Google Cloud

Preprocessing and feature creation

Carl Osipov

Learn how to...

Get started with preprocessing and feature creation

Learn how to...

Get started with preprocessing and feature creation

Use Apache Beam and Cloud Dataflow for feature engineering

Feature engineering often requires global statistics and vocabularies

```
features['scaled_price'] =
        (features['price'] - min_price) / (max_price - min_price)
```

```
tf.feature_column.categorical_column_with_vocabulary_list('city',
    keys=['San Diego', 'Los Angeles', 'San Francisco', Sacramento']),
```

How do you get full vocabulary of cities in the training dataset?

Feature engineering often requires global statistics and vocabularies

```
features['scaled_price'] =
       (features['price'] - min_price) / (max_price - min_price)
```

```
tf.feature_column.categorical_column_with_vocabulary_list('city',
    keys=['San Diego', 'Los Angeles', 'San Francisco', Sacramento']),
```

How do you get full vocabulary of cities in the training dataset?

Feature engineering often requires global statistics and vocabularies

```
features['scaled_price'] =
       (features['price'] - min_price) / (max_price - min_price)
```

```
tf.feature_column.categorical_column_with_vocabulary_list('city',
    keys=['San Diego', 'Los Angeles', 'San Francisco', Sacramento']),
```

How do you get full vocabulary of cities in the training dataset?

Feature engineering often requires global statistics and vocabularies

```
features['scaled_price'] =
       (features['price'] - min_price) / (max_price - min_price)
```

```
tf.feature_column.categorical_column_with_vocabulary_list('city',
    keys=['San Diego', 'Los Angeles', 'San Francisco', Sacramento']),
```

How do you get full vocabulary of cities in the training dataset?

Preprocess with...

1. BigQuery
2. Apache Beam
3. TensorFlow

Things that are commonly done in preprocessing

Remove examples that you don't want to train on

In BigQuery
or Beam

Things that are commonly done in preprocessing

Remove examples that you don’t want to train on

Compute vocabularies for categorical columns

Compute aggregate statistics for numeric columns

In BigQuery
or Beam

Things that are commonly done in preprocessing

Remove examples that you don't want to train on

Compute vocabularies for categorical columns

Compute aggregate statistics for numeric columns

In BigQuery or Beam

Compute time-windowed statistics (e.g. number of products sold in previous hour) for use as input features

In Beam only

Things that are commonly done in preprocessing

Scaling, discretization, etc. of numeric features

Splitting, lower-casing, etc. of textual features

Resizing of input images

Normalizing volume level of input audio

In TensorFlow or Beam

Example of preprocessing in BigQuery

```
SELECT
  (tolls_amount + fare_amount)
     AS fare_amount,
  DAYOFWEEK(pickup_datetime)
     AS dayofweek,
  HOUR(pickup_datetime)
          AS hourofday,
    ...
  FROM
    `nyc-tlc.yellow.trips`
  WHERE
    trip_distance > 0
```

Example of preprocessing in BigQuery

```
SELECT
  (tolls_amount + fare_amount)
     AS fare_amount,
  DAYOFWEEK(pickup_datetime)
     AS dayofweek,
  HOUR(pickup_datetime)
           AS hourofday,
    ...
  FROM
    `nyc-tlc.yellow.trips`
  WHERE
    trip_distance > 0
```

Example of preprocessing in BigQuery

```
SELECT
  (tolls_amount + fare_amount)
     AS fare_amount,
   DAYOFWEEK(pickup_datetime)
     AS dayofweek,
   HOUR(pickup_datetime)
           AS hourofday,
    ...
   FROM
    `nyc-tlc.yellow.trips`
   WHERE
    trip_distance > 0
```

Example of preprocessing in BigQuery

```
SELECT
  (tolls_amount + fare_amount)
     AS fare_amount,
  DAYOFWEEK(pickup_datetime)
     AS dayofweek,
  HOUR(pickup_datetime)
           AS hourofday,
    ...
  FROM
    `nyc-tlc.yellow.trips`
  WHERE
    trip_distance > 0
```

There are two places for feature creation in TensorFlow

```
features['capped_rooms'] = tf.clip_by_value(
    features['rooms'] ,
    clip_value_min=0,
    clip_value_max=4
)
```

1. Features are preprocessed In input_FN (train, eval, serving)

```
lat = tf.feature_column.numeric_column('latitude')
dlat = tf.feature_column.bucketized_column(lat,
boundaries=np.arange(32,42,1).tolist())
```

2. Feature columns are Passed into the estimator during construction

There are two places for feature creation in TensorFlow

```
features['capped_rooms'] = tf.clip_by_value(
    features['rooms'] ,
    clip_value_min=0,
    clip_value_max=4
)
```

1. Features are preprocessed In input_FN (train, eval, serving)

```
lat = tf.feature_column.numeric_column('latitude')
dlat = tf.feature_column.bucketized_column(lat,
boundaries=np.arange(32,42,1).tolist())
```

2. Feature columns are Passed into the estimator during construction

1. Example of preprocessing in TensorFlow input_fn

```
def add_engineered(features):
    lat1 = features['pickuplat']
    ...
    dist = tf.sqrt(latdiff*latdiff + londiff*londiff)
    features['euclidean'] = dist
    return features
```

How do we make sure this function gets
called during both training and prediction?

1. Example of preprocessing in TensorFlow input_fn

```
def add_engineered(features):
    lat1 = features['pickuplat']
    ...
    dist = tf.sqrt(latdiff*latdiff + londiff*londiff)
    features['euclidean'] = dist
    return features
```

How do we make sure this function gets
called during both training and prediction?

Wrap features by call to the feature engineering to function

Wrap features in training/evaluation input function:

```
def input_fn():
  features = ...
  label = ...
  return add_engineered(features), label
```

Wrap features in serving input function also:

```
def serving_input_fn():
    feature_placeholders = ...
    features = ...
    return tf.estimator.export.ServingInputReceiver(
      add_engineered(features), feature_placeholders)
```

Wrap features by call to the feature engineering to function

Wrap features in training/evaluation input function:

```
def input_fn():
  features = ...
  label = ...
  return add_engineered(features), label
```

Wrap features in serving input function also:

```
def serving_input_fn():
    feature_placeholders = ...
    features = ...
    return tf.estimator.export.ServingInputReceiver(
      add_engineered(features), feature_placeholders)
```

Wrap features by call to the feature engineering to function

Wrap features in training/evaluation input function:

```
def input_fn():
  features = ...
  label = ...
  return add_engineered(features), label
```

Wrap features in serving input function also:

```
def serving_input_fn():
    feature_placeholders = ...
    features = ...
    return tf.estimator.export.ServingInputReceiver(
      add_engineered(features), feature_placeholders)
```

2. Example of preprocessing via feature columns

```
def build_estimator(model_dir, nbuckets):
  latbuckets = np.linspace(38.0, 42.0, nbuckets).tolist()
  b_plat = tf.feature_column.bucketized_column(plat, latbuckets)
  b_dlat = tf.feature_column.bucketized_column(dlat, latbuckets)

  return tf.estimator.LinearRegressor(
      model_dir=model_dir,
      feature_columns=[..., b_plat, b_dlat, …])
```

2. Example of preprocessing via feature columns

```
def build_estimator(model_dir, nbuckets):
  latbuckets = np.linspace(38.0, 42.0, nbuckets).tolist()
  b_plat = tf.feature_column.bucketized_column(plat, latbuckets)
  b_dlat = tf.feature_column.bucketized_column(dlat, latbuckets)

  return tf.estimator.LinearRegressor(
       model_dir=model_dir,
       feature_columns=[..., b_plat, b_dlat, ...])
```

2. Example of preprocessing via feature columns

```
def build_estimator(model_dir, nbuckets):
  latbuckets = np.linspace(38.0, 42.0, nbuckets).tolist()
  b_plat = tf.feature_column.bucketized_column(plat, latbuckets)
  b_dlat = tf.feature_column.bucketized_column(dlat, latbuckets)

  return tf.estimator.LinearRegressor(
      model_dir=model_dir,
      feature_columns=[..., b_plat, b_dlat, ...])
```

2. Normalization can be done in feature columns

```
def zscore(col):
  mean = 3.04
  std = 1.2
  return (col – mean)/std

feature_name = ‘total_bedrooms’
normalized_feature = tf.feature_column.numeric_column(
  feature_name,
  normalizer_fn=zscore)
```

Example of preprocessing in Beam (covered next)

```
def to_csv(rowdict):
  if distance(rowdict['pickuplon'], ...) > 10: # only rides of more than 10km
    CSV_COLUMNS = 'fare_amount,dayofweek,...,key'.split(',')
    yield ','.join([str(rowdict[k]) for k in CSV_COLUMNS])

def preprocess():
  ...
  for n, step in enumerate(['train', 'valid']):
     (p  | 'read_{}'.format(step) >>
beam.io.Read(beam.io.BigQuerySource(query=query))
        | 'tocsv_{}'.format(step) >> beam.FlatMap(to_csv)
        | 'write_{}'.format(step) >> beam.io.Write(beam.io.WriteToText(outfile))
     )
  p.run()
```

Recap: things that are commonly done in preprocessing

Remove examples that you don't want to train on

Compute vocabularies for categorical columns

Compute aggregate statistics for numeric columns

In BigQuery or Beam

Compute time-windowed statistics (e.g. number of products sold in previous hour) for use as input features

In Beam only

Scaling, discretization, etc. of numeric features

Splitting, lower-casing, etc. of textual features

Resizing of input images

Normalizing volume level of input audio

In TensorFlow or Beam

Apache Beam/Cloud Dataflow

Carl Osipov

Beam is a way to write elastic data processing pipelines

Cloud
Dataflow

Beam is a way to write elastic data processing pipelines

```python
def packageHelp(record, keyword):
   count=0
   package_name=''
   if record is not None:
     lines=record.split('\n')
     for line in lines:
       if line.startswith(keyword):
         package_name=line
       if 'FIXME' in line or 'TODO' in l
         count+=1
     packages = (getPackages(package_nam
     for p in packages:
         yield (p,count)
```

Cloud Dataflow

Beam is a way to write elastic data processing pipelines

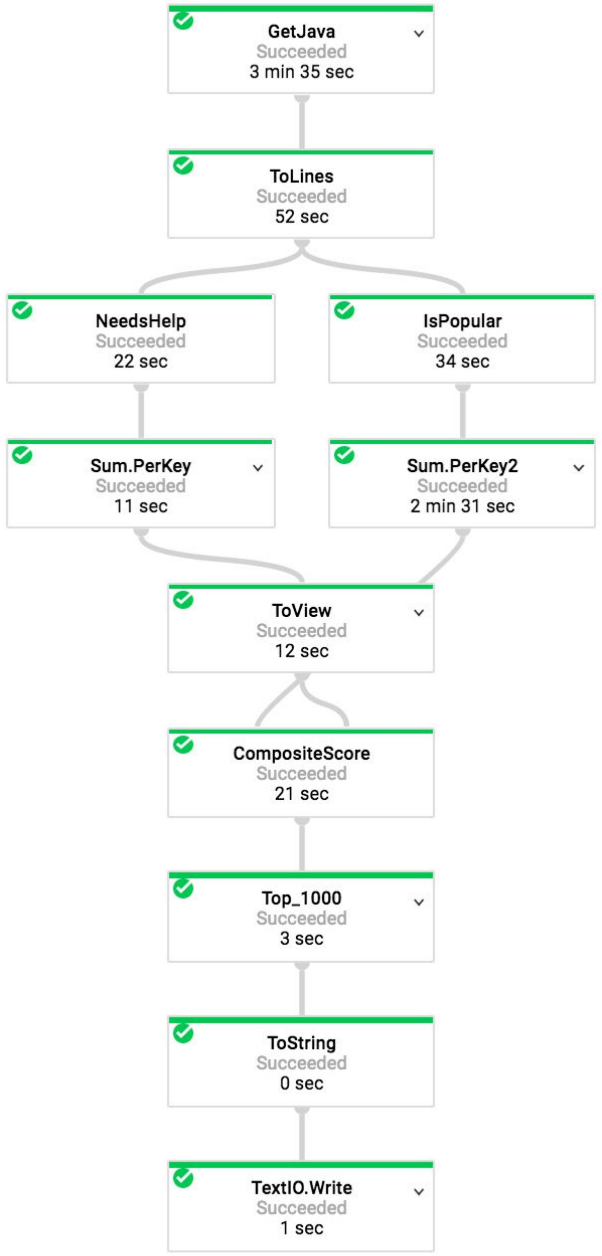
BigQuery

Beam is a way to write elastic data processing pipelines

```python
def packageHelp(record, keyword):
   count=0
   package_name=''
   if record is not None:
     lines=record.split('\n')
     for line in lines:
       if line.startswith(keyword):
         package_name=line
       if 'FIXME' in line or 'TODO' in l
         count+=1
     packages = (getPackages(package_nam
     for p in packages:
         yield (p,count)
```

