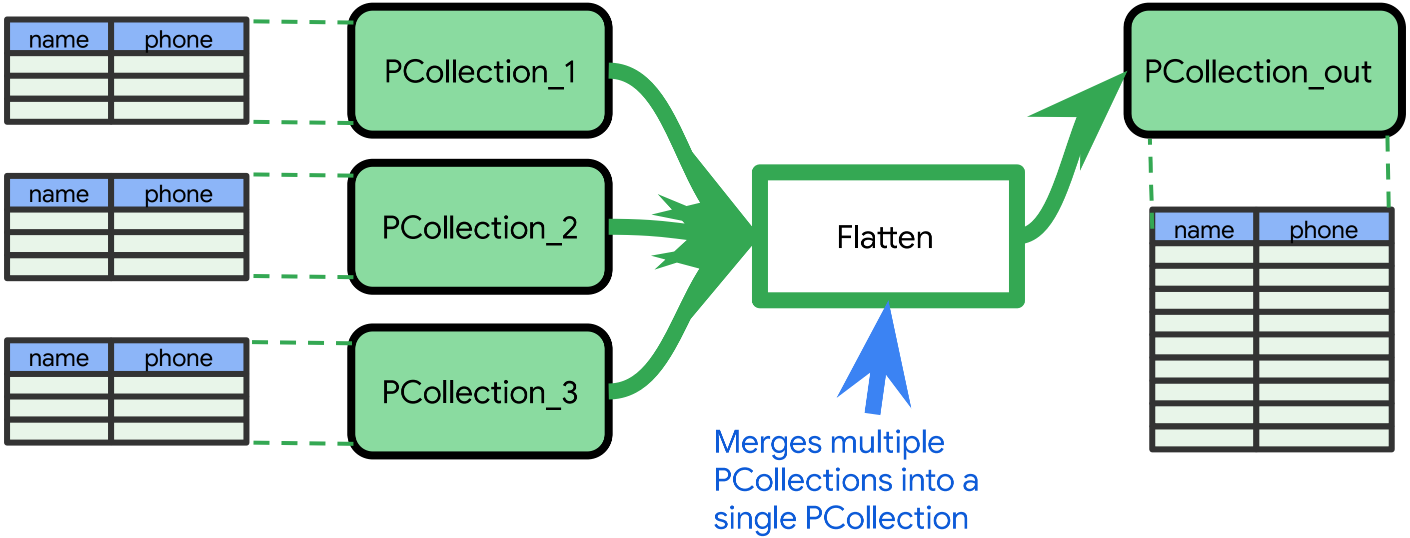
```
class AverageFn(beam.CombineFn):

  def create_accumulator(self):
    return (0.0, 0)

  def add_input(self, sum_count, input):
    (sum, count) = sum_count
    return sum + input, count + 1

  def merge_accumulators(self, accumulators):
    sums, counts = zip(*accumulators)
    return sum(sums), sum(counts)

  def extract_output(self, sum_count):
    (sum, count) = sum_count
    return sum / count if count else float('NaN')
```

```
pc = ...
average = pc | beam.CombineGlobally(AverageFn())
```

Combine is more efficient than GroupByKey

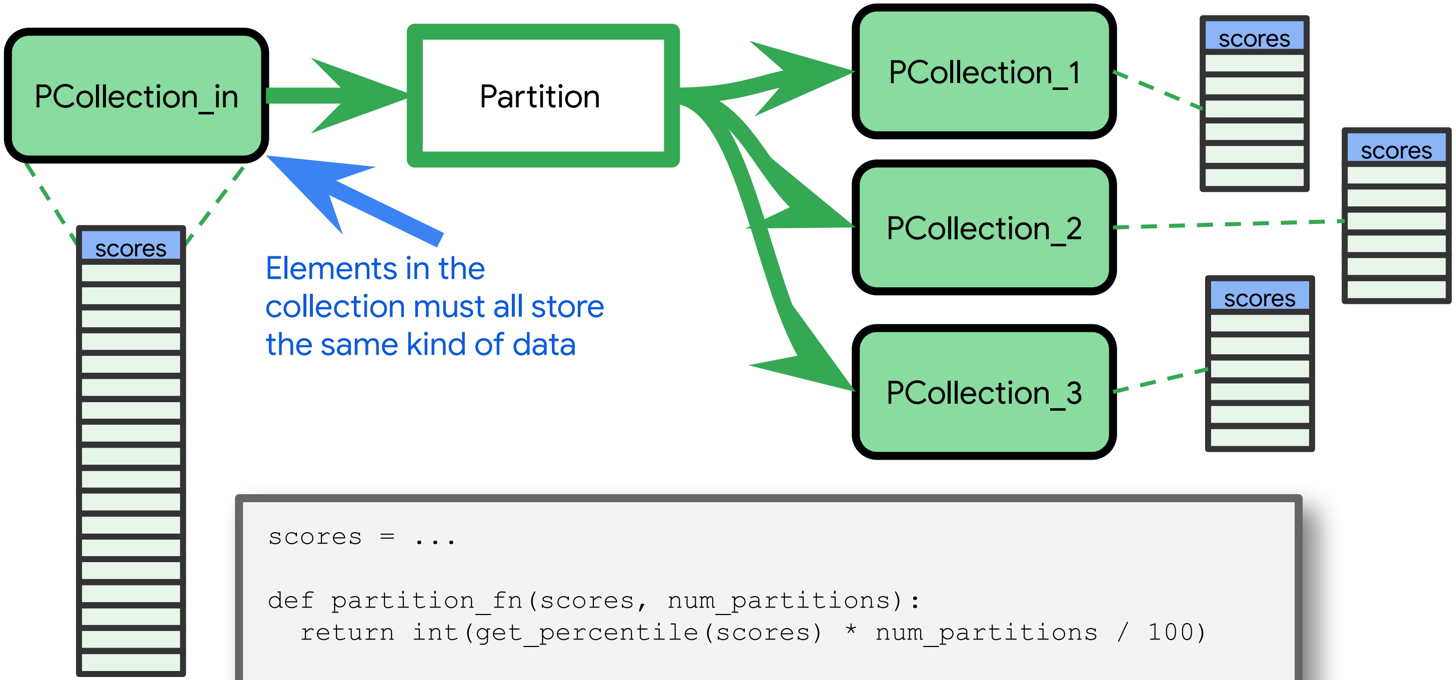


Flatten merges identical PCollections




```
merged = ((pcoll1, pcoll2, pcoll3) | beam.Flatten())
```

Partition splits PCollections into smaller PCollections




```
scores = ...

def partition_fn(scores, num_partitions):
  return int(get_percentile(scores) * num_partitions / 100)

by_decile = scores | beam.Partition(partition_fn, 10)
```

Google Cloud

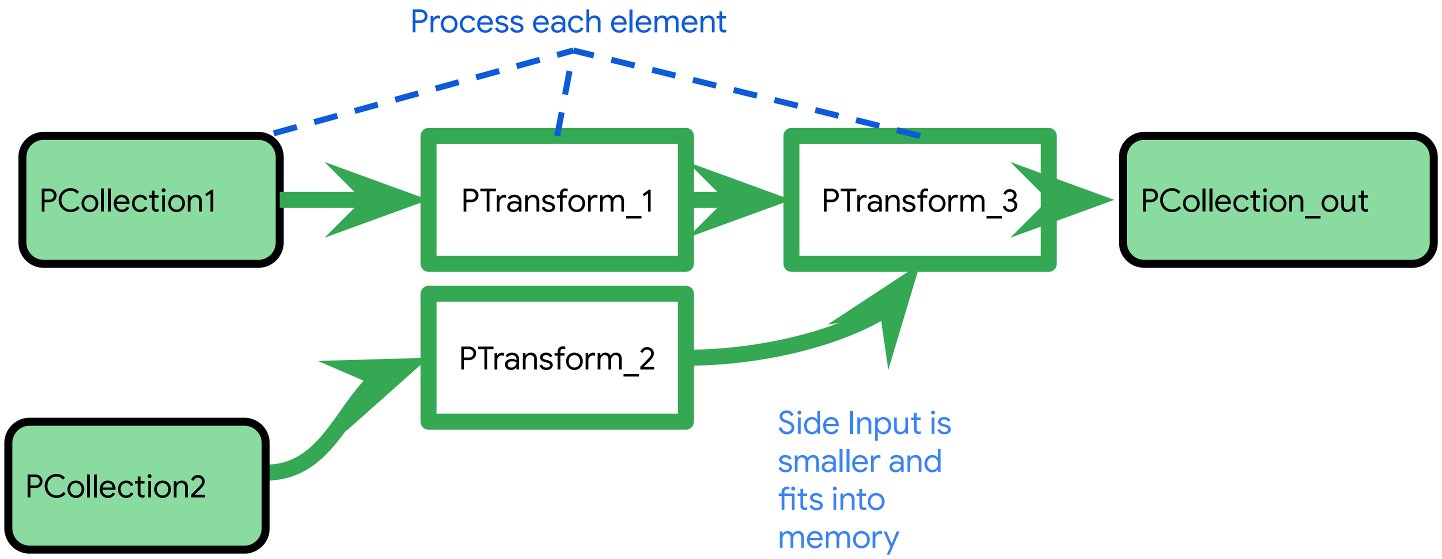
Lab

MapReduce in Dataflow (Python/Java)