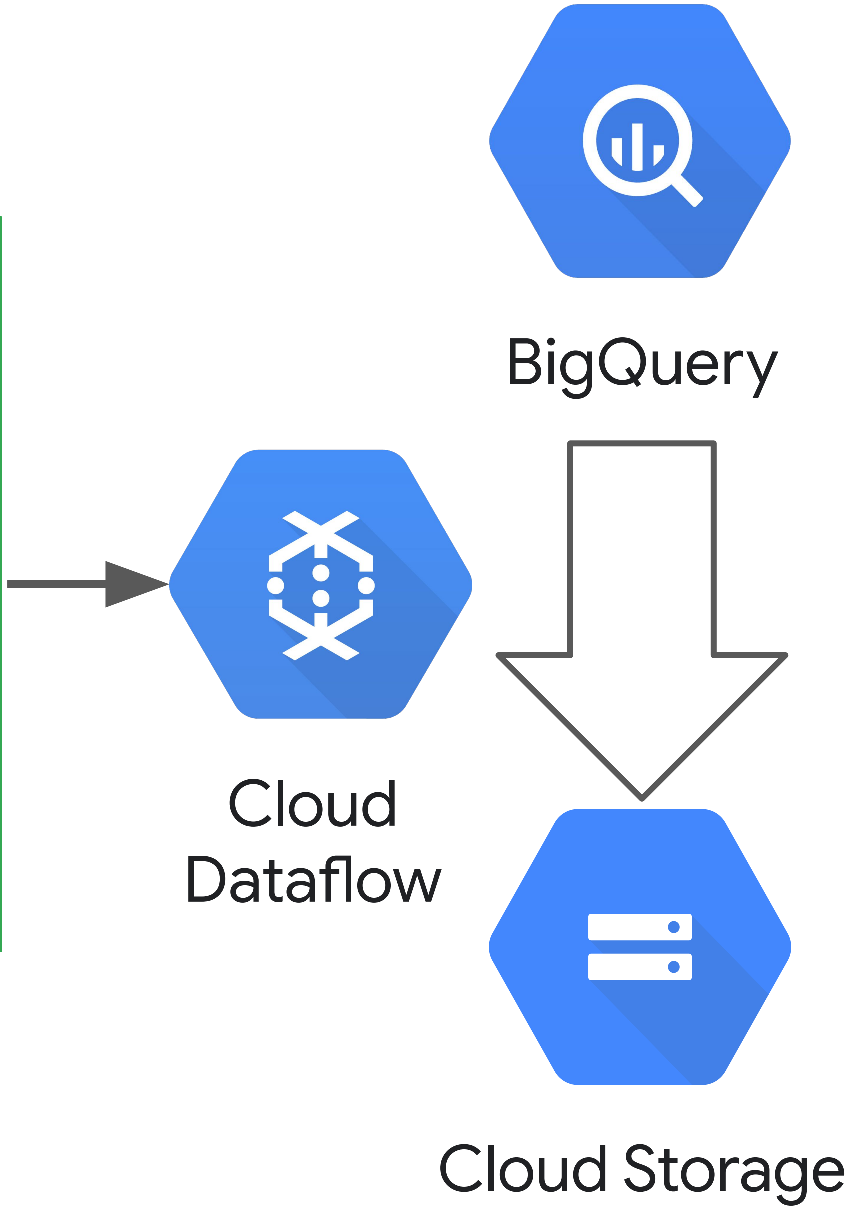
Cloud Dataflow

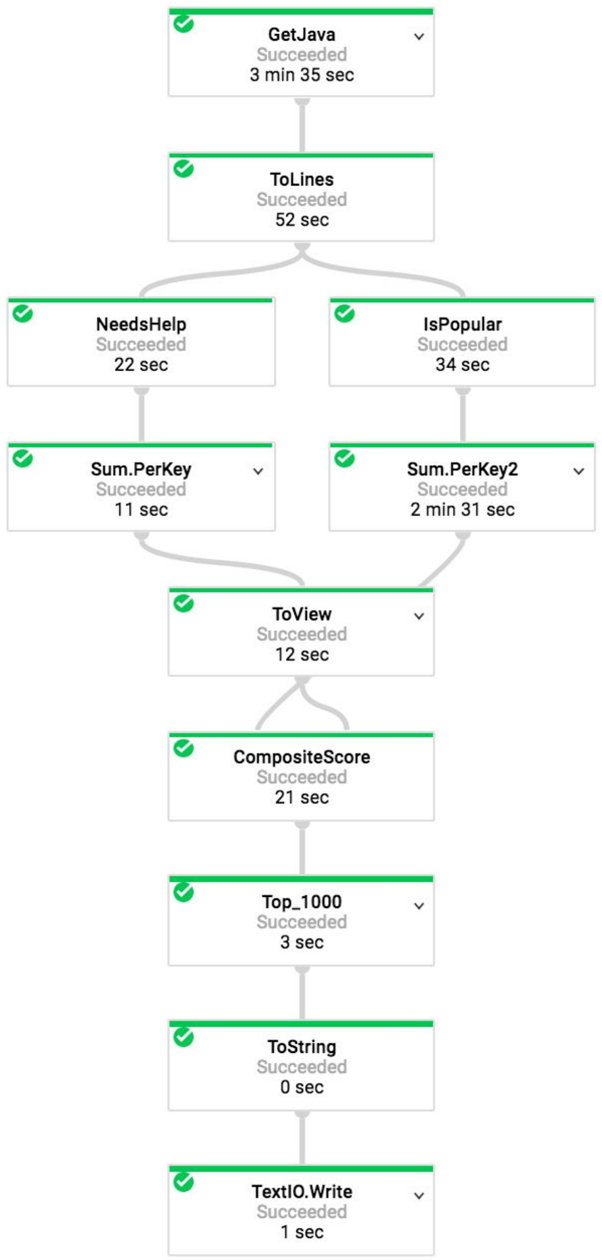
BigQuery



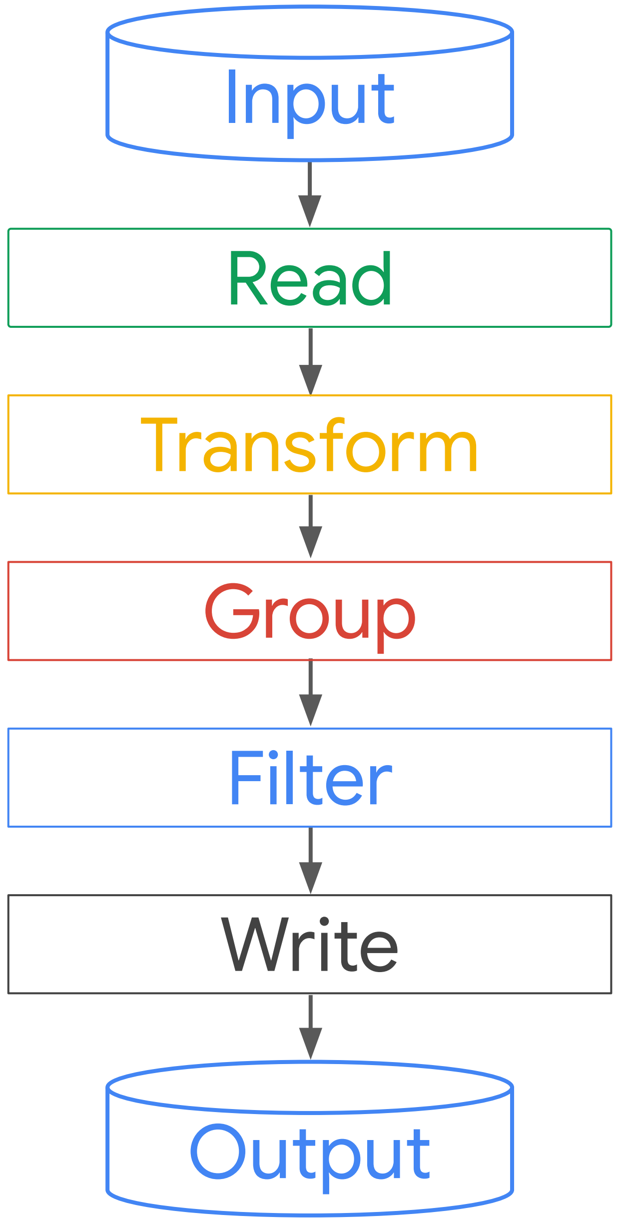
Beam is a way to write elastic data processing pipelines

```
def packageHelp(record, keyword):
   count=0
   package_name=''
   if record is not None:
     lines=record.split('\n')
     for line in lines:
       if line.startswith(keyword):
         package_name=line
       if 'FIXME' in line or 'TODO' in l
         count+=1
     packages = (getPackages(package_nam
     for p in packages:
         yield (p,count)
```





Open-source API, Google infrastructure



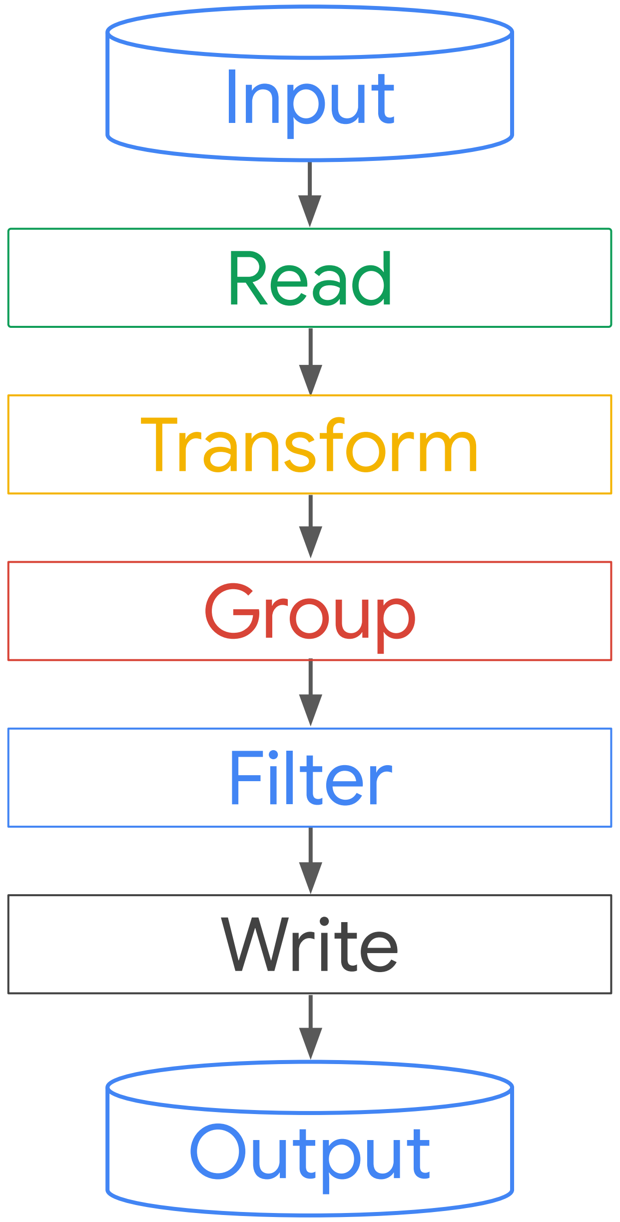

```
p = beam.Pipeline()
(p
     | beam.io.ReadFromText('gs://..')
     | beam.Map(Transform)
     | beam.GroupByKey()
     | beam.FlatMap(Filter)
     | beam.io.WriteToText('gs://...')
 )
p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
      return
(count_words(line), 1)

def Filter(key, values):
      return key > 10
```

Open-source API, Google infrastructure



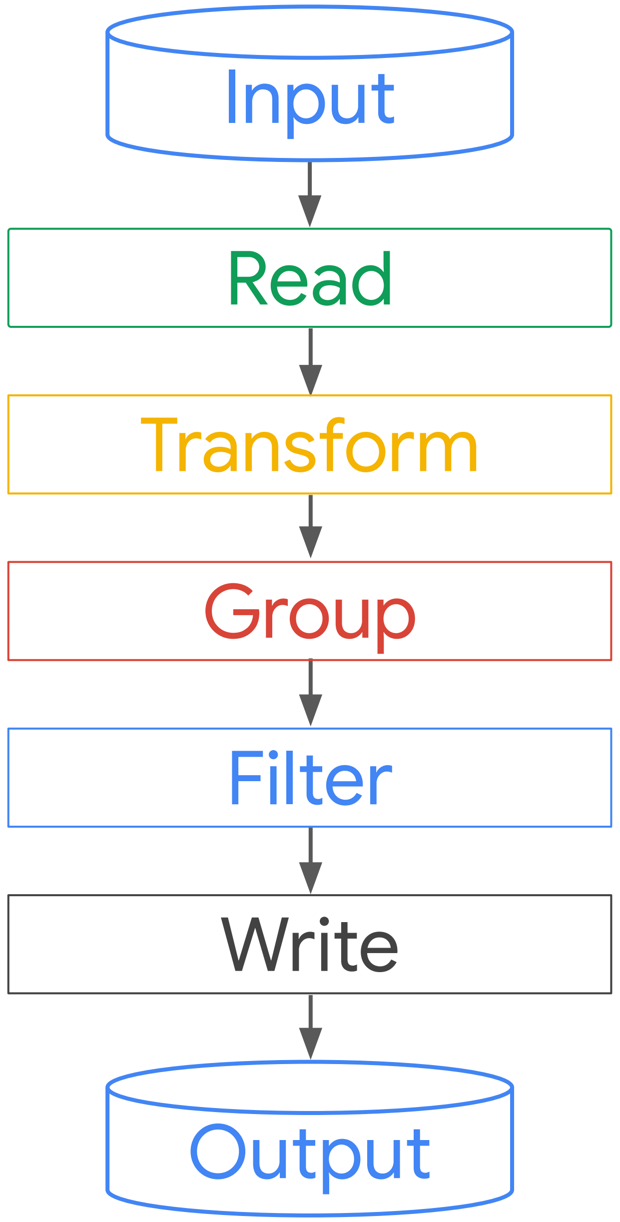

```
p = beam.Pipeline()
(p
    | beam.io.ReadFromText('gs://..')

    | beam.Map(Transform)

    | beam.GroupByKey()

    | beam.FlatMap(Filter)

    | beam.io.WriteToText('gs://...')
 )
p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
      return
(count_words(line), 1)

def Filter(key, values):
      return key > 10
```

Open-source API, Google infrastructure



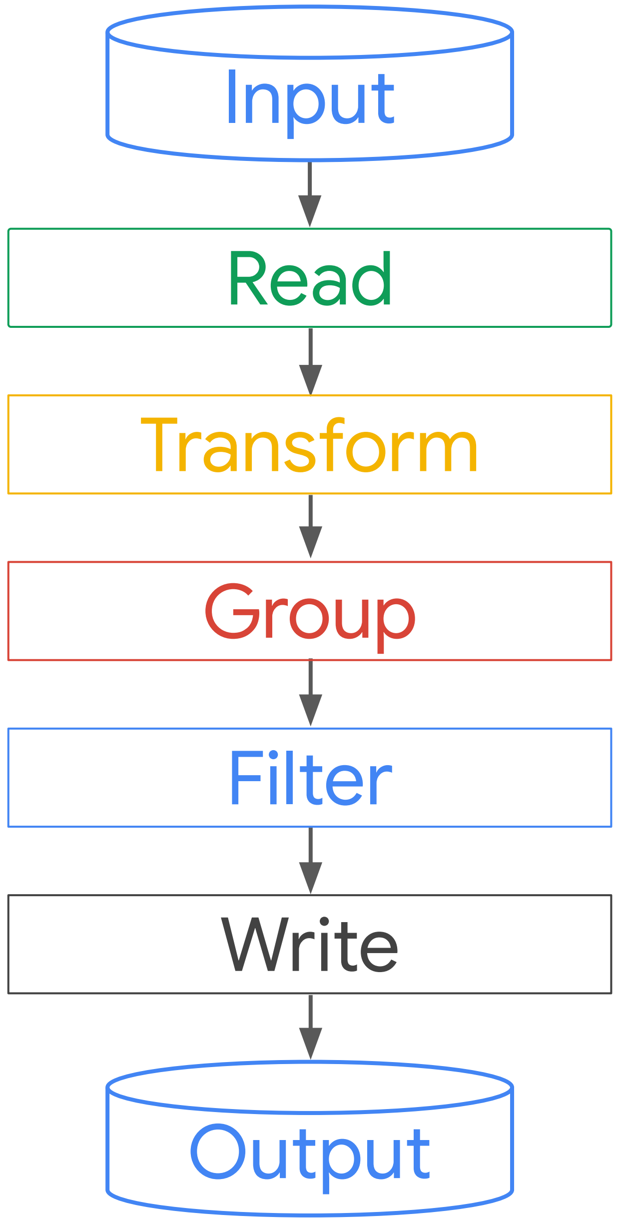

```
p = beam.Pipeline()
(p
    | beam.io.ReadFromText('gs://..')
    | beam.Map(Transform)
    | beam.GroupByKey()
    | beam.FlatMap(Filter)
    | beam.io.WriteToText('gs://...')
 )
p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
      return
(count_words(line), 1)

def Filter(key, values):
      return key > 10
```

Open-source API, Google infrastructure



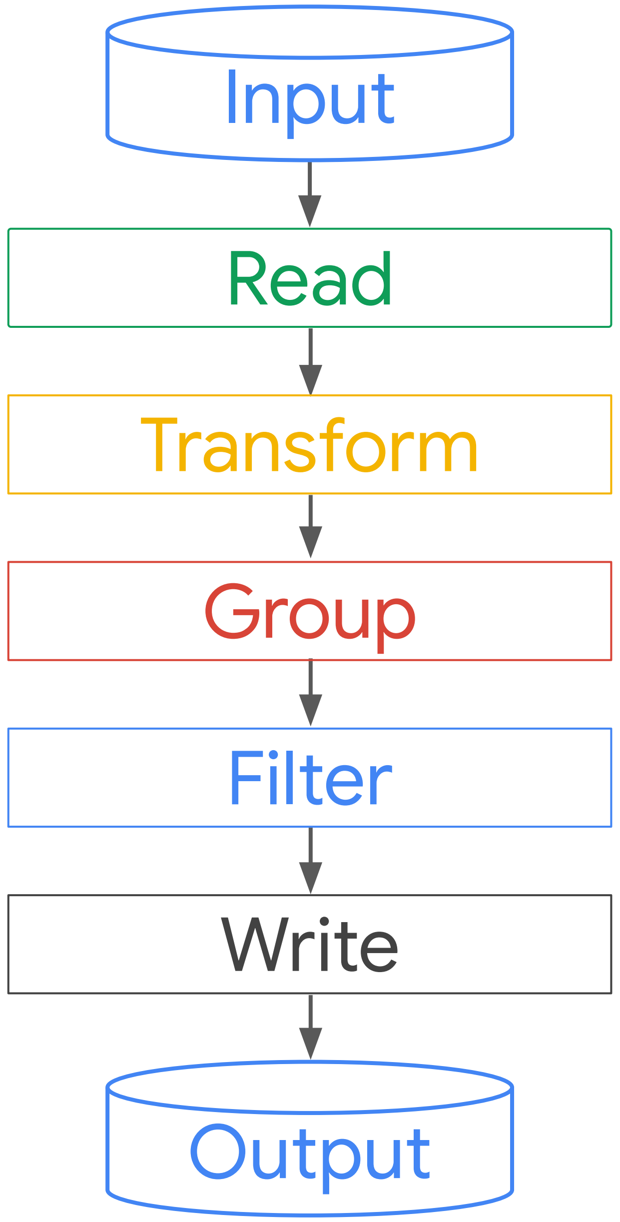

```
p = beam.Pipeline()
(p

      | beam.io.ReadFromText('gs://..')

      | beam.Map(Transform)

      | beam.GroupByKey()

      | beam.FlatMap(Filter)

      | beam.io.WriteToText('gs://...')
 )
p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
      return
(count_words(line), 1)

def Filter(key, values):
      return key > 10
```

Open-source API, Google infrastructure



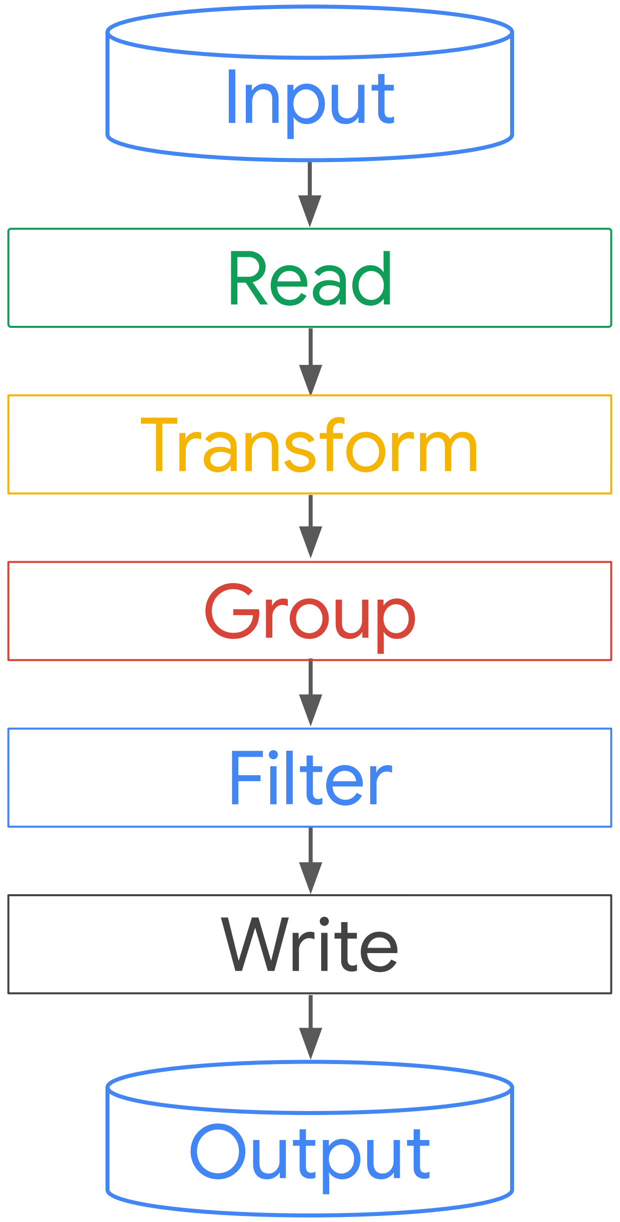

```
p = beam.Pipeline()
(p
      | beam.io.ReadFromText('gs://..')
      | beam.Map(Transform)
      | beam.GroupByKey()
      | beam.FlatMap(Filter)
      | beam.io.WriteToText('gs://...')
 )
p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
      return
(count_words(line), 1)

def Filter(key, values):
      return key > 10
```

Open-source API, Google infrastructure



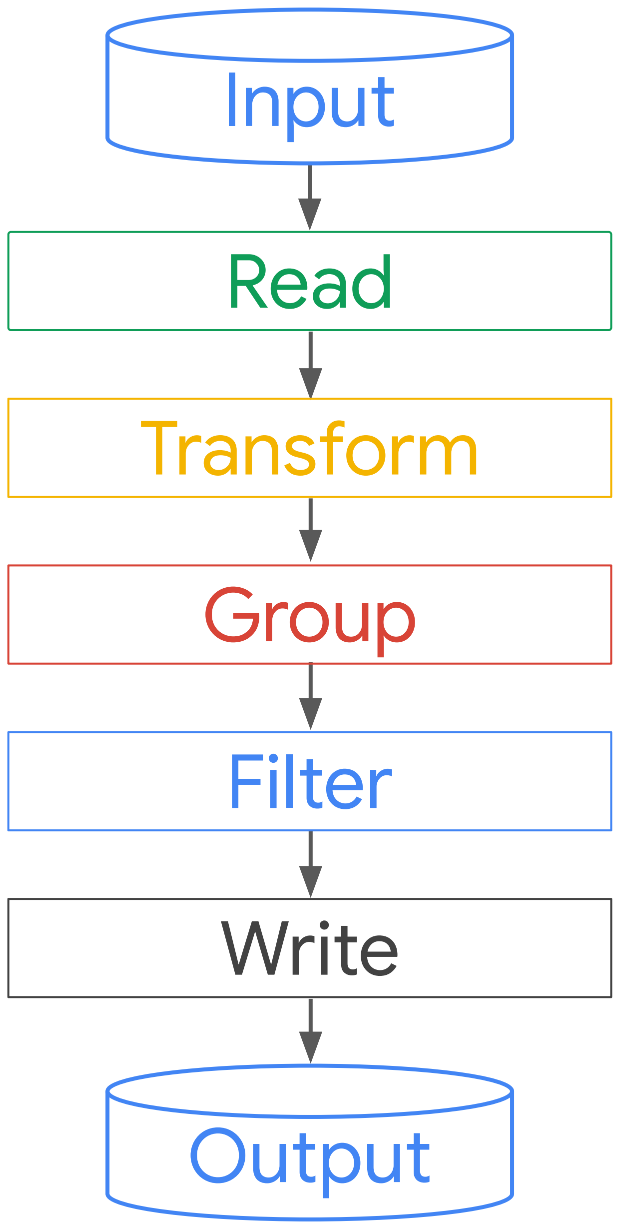

```
p = beam.Pipeline()
(p
    | beam.io.ReadFromText('gs://..')
    | beam.Map(Transform)
    | beam.GroupByKey()
    | beam.FlatMap(Filter)
    | beam.io.WriteToText('gs://...')
 )
p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
      return
(count_words(line), 1)

def Filter(key, values):
      return key > 10
```

Open-source API, Google infrastructure



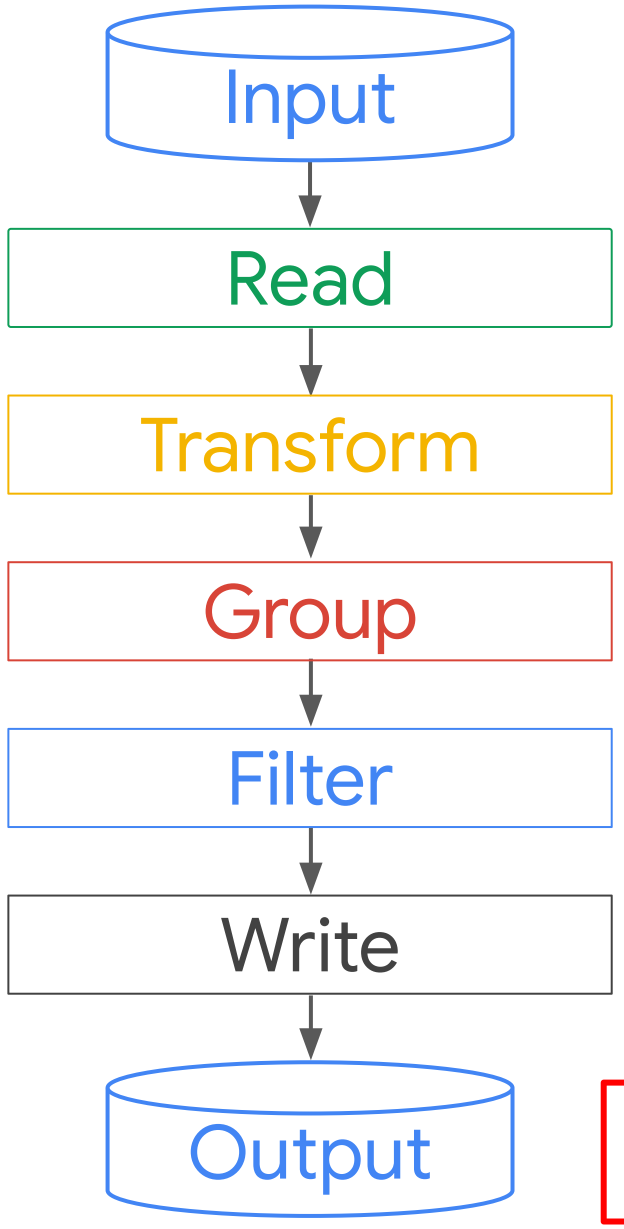

```
p = beam.Pipeline()
(p

      | beam.io.ReadFromText('gs://..')

      | beam.Map(Transform)

      | beam.GroupByKey()

      | beam.FlatMap(Filter)

      | beam.io.WriteToText('gs://...')
 )
p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
      return
(count_words(line), 1)

def Filter(key, values):
      return key > 10
```