Objectives

- Identify Map and Reduce operations
- Execute the pipeline
- Use command line parameters

Use side inputs to inject additional runtime data



How side inputs work

```
words = …

def filter_using_length(word, lower_bound, upper_bound=float('inf')):
  if lower_bound <= len(word) <= upper_bound:
    yield word

small_words = words | 'small' >> beam.FlatMap(filter_using_length, 0, 3)

avg_word_len = (words
                | beam.Map(len)
                | beam.CombineGlobally(beam.combiners.MeanCombineFn()))


larger_than_average = (words | 'large' >> beam.FlatMap(
    filter_using_length,
    lower_bound=pvalue.AsSingleton(avg_word_len)))
```

Side input

Lab

Side Inputs (Python/Java)

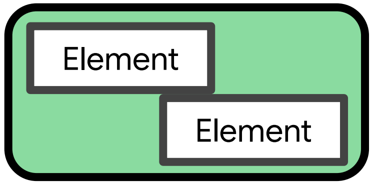
Objectives

- Try out a BigQuery query
- Explore the pipeline code
- Execute the pipeline

Processing Time-series data using Windowing

Every PCollection is processed within a Window

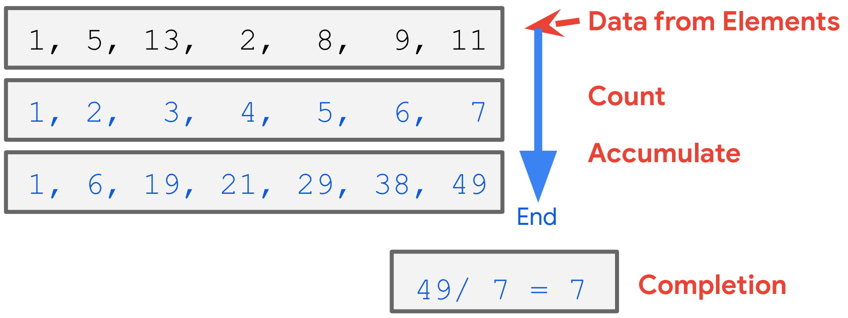
Bounded PCollection



1 The default window is called the global window, it starts when the data is input and ends when the last element in the collection is processed.

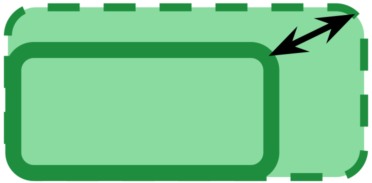
2 In Bounded PCollections, commonly the Elements are all marked as occurring at the same time. (Example: TextIO does this.) So the global window basically ignores the timing information.

3



The global window is not very useful for an unbounded PCollection

Unbounded PCollection



1

The timing associated with the elements in an Unbounded PCollection is usually important to processing the data.

3

The discussion about Unbounded PCollections and Windows will be continued in the course on Processing Streaming Data.

2

An Unbounded PCollection has no defined end or last element. So it can never perform the completion step.

This is particularly important for **GroupByKey** and **Combine**, which perform the shuffle after 'end'.

Setting a single global window for a PCollection.

Single global window

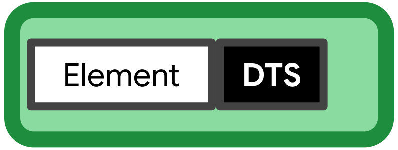
Python

```
from apache_beam import window
session_windowed_items = (
    items | 'window' >> beam.WindowInto(window.GlobalWindows()))
```

This is the default.

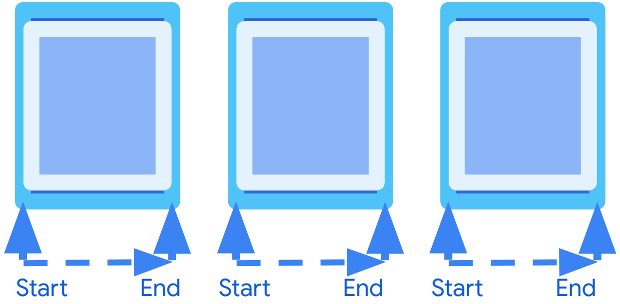
This code illustrates how you could explicitly set it.

Time-based Windows can be useful for processing time-series data



1

You may have to prepare the date-timestamp. In this example, the dts of the data (log writing time) becomes the element time. Now the elements have different times from one another.



2

Using time based windowing the data is processed in groups.

In the example, each group gets its own average.

3

There are different kinds of windowing.

Shown is "Fixed" There is also "Sliding" and "Session".