Open-source API, Google infrastructure



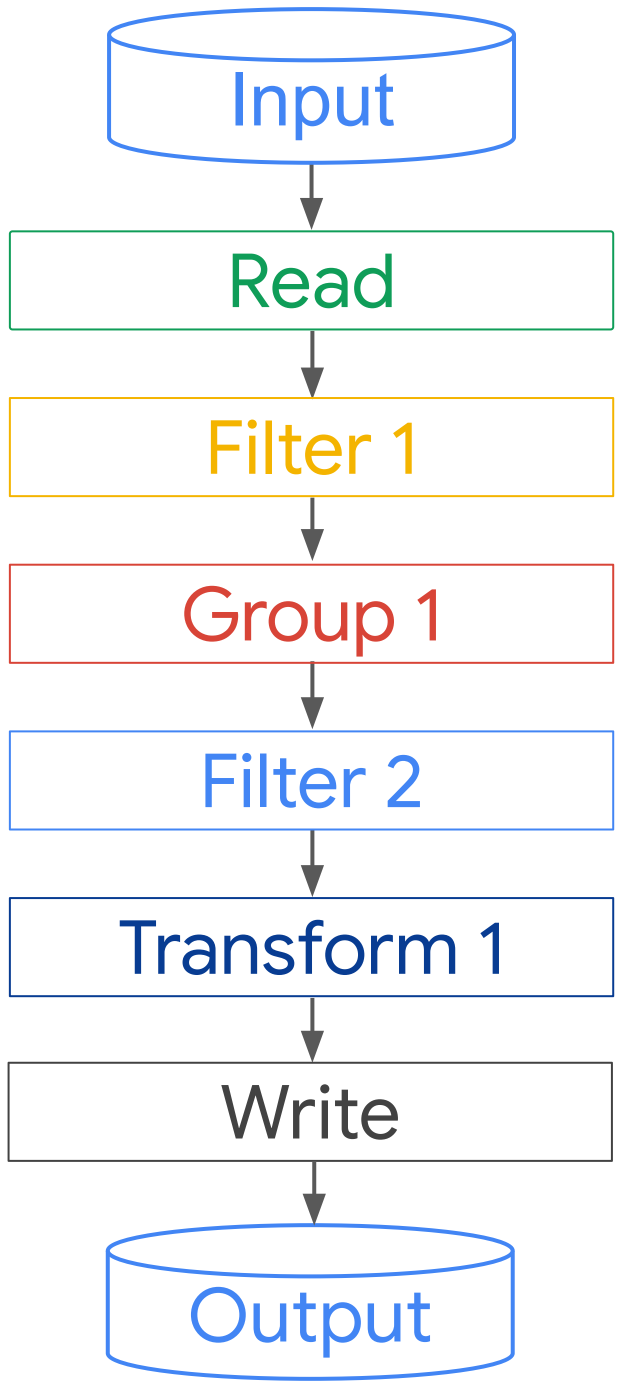

```
p = beam.Pipeline()
(p
     | beam.io.ReadFromText('gs://..')
     | beam.Map(Transform)
     | beam.GroupByKey()
     | beam.FlatMap(Filter)
     | beam.io.WriteToText('gs://...')
)
p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
      return
(count_words(line), 1)

def Filter(key, values):
      return key > 10
```

Open-source API, Google infrastructure



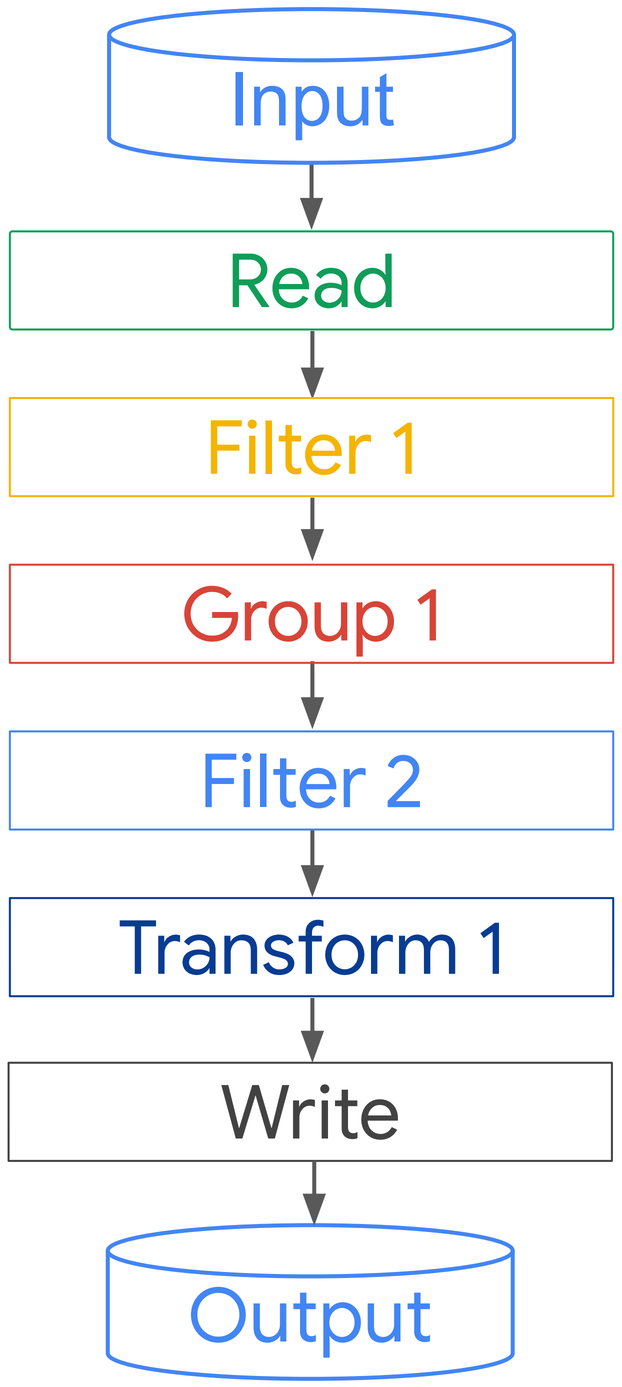

```
Pipeline p = Pipeline.create();
p

   .apply(TextIO.read().from("gs://…"))

   .apply(ParDo.of(new Filter1()))

   .apply(new Group1())

   .apply(ParDo.of(new Filter2())

   .apply(new Transform1())

   .apply(TextIO.write().to("gs://…"));

p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel task (autoscaled by execution framework)

```
class Filter1 extends DoFn<...> {
  public void
  processElement(ProcessContext c) {
      ... = c.element();
      ...
      c.output(...);
  }
```

Open-source API, Google infrastructure



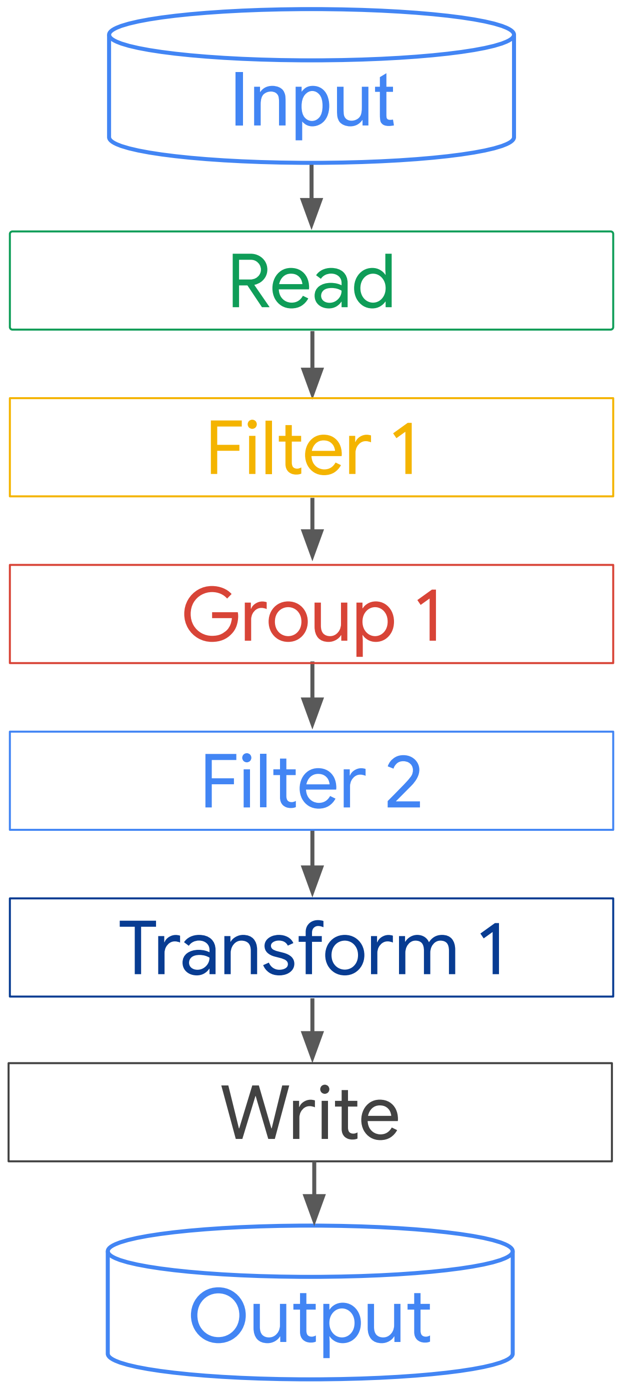

```
Pipeline p = Pipeline.create();
p
    .apply(TextIO.read().from("gs://…"))

    .apply(ParDo.of(new Filter1()))

    .apply(new Group1())

    .apply(ParDo.of(new Filter2())

    .apply(new Transform1())

    .apply(TextIO.write().to("gs://…"));

p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel task (autoscaled by execution framework)

```
class Filter1 extends DoFn<...> {
  public void
  processElement(ProcessContext c) {
      ... = c.element();
      ...
      c.output(...);
  }
```

Open-source API, Google infrastructure



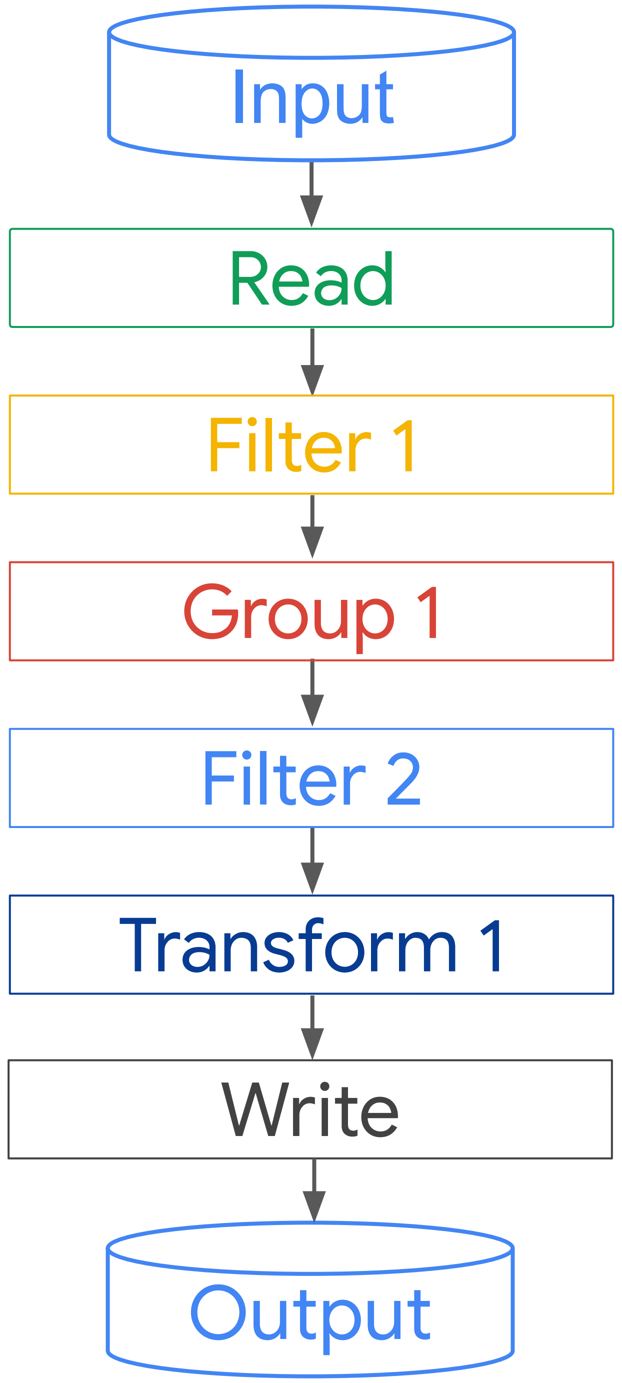

```
Pipeline p = Pipeline.create();
p

    .apply(TextIO.read().from(“gs://…”))

    .apply(ParDo.of(new Filter1()))

    .apply(new Group1())

    .apply(ParDo.of(new Filter2())

    .apply(new Transform1())

    .apply(TextIO.write().to(“gs://…”));

p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel task (autoscaled by execution framework)

```
class Filter1 extends DoFn<...> {
  public void
  processElement(ProcessContext c) {
      ... = c.element();
      ...
      c.output(...);
  }
```

Open-source API, Google infrastructure



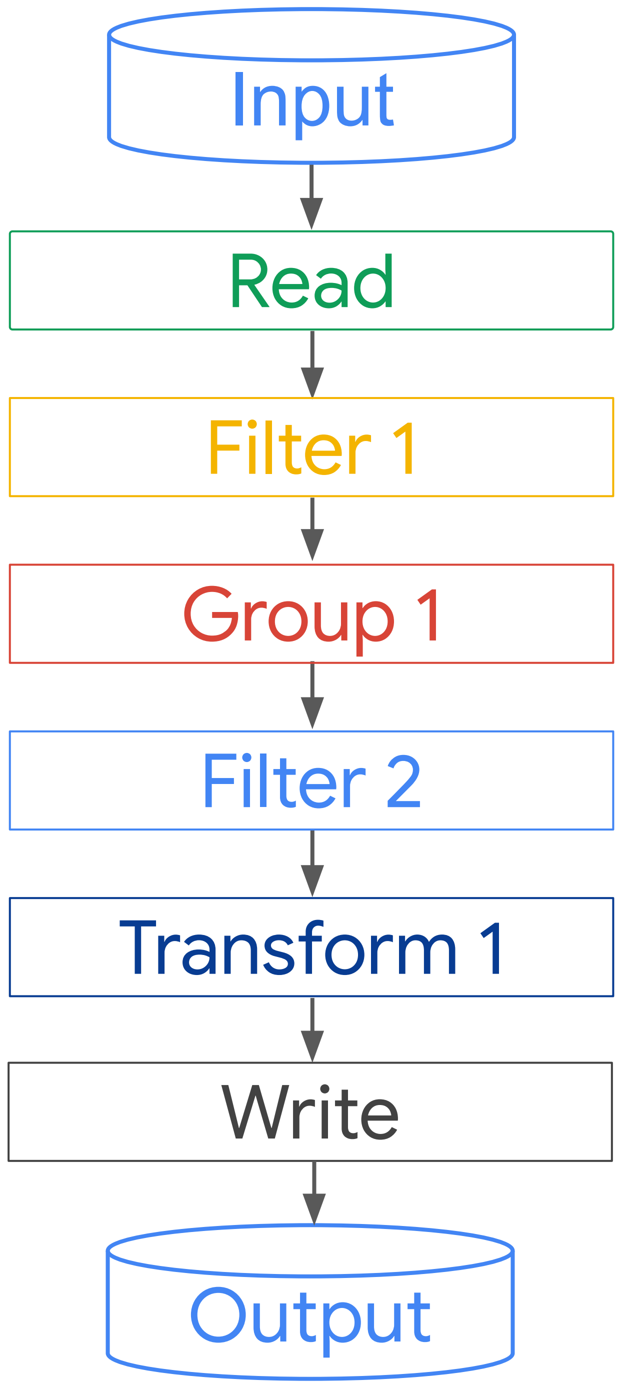

```
Pipeline p = Pipeline.create();
p
    .apply(TextIO.read().from("gs://..."))
    .apply(ParDo.of(new Filter1()))
    .apply(new Group1())
    .apply(ParDo.of(new Filter2())
    .apply(new Transform1())
    .apply(TextIO.write().to("gs://..."));

p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel task (autoscaled by execution framework)

```
class Filter1 extends DoFn<...> {
  public void
  processElement(ProcessContext c) {
      ... = c.element();
      ...
      c.output(...);
  }
```

Open-source API, Google infrastructure



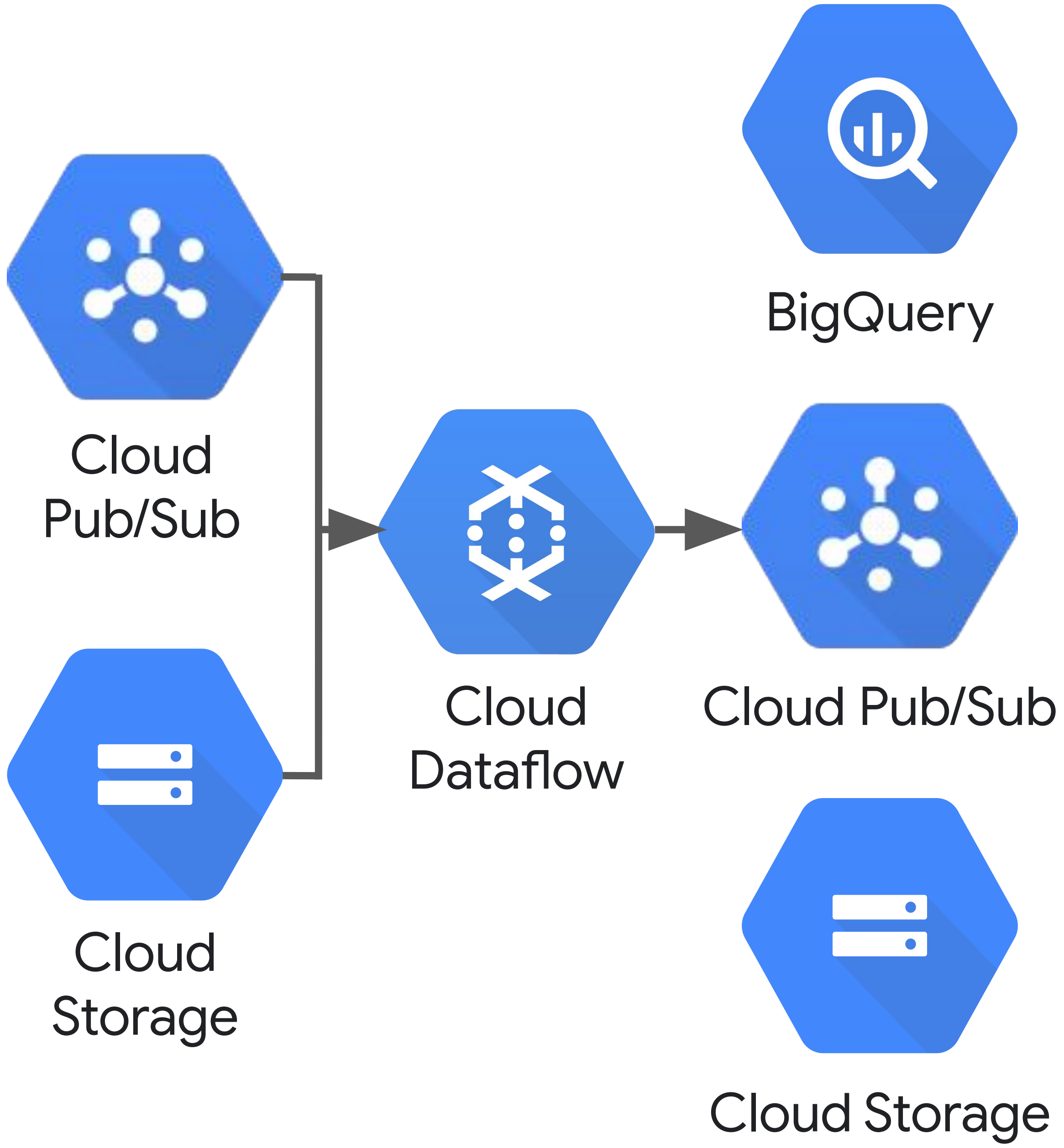

```
Pipeline p = Pipeline.create();
p

    .apply(TextIO.read().from(“gs://…”))

    .apply(ParDo.of(new Filter1()))

    .apply(new Group1())

    .apply(ParDo.of(new Filter2())

    .apply(new Transform1())

    .apply(TextIO.write().to(“gs://…”));

p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel task (autoscaled by execution framework)

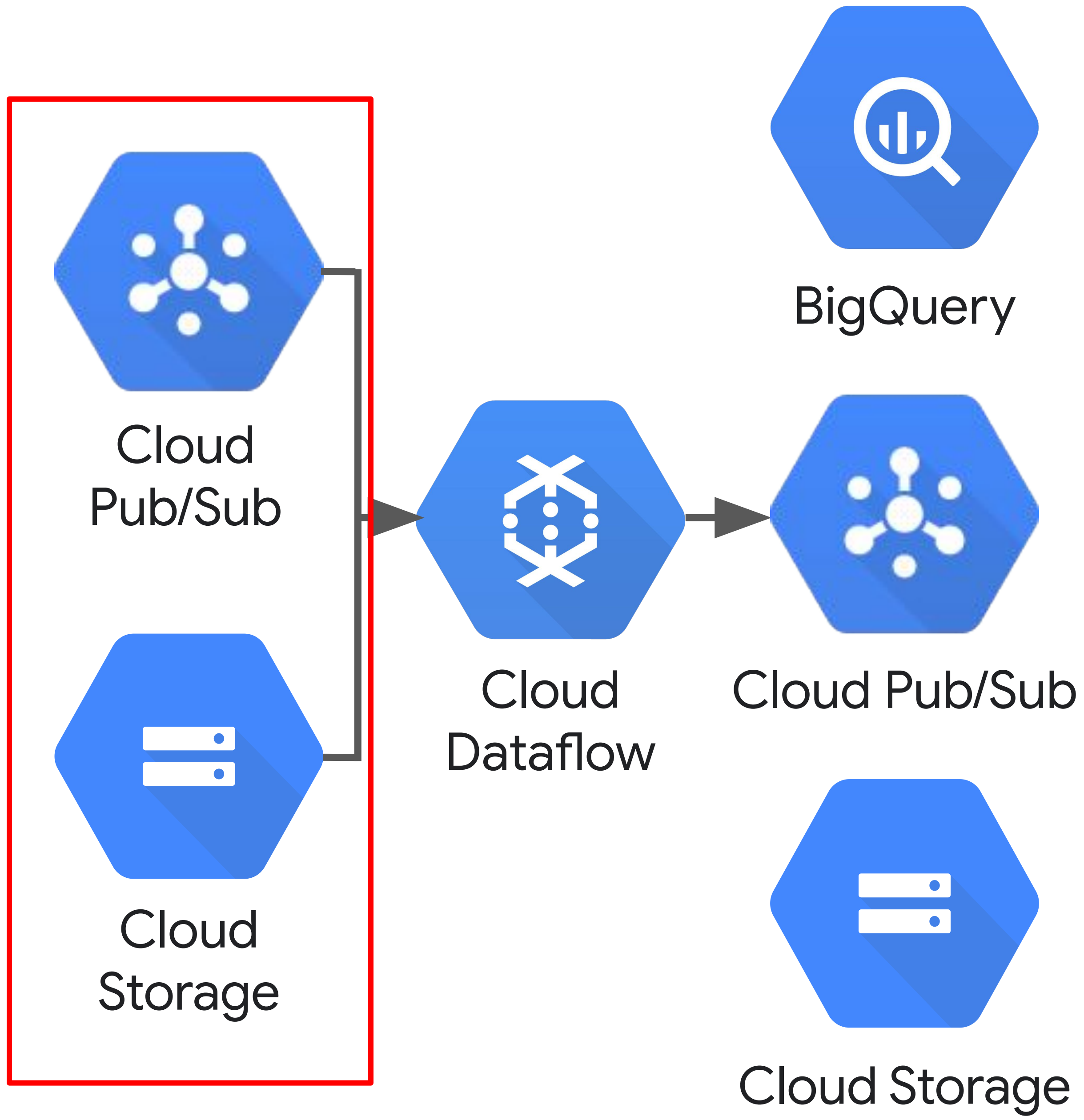
```
class Filter1 extends DoFn<...> {
  public void
  processElement(ProcessContext c) {
      ... = c.element();
      ...
      c.output(...);
  }
```

The code is the same between real-time and batch (Java)



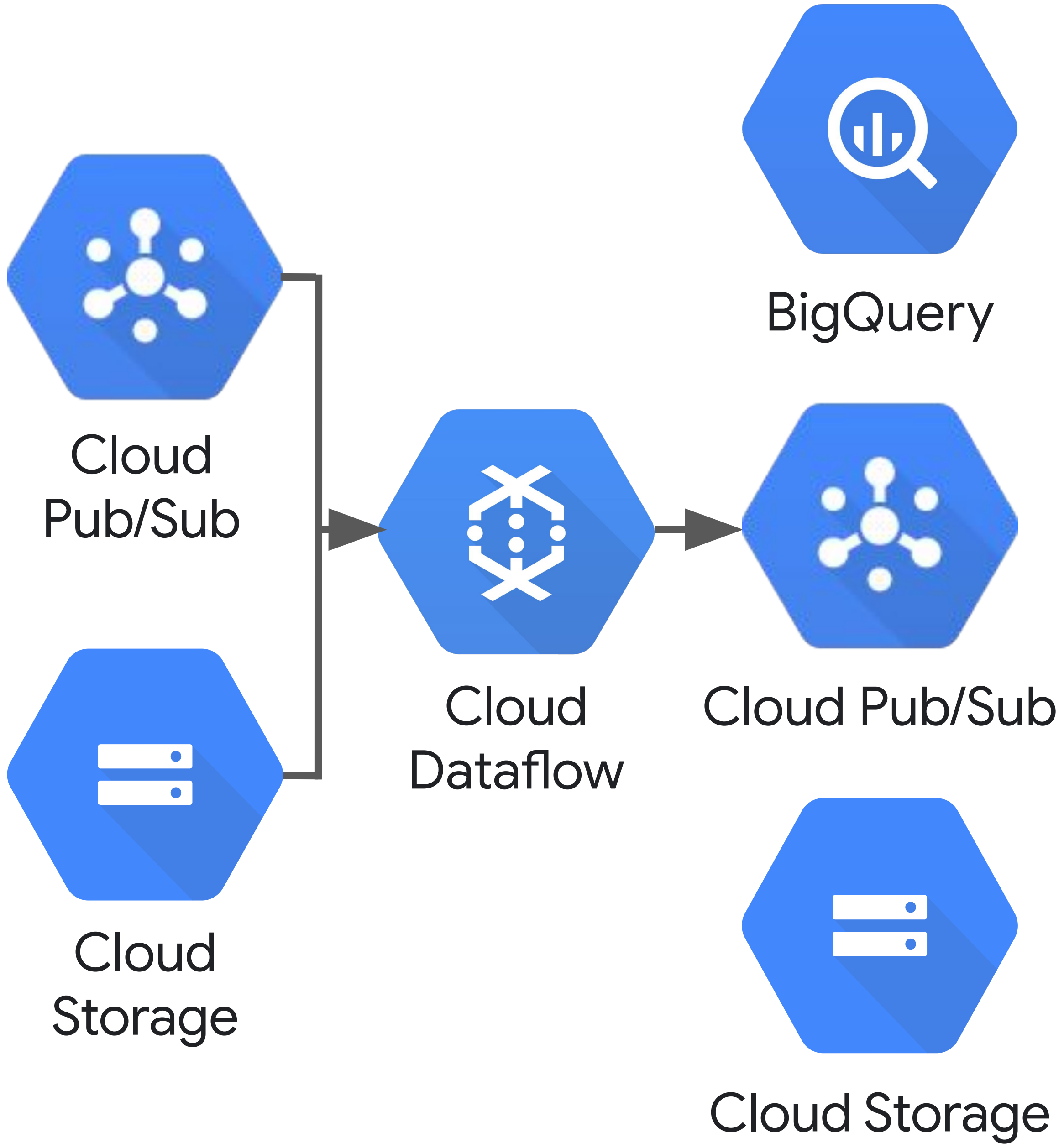