Using Windowing with Batch (group by time)

Python

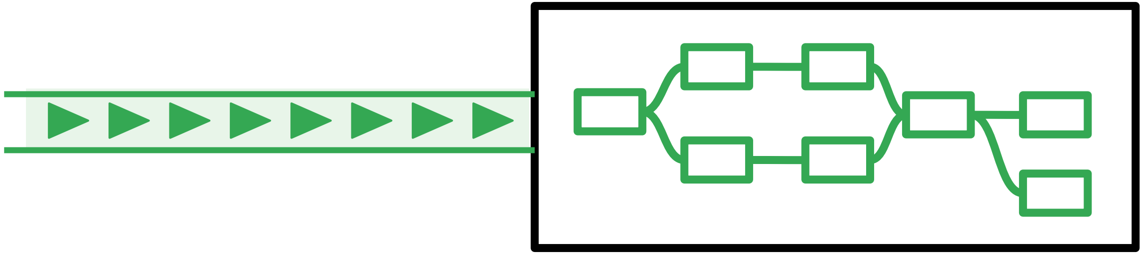
```
lines = p | 'Create' >> beam.io.ReadFromText('access.log')
windowed_counts = (
    lines
    | 'Timestamp' >> beam.Map(lambda x: beam.window.TimestampedValue(x, extract_timestamp(x)))
    | 'Window' >> beam.WindowInto(beam.window.SlidingWindows(60, 30))
    | 'Count' >> (beam.CombineGlobally(beam.combiners.CountCombineFn()).without_defaults())
)
windowed_counts =  windowed_counts | beam.ParDo(PrintWindowFn())
```

access.log (example)

```
131.108.5.17 - - [29/Apr/2019:04:53:15 -0800] "GET /view HTTP/1.1" 200 7352
131.108.5.17 - - [29/Apr/2019:05:21:35 -0800] "GET /view HTTP/1.1" 200 5253
```

Date Time Stamp

Streaming data processing with Cloud Dataflow



Discussion of streaming continues in the Streaming Data Processing course.

Agenda

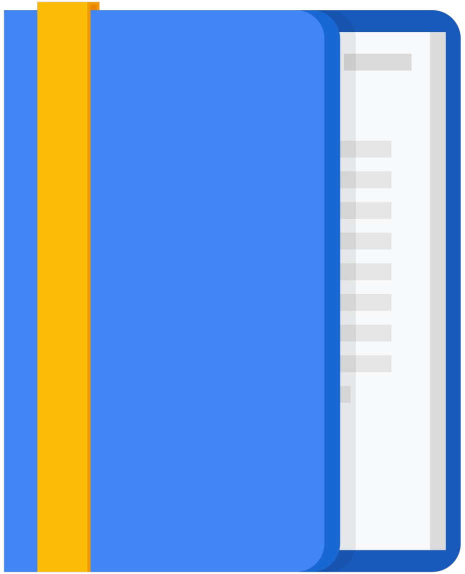
Cloud Dataflow

Why customers value Dataflow

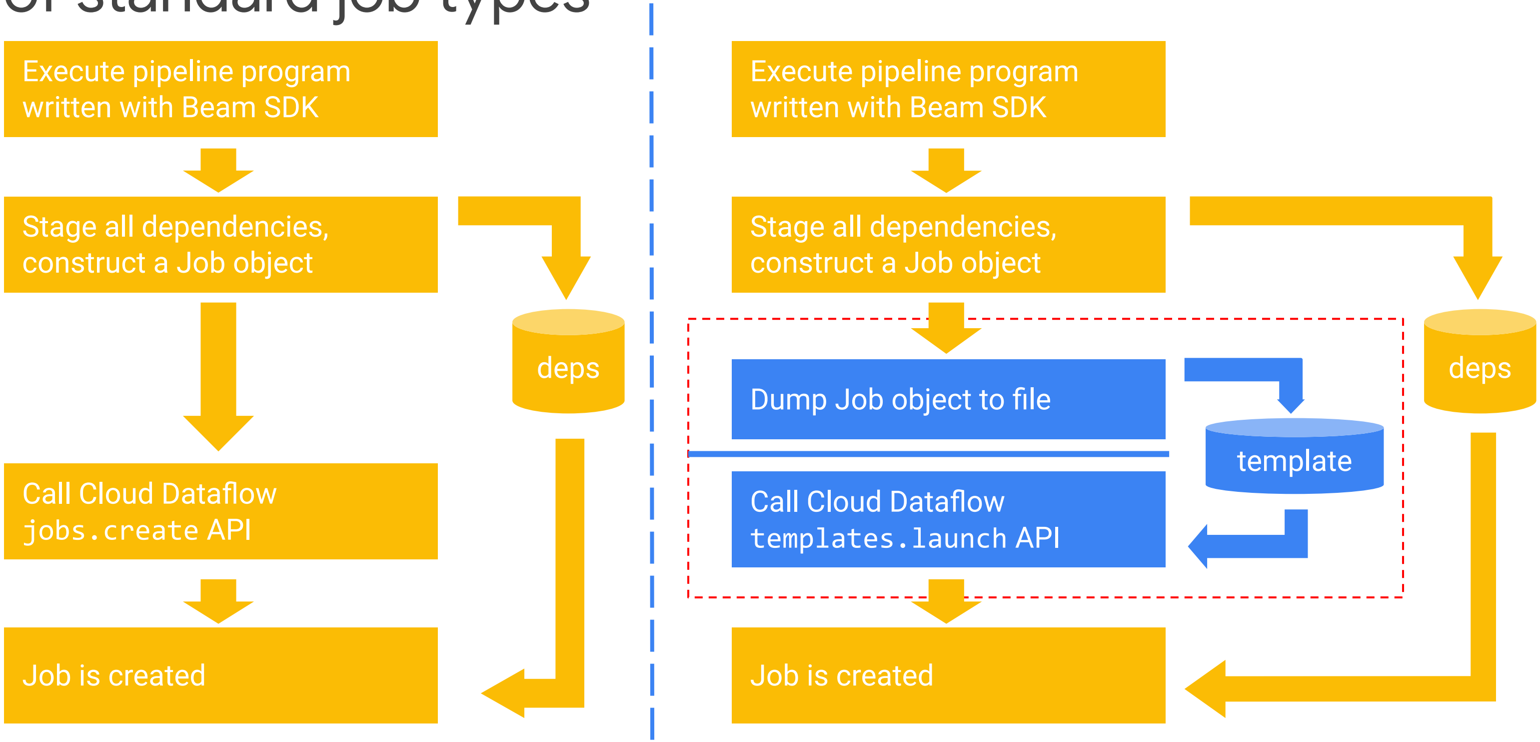
Dataflow Pipelines

Dataflow Templates

Dataflow SQL

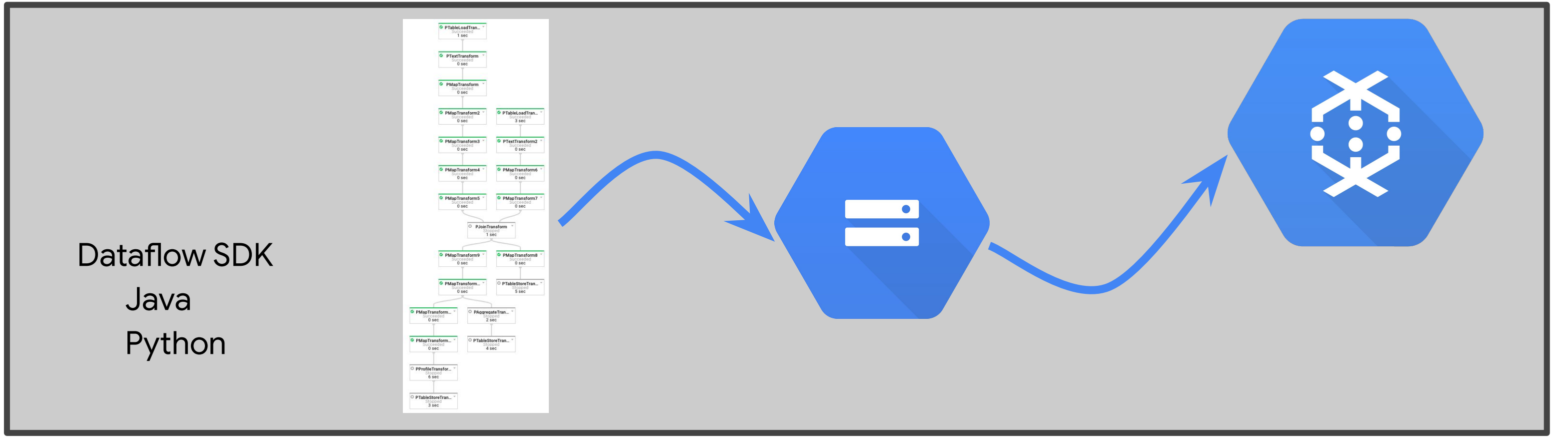


Cloud Dataflow templates enable the rapid deployment of standard job types



Traditional workflow all happens in one environment

Development environment



Developer executes pipeline on Dataflow

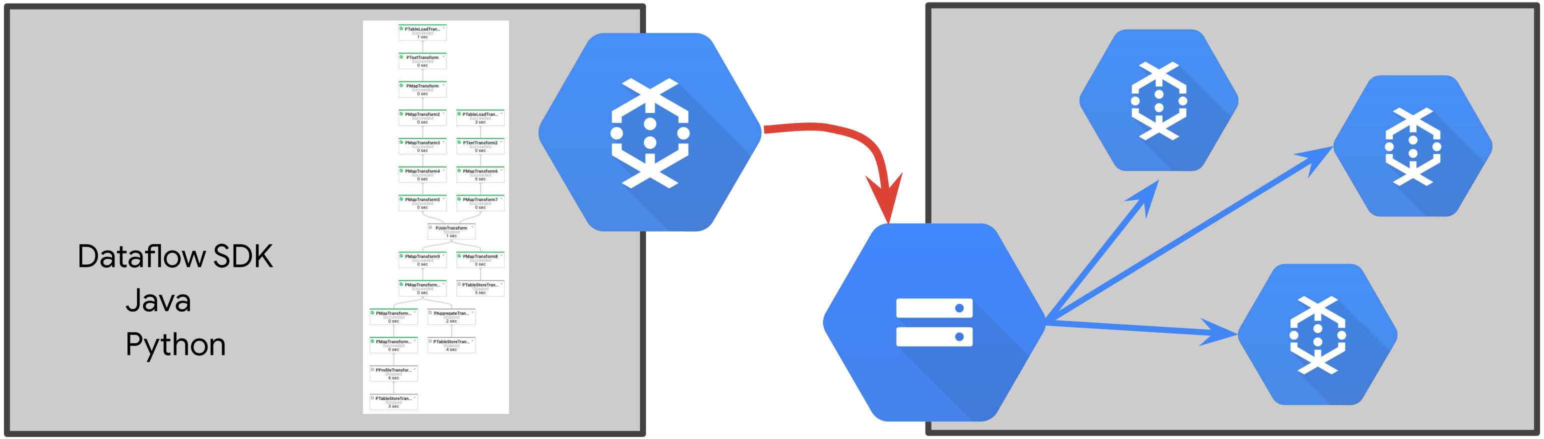
SDK stages files in Cloud Storage

Developer or User submits source code to run Dataflow jobs

Template workflow supports non-developer users

Development environment

Production environment



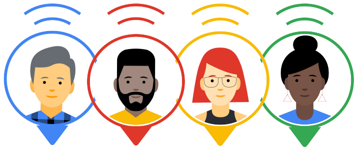
Developer creates pipeline in the development environment

Dataflow stores template in cloud storage