```
p = beam.Pipeline()
(p
    | beam.io.ReadStringsFromPubSub('project/topic')
    | beam.WindowInto(SlidingWindows(60))
    | beam.Map(Transform)
    | beam.GroupByKey()
    | beam.FlatMap(Filter)
    | beam.io.WriteToBigQuery(table)
)
p.run()
```

The code is the same between real-time and batch (Java)



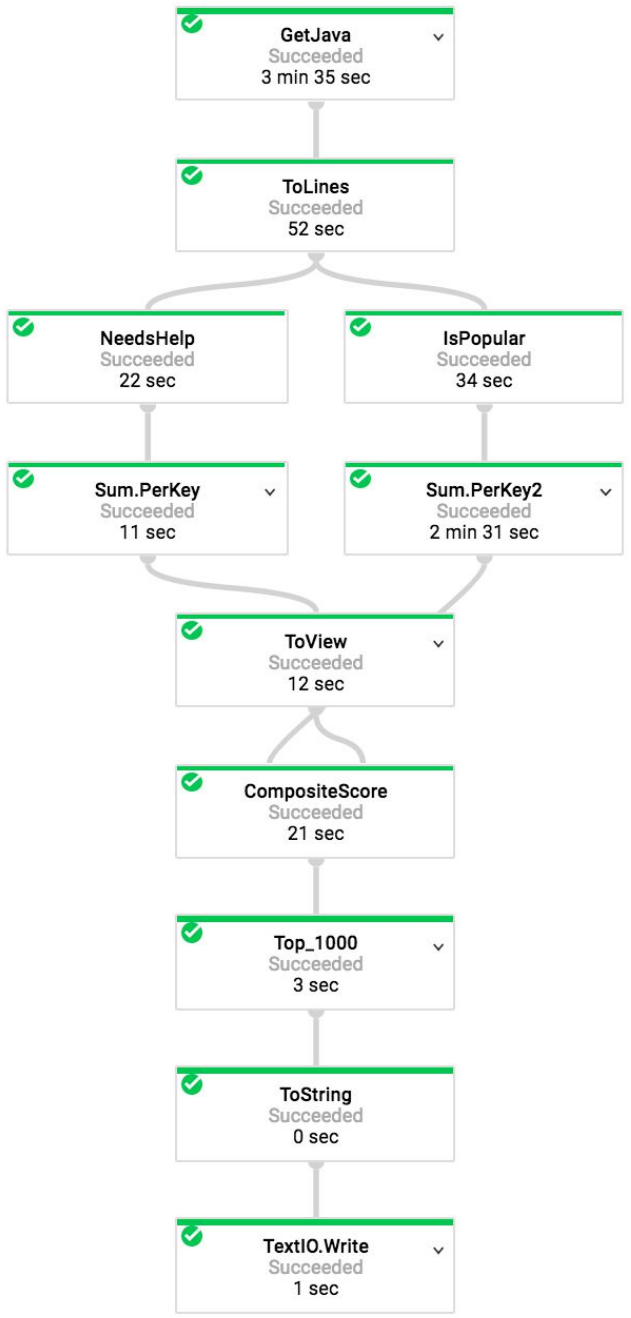

```
p = beam.Pipeline()
(p
    | beam.io.ReadStringsFromPubSub('project/topic')
    | beam.WindowInto(SlidingWindows(60))
    | beam.Map(Transform)
    | beam.GroupByKey()
    | beam.FlatMap(Filter)
    | beam.io.WriteToBigQuery(table)
)
p.run()
```

The code is the same between real-time and batch (Java)

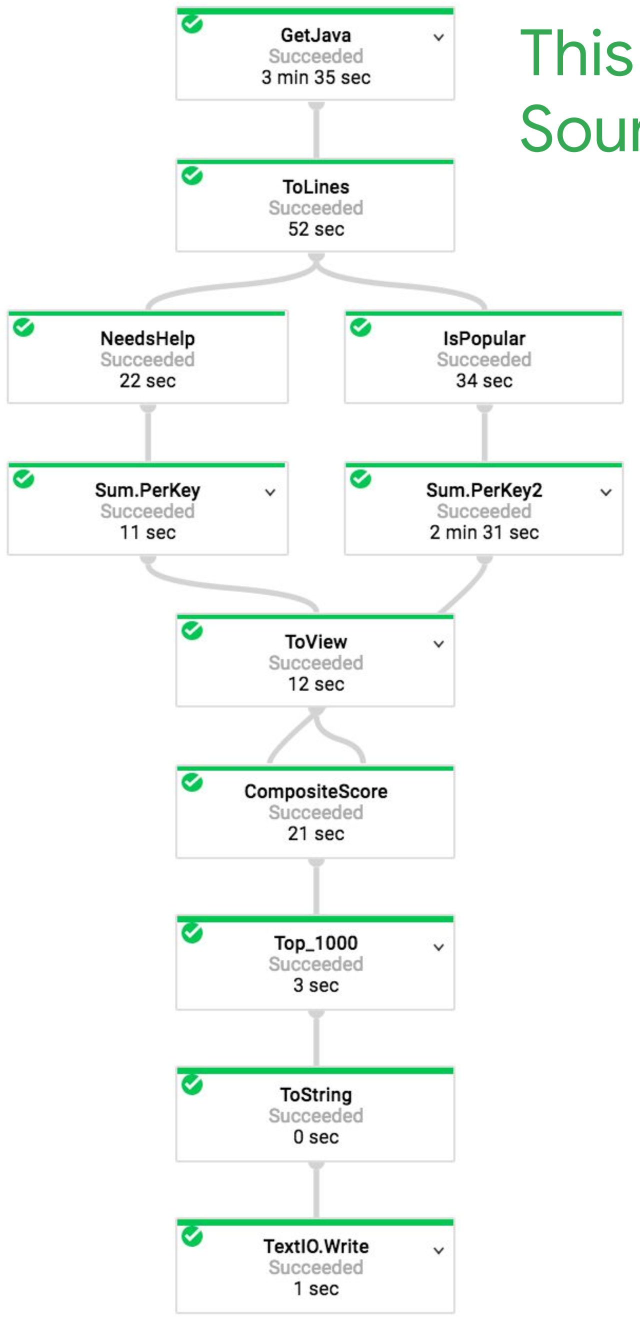