Users submit templates to run jobs

Get started with Google-provided templates

Pre-written Cloud Dataflow pipelines for common data tasks that can be triggered with a single command or UI form.



Target users

- App developers
- DB admins
- Analysts
- Data scientists
- Data engineers

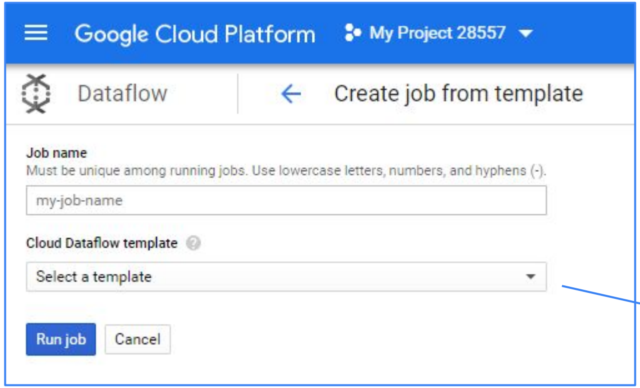
Exposure

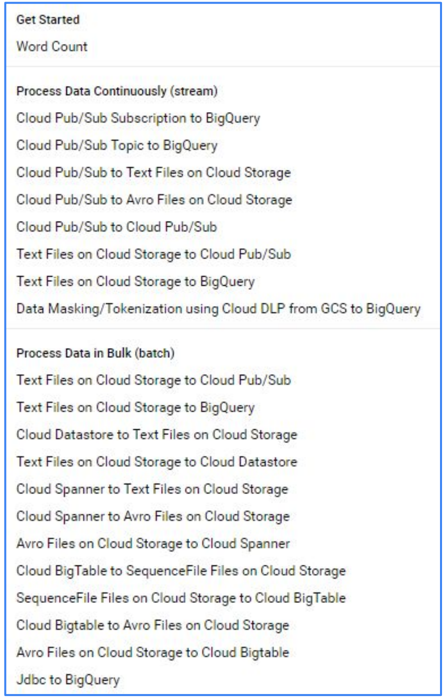
- Through Google-provided Cloud Dataflow templates
- Embedded in other GCP products calling templates API

Data Fusion

- Branded Google product
- UI pipeline builder
- Scheduler/orchestrator

Execute templates with the GCP Console, `gcloud` command-line tool, or the REST API






```
gcloud dataflow jobs run \
--gcs-location=gs://df-ts/latest/PubsubToBigQuery \
--parameters inputTopic=X outputTable=Y
```

Google Cloud

Google-provided templates documentation

How-to guides

- All how-to guides
- Installing the SDK
- Creating a pipeline
- Specifying execution parameters
- Deploying a pipeline
- Using the monitoring UI
- Using the command-line interface
- Using Stackdriver Monitoring
- Logging pipeline messages
- Troubleshooting your pipeline
- Updating an existing pipeline
- Stopping a running pipeline
- Creating and executing templates
  - Overview
  - Google-provided templates
    - Get started
    - Streaming templates
    - Batch templates
    - Utility templates
  - Creating templates
  - Executing templates
- Migrating from MapReduce
- Migrating from SDK 1.x for Java
- Configuring networking
- Using Cloud Pub/Sub Seek
- Using Flexible Resource Scheduling
- Creating Cloud Dataflow SQL jobs

Cloud Dataflow > Documentation

Get started with Google-provided templates

SEND FEEDBACK

Contents

WordCount

Google provides a set of open-source Cloud Dataflow templates. For general information about templates, see the Overview page. To get started, use the WordCount template documented in the section below. See other Google-provided templates:

Streaming templates - Templates for processing data continuously:

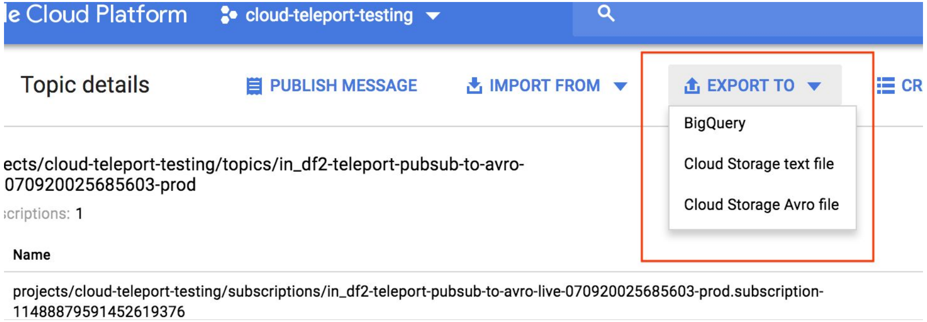
- Cloud Pub/Sub Subscription to BigQuery
- Cloud Pub/Sub Topic to BigQuery
- Cloud Pub/Sub to Cloud Pub/Sub
- Cloud Pub/Sub to Cloud Storage Avro
- Cloud Pub/Sub to Cloud Storage Text
- Cloud Storage Text to BigQuery (Stream)
- Cloud Storage Text to Cloud Pub/Sub (Stream)
- Data Masking/Tokenization using Cloud DLP from Cloud Storage to BigQuery (Stream)

Batch templates - Templates for processing data in bulk:

- Cloud Bigtable to Cloud Storage Avro
- Cloud Bigtable to Cloud Storage SequenceFiles
- Cloud Datastore to Cloud Storage Text
- Cloud Spanner to Cloud Storage Avro
- Cloud Spanner to Cloud Storage Text
- Cloud Storage Avro to Cloud Bigtable

Use cases of Google-provided templates

- Code-free routine job launcher for data engineers
- Building block for import/export feature of other services on GCP
- OSS code base works as good knowledge base



Cloud Pub/Sub

Cloud Spanner

Cloud BigTable

Which means now you can...

- Launch Dataflow jobs programmatically (via API).
- Launch Dataflow jobs instantaneously.
- Re-use Dataflow jobs
- Letting you customize the execution of your pipeline

What if you want to create your own template?

- Doc: https://cloud.google.com/dataflow/docs/templates/overview
- Steps
  1. Modify pipeline options with ValueProviders.
  2. Generate template file.

```
mvn compile exec:java \
 -Dexec.mainClass=com.example.myclass \
 -Dexec.args="--runner=DataflowRunner \
              --project=[YOUR_PROJECT_ID] \
              --stagingLocation=gs://[YOUR_BUCKET_NAME]/staging \
              --output=gs://[YOUR_BUCKET_NAME]/output \
              --templateLocation=gs://[YOUR_BUCKET_NAME]/templates/MyTemplate"
```

  3. Call it from API.

```
POST https://dataflow.googleapis.com/v1b3/projects/[YOUR_PROJECT_ID]/templates:launch?gcsPath=gs://[
{
    "jobName": "[JOB_NAME]",
    "parameters": {
        "inputFile" : "gs://[YOUR_BUCKET_NAME]/input/my_input.txt",
        "outputFile": "gs://[YOUR_BUCKET_NAME]/output/my_output"
    },
    "environment": {
        "tempLocation": "gs://[YOUR_BUCKET_NAME]/temp",
        "zone": "us-central1-f"
    }
}
```