```
p = beam.Pipeline()
(p
   | beam.io.ReadStringsFromPubSub('project/topic')
   | beam.WindowInto(SlidingWindows(60))
   | beam.Map(Transform)
   | beam.GroupByKey()
   | beam.FlatMap(Filter)
   | beam.io.WriteToBigQuery(table)
)
p.run()
```

Dataflow terms and concepts

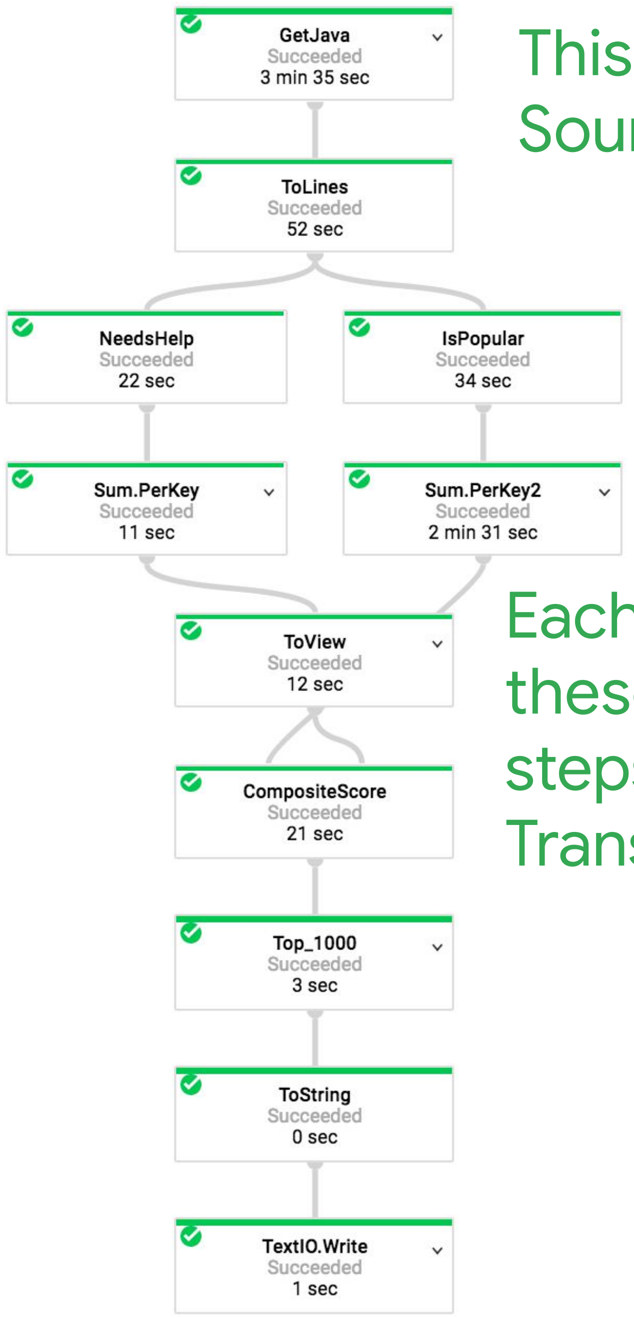


Dataflow terms and concepts



This is a Source

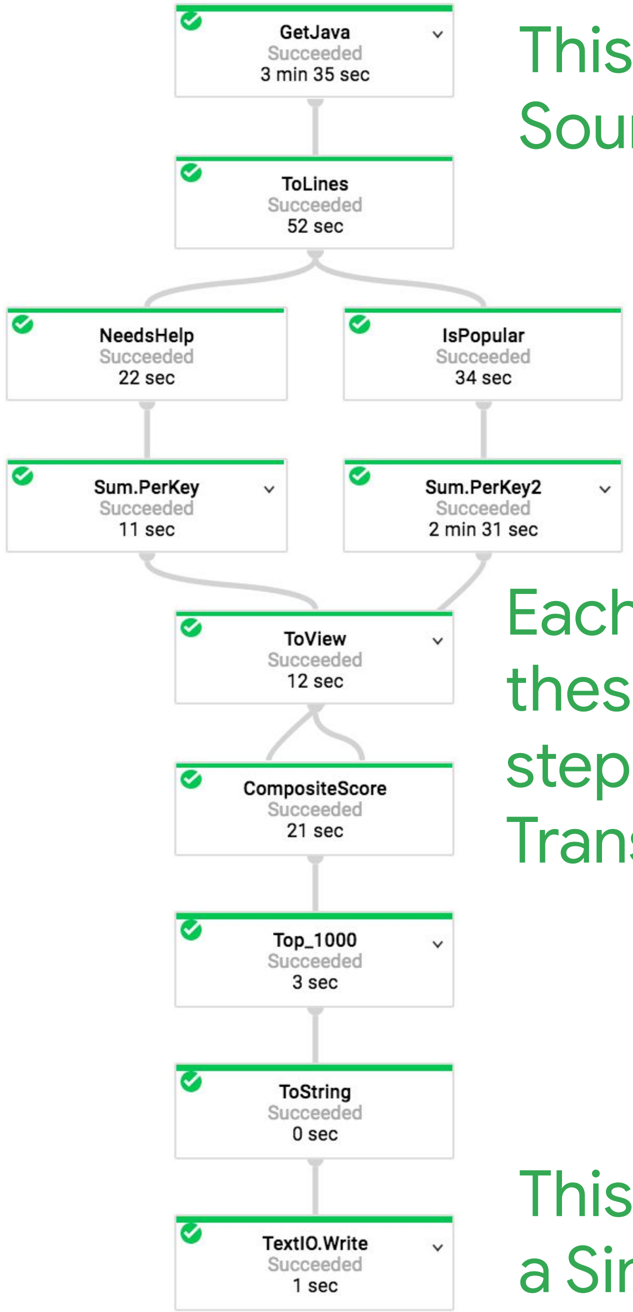
Dataflow terms and concepts



This is a Source

Each of these steps is a Transform

Dataflow terms and concepts



This is a Source

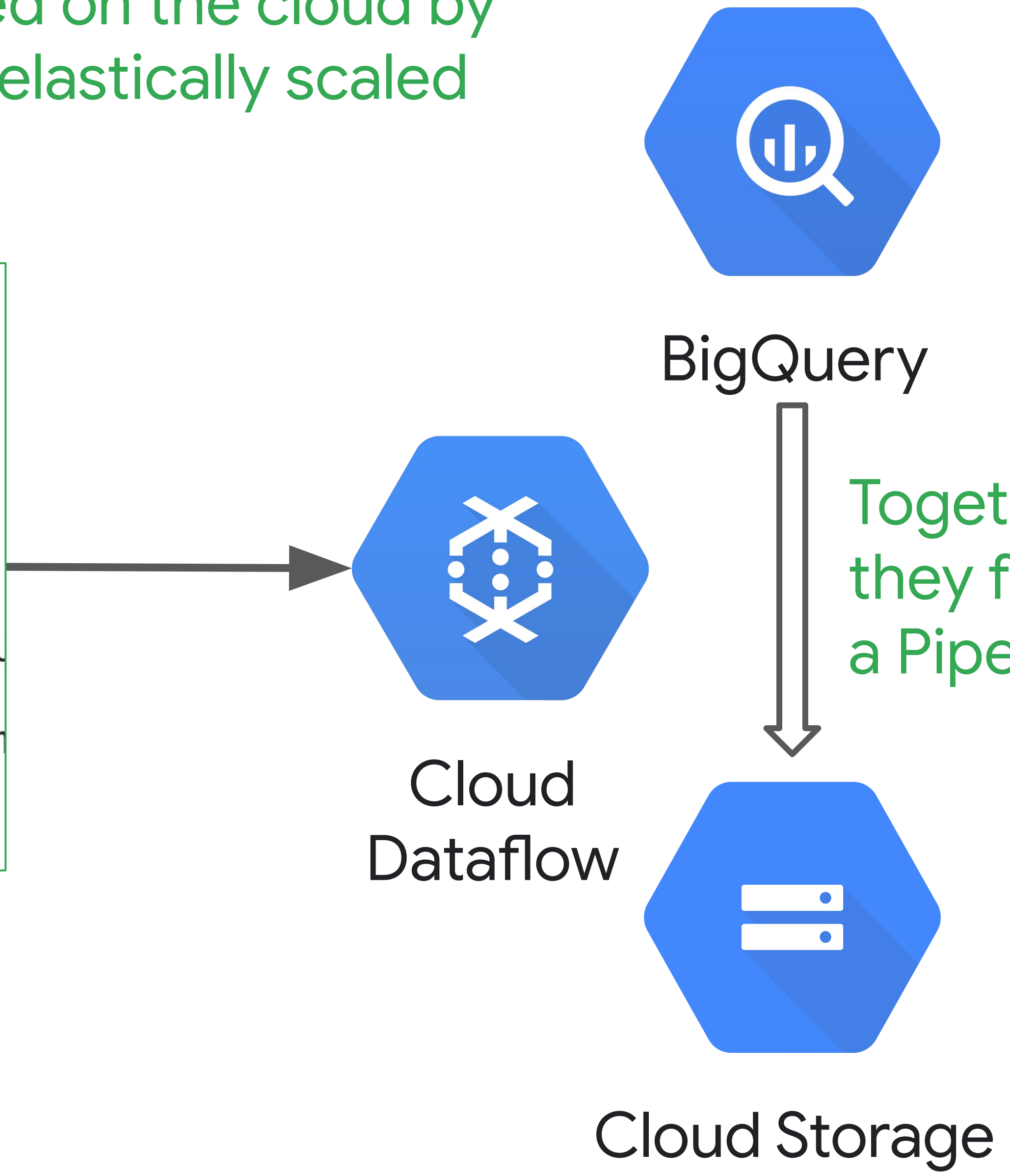
Each of these steps is a Transform

This is a Sink

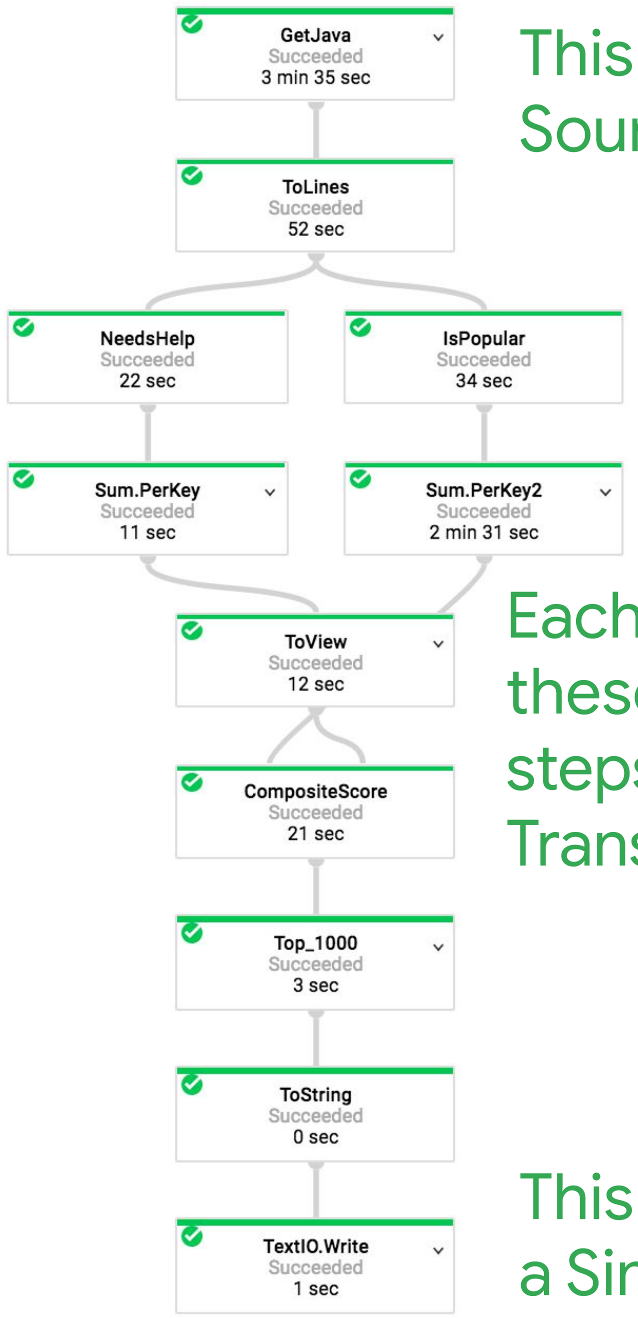
Dataflow terms and concepts

The Pipeline is executed on the cloud by a Runner; each step is elastically scaled

```
def packageHelp(record, keyword):
   count=0
   package_name=''
   if record is not None:
     lines=record.split('\n')
     for line in lines:
       if line.startswith(keyword):
         package_name=line
       if 'FIXME' in line or 'TODO' in l
         count+=1
     packages = (getPackages(package_nam
     for p in packages:
         yield (p,count)
```



Together, they form a Pipeline



This is a Source

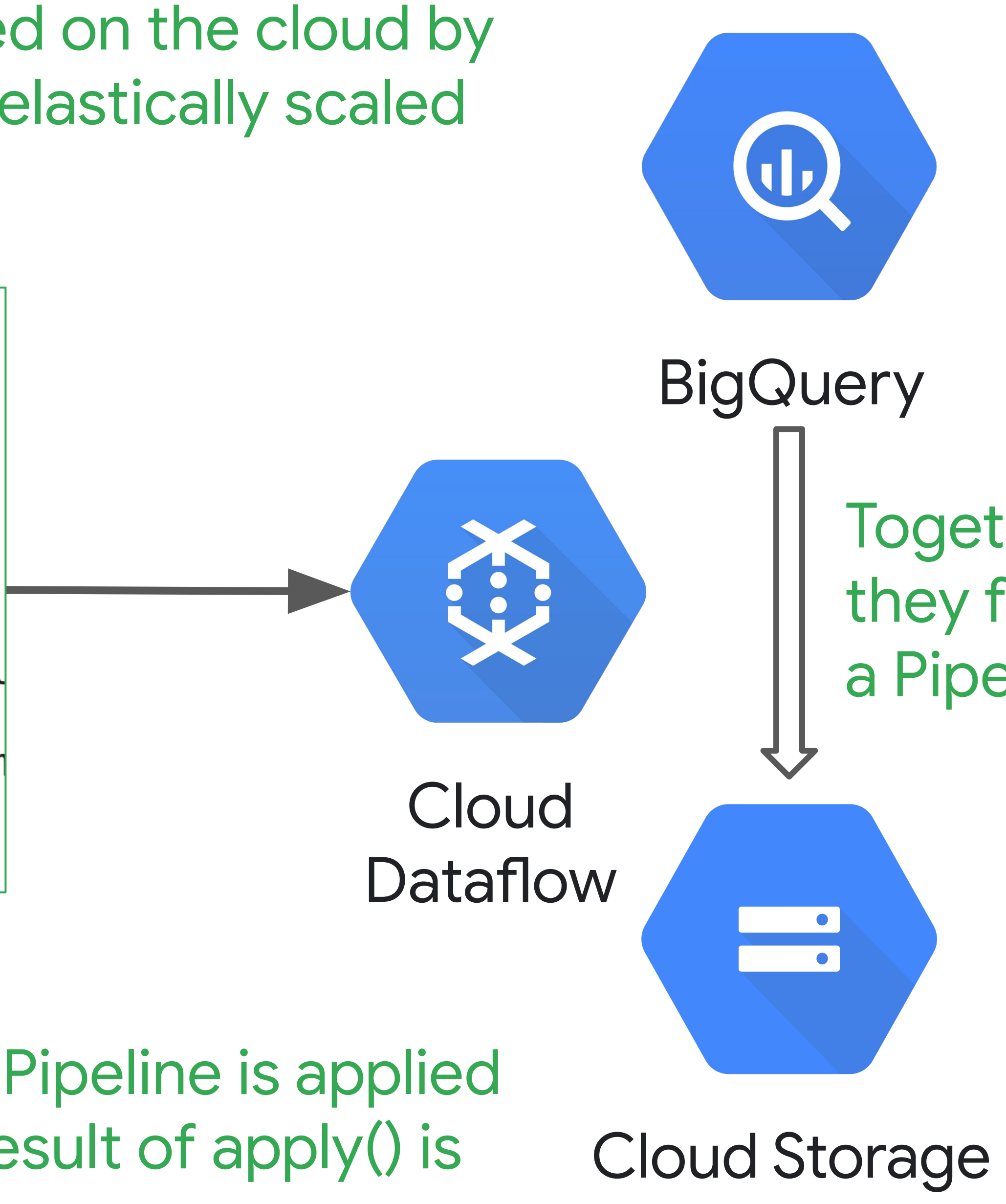
Each of these steps is a Transform

This is a Sink

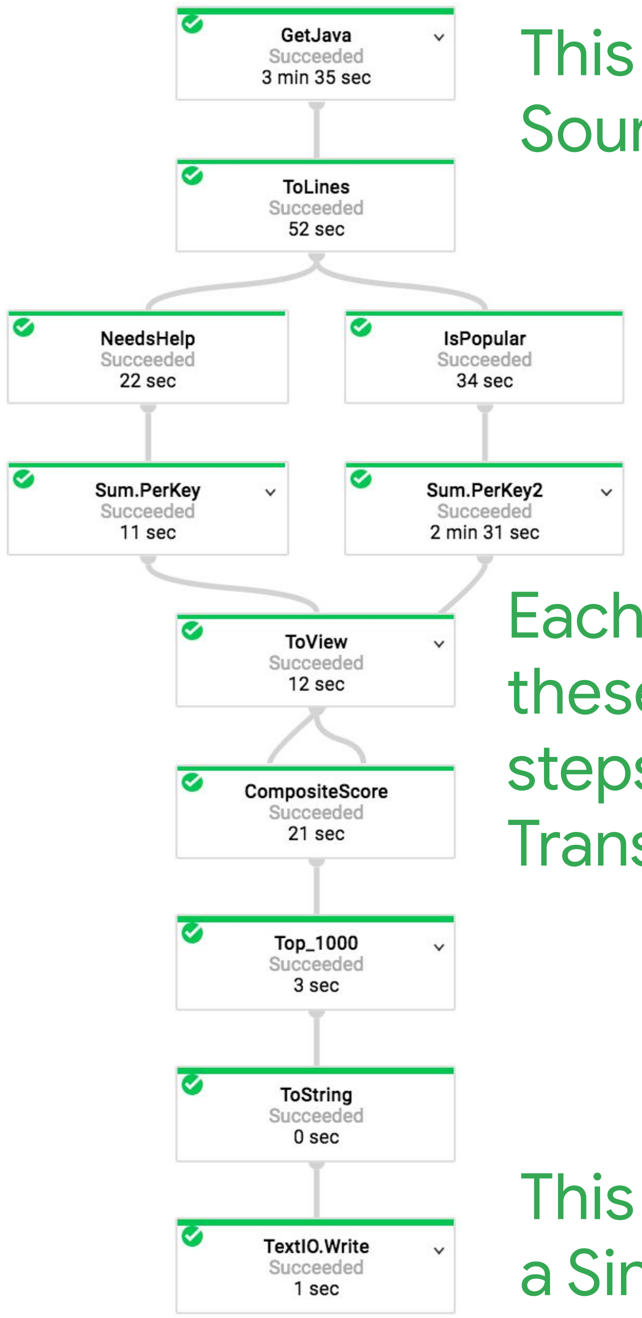
Dataflow terms and concepts

The Pipeline is executed on the cloud by a Runner; each step is elastically scaled

```
def packageHelp(record, keyword):
   count=0
   package_name=''
   if record is not None:
     lines=record.split('\n')
     for line in lines:
       if line.startswith(keyword):
         package_name=line
       if 'FIXME' in line or 'TODO' in l
         count+=1
     packages = (getPackages(package_nam
     for p in packages:
         yield (p,count)
```



Together, they form a Pipeline



This is a Source

Each of these steps is a Transform

This is a Sink

Each Transform of the Pipeline is applied on a PCollection; the result of apply() is another PCollection

A Pipeline is a directed graph of steps

Read in data, transform it, write out

- Can branch, merge, use if-then statements, etc.
- Pythonic syntax

```
import apache_beam as beam
if __name__ == '__main__':

   # create a pipeline parameterized by commandline flags
   p = beam.Pipeline(argv=sys.argv)

   (p
      | 'Read' >> beam.io.ReadFromText('gs://...') # read input
      | 'CountWords' >> beam.FlatMap(lambda line: count_words(line))
      | 'Write' >> beam.io.WriteToText('gs://...') # write output
   )

   p.run() # run the pipeline
```

A Pipeline is a directed graph of steps

Read in data, transform it, write out

- Can branch, merge, use if-then statements, etc.