Templates require modifying parameters for runtime

Python

```
class WordcountOptions(PipelineOptions):
    @classmethod
    def _add_argparse_args(cls, parser):
      parser.add_value_provider_argument(
          '--input',
          default='gs://dataflow-samples/shakespeare/kinglear.txt',
          help='Path of the file to read from')
      parser.add_argument(
          '--output',
          required=True,
          help='Output file to write results to.')
  pipeline_options = PipelineOptions(['--output', 'some/output_path'])
  p = beam.Pipeline(options=pipeline_options)

  wordcount_options = pipeline_options.view_as(WordcountOptions)
  lines = p | 'read' >> ReadFromText(wordcount_options.input)
```

Run-time parameters

Non run-time parameters can stay

Runtime parameters must be modified

Creating a template

- ValueProviders are passed down throughout the whole pipeline construction phase
- ValueProvider.get() only available in processElement()
  - Because it is fulfilled via API call

```
public interface SumIntOptions extends PipelineOptions {
    // New runtime parameter, specified by the --int
    // option at runtime.
    ValueProvider<Integer> getInt();
    void setInt(ValueProvider<Integer> value);
}

class MySumFn extends DoFn<Integer, Integer> {
    ValueProvider<Integer> mySumInteger;

    MySumFn(ValueProvider<Integer> sumInt) {
        // Store the value provider
        this.mySumInteger = sumInt;
    }

    @ProcessElement
    public void processElement(ProcessContext c) {
       // Get the value of the value provider and add it to
       // the element's value.
       c.output(c.element() + mySumInteger.get());
    }
}

public static void main(String[] args) {
  SumIntOptions options =
        PipelineOptionsFactory.fromArgs(args).withValidation()
          .as(SumIntOptions.class);
```

Nested Value Providers

Sometimes we need to transform a value from what the user passes at Runtime to what a Source/Sink expects to consume

NestedValueProviders meet this need

```
public static void main(String[] args) {
  pipeline
     .apply(Create.of(1, 2, 3).withCoder(BigEndianIntegerCoder.of()));
    // Write to the computed complete file path
     .apply("OutputNums", TextIO.write().to(NestedValueProvider.of(
      options.getFileName(),
      new SerializableFunction<String, String>() {
        @Override
        public String apply(String file) {
          return "gs://bucket/" + file;
        }
      })));
  pipeline.run();
}
```

Template Metadata

- Located at the same directory, named <template_name>_metadata

```
{
  "name": "WordCount",
  "description": "An example pipeline that counts words in the input file.",
  "parameters": [{
    "name": "inputFile",
    "label": "Input Cloud Storage File(s)",
    "help_text": "Path of the file pattern glob to read from.",
    "regexes": ["^gs:\/\/[^\n\r]+$"],
    "is_optional": true
  },
  {
    "name": "output",
    "label": "Output Cloud Storage File Prefix",
    "help_text": "Path and filename prefix for writing output files. ex: gs://MyBucket/counts",
    "regexes": ["^gs:\/\/[^\n\r]+$"]
  }]
}
```

Agenda

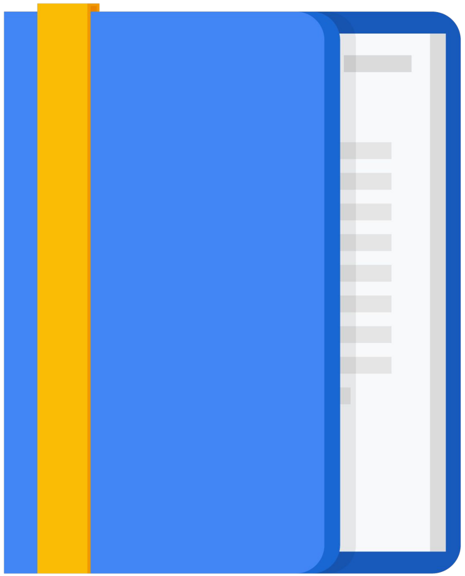
Cloud Dataflow

Why customers value Dataflow

Dataflow Pipelines

Dataflow Templates

Dataflow SQL



Cloud Dataflow SQL lets you use SQL queries to develop and run Cloud Dataflow jobs from the BigQuery web UI

Query editor

```
SELECT
  sr.sales_region,
  TUMBLE_START("INTERVAL 15 SECOND") AS period_start,
  SUM(tr.payload.amount) as amount
FROM pubsub.topic.`dataflow-sql`.transactions AS tr
  INNER JOIN bigquery.table.`dataflow-sql`.dataflow_sql_dataset.us_state_salesregions AS sr
  ON tr.payload.state = sr.state_code
GROUP BY
  sr.sales_region,
  TUMBLE(tr.event_timestamp, "INTERVAL 15 SECOND")
```

Valid.

Cloud Dataflow engine alpha

Create Cloud Dataflow job

More