```
import org.apache.beam.sdk.Pipeline;  // etc.

public static void main(String[] args) {
    // Create a pipeline parameterized by commandline flags.
    Pipeline p = Pipeline.create(PipelineOptionsFactory.fromArgs(args));

    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();
}
```

Python API conceptually similar

Read in data, transform it, write out

- Pythonic syntax

```
import apache_beam as beam

if __name__ == '__main__':
   # create a pipeline parameterized by commandline flags
   p = beam.Pipeline(argv=sys.argv)

   (p
      | beam.io.ReadFromText('gs://...') # read input
      | beam.FlatMap(lambda line: count_words(line)) # do some processing
      | beam.io.WriteToText('gs://...') # write output
   )

   p.run() # run the pipeline
```

Apply Transform to PCollection

Data in a pipeline are represented by PCollection

- Supports parallel processing
- Not an in-memory collection; can be unbounded

```
lines = p | ...
```

- Apply Transform to PCollection; returns PCollection

```
sizes = lines | 'Length' >> beam.Map(lambda line: len(line) )
```

Apply Transform to PCollection

Data in a pipeline are represented by PCollection

- Supports parallel processing
- Not an in-memory collection; can be unbounded

```
PCollection<String>  lines = p.apply(...)   //
```

Apply Transform to PCollection

Apply Transform to PCollection; returns PCollection

```
PCollection<Integer> sizes =
         lines.apply("Length", ParDo.of(new DoFn<String, Integer>() {
                  @ProcessElement
                  public void processElement(ProcessContext c) throws
Exception {
                                                             String line =
c.element();
                                                             c.output(line.length());
}}))
```

Apply Transform to PCollection (Python)

Data in a pipeline are represented by PCollection

Supports parallel processing

Not an in-memory collection; can be unbounded

```
lines = p | ...
```

Apply Transform to PCollection; returns PCollection

```
sizes = lines | 'Length' >> beam.Map(lambda line: len(line) )
```

Ingesting data into a pipeline (Python)

Read data from file system, GCS or BigQuery

- Text formats return String

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

- BigQuery returns a TableRow

```
rows = beam.io.Read(beam.io.BigQuerySource(query='SELECT x, y, z ' \
                'FROM [project:dataset.tablename]', project='PROJECT'))
```

Ingesting data into a pipeline (Java)

Read data from file system, GCS, BigQuery, Pub/Sub

- Text formats return String

```
PCollection<String> lines = p.apply(TextIO.read().from("gs://.../input-*.csv.gz");
```

```
PCollection<String> lines = p.apply(PubsubIO.readStrings().fromTopic(topic));
```

- BigQuery returns a TableRow

```
String javaQuery = "SELECT x, y, z FROM [project:dataset.tablename]";
PCollection<TableRow> javaContent = p.apply(BigQueryIO.read().fromQuery(javaQuery))
```

Can write data out to same formats (Python)

Write data to file system, GCS or BigQuery

```
beam.io.WriteToText(file_path_prefix='/data/output', file_name_suffix='.txt')
```

Can prevent sharding of output (do only if it is small)

```
beam.io.WriteToText(file_path_prefix='/data/output',
file_name_suffix='.txt', num_shards = 1)
```

The output must be a PCollection of Strings before writing out

Can write data out to same formats (Java)

Write data to file system, GCS, BigQuery, Pub/Sub

```
lines.apply(TextIO.write().to("/data/output").withSuffix(".txt"))
```

Can prevent sharding of output (do only if it is small)

```
.apply(TextIO.write().to("/data/output").withSuffix(".csv").withoutSharding()
)
```

May have to transform PCollection<`Integer`>, etc. to PCollection<`String`> before writing out

Executing pipeline (Java)

Simply running main() runs pipeline locally

```
java -classpath ...        com...
```

```
mvn compile -e exec:java -Dexec.mainClass=$MAIN
```

To run on cloud, submit job to Dataflow

```
mvn compile -e exec:java \
      -Dexec.mainClass=$MAIN \
      -Dexec.args="--project=$PROJECT \
      --stagingLocation=gs://$BUCKET/staging/ \
      --tempLocation=gs://$BUCKET/staging/ \
      --runner=DataflowRunner"
```

Executing pipeline (Python)

Simply running main() runs pipeline locally

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

To run on cloud, specify cloud parameters, and submit the job to Dataflow

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

Executing pipeline (Python)

Simply running main() runs pipeline locally

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

To run on cloud, specify cloud parameters

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

Lab

A simple Dataflow pipeline

Carl Osipov

Lab: A simple Dataflow pipeline (Java/Python)

In this lab, you will learn how to:

1. Set up a Dataflow project
2. Write a simple pipeline
3. Execute the pipeline on the local machine
4. Execute the pipeline on the cloud

```
p //
                .apply("GetJava", TextIO.
                .apply("Grep", ParDo.of(n
                        @ProcessElement
                        public void proce
                                String li
                                if (line.
                                        c
                                }
                        }
                })) //
                .apply(TextIO.write().to(
```

MapReduce approach splits Big Data so that each compute node processes data local to it

MapReduce

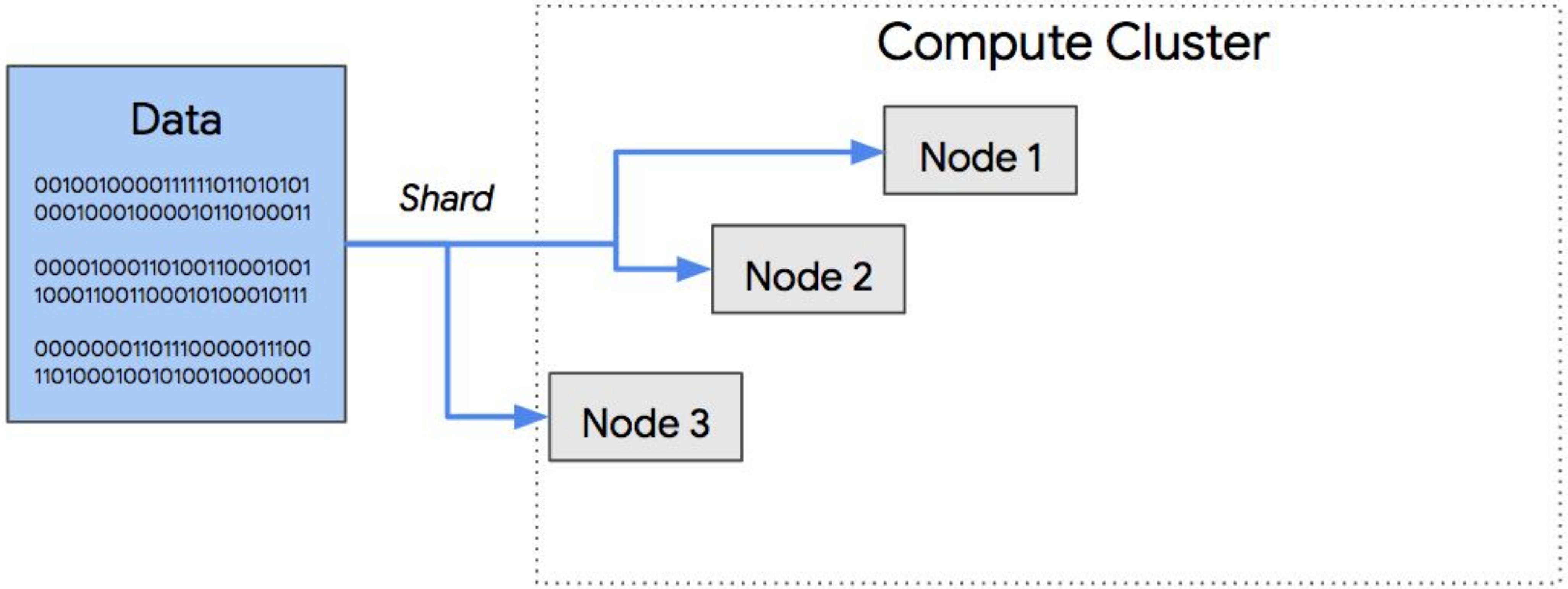
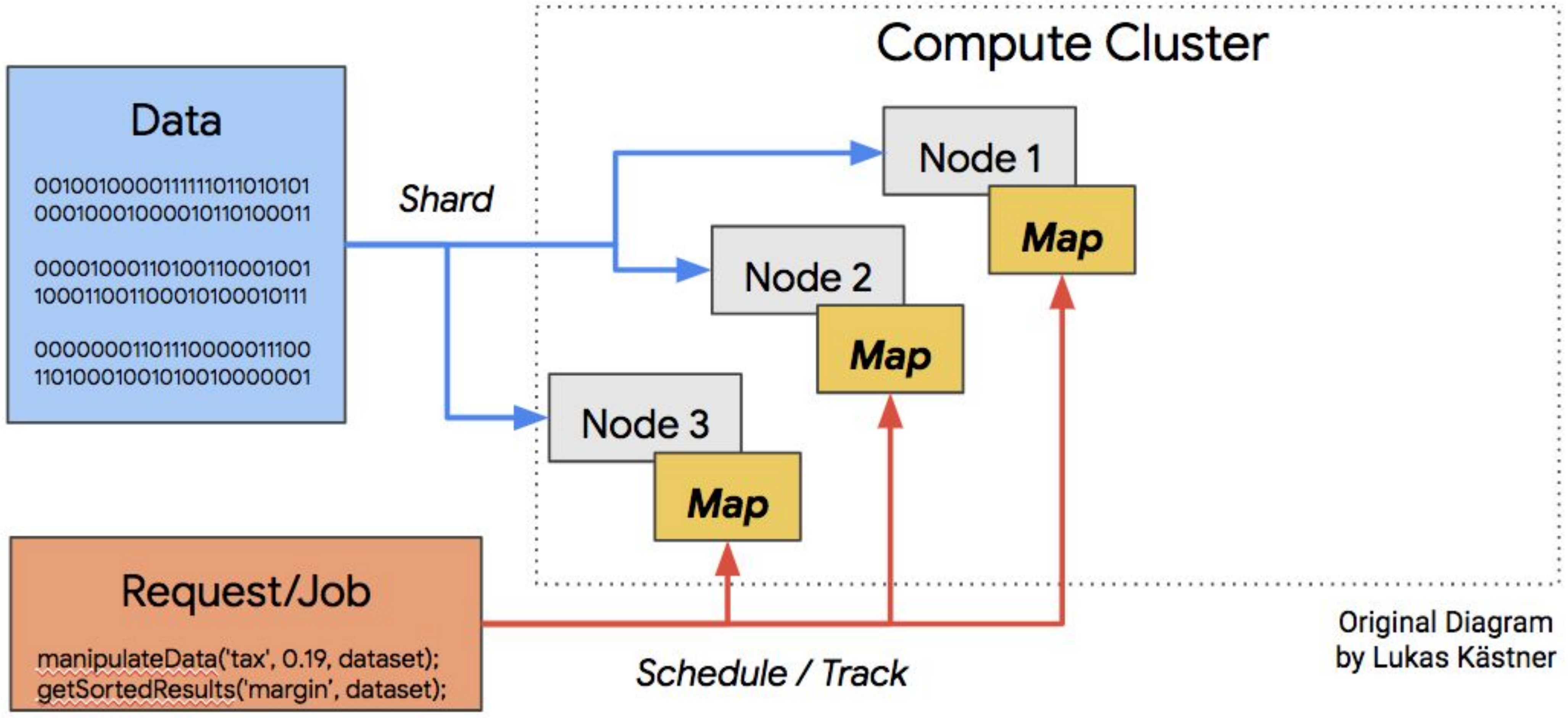


Diagram by Lukas Kästner

2002 2004 2006 2008 2010 2012 2014 2016

MapReduce approach splits Big Data so that each compute node processes data local to it

MapReduce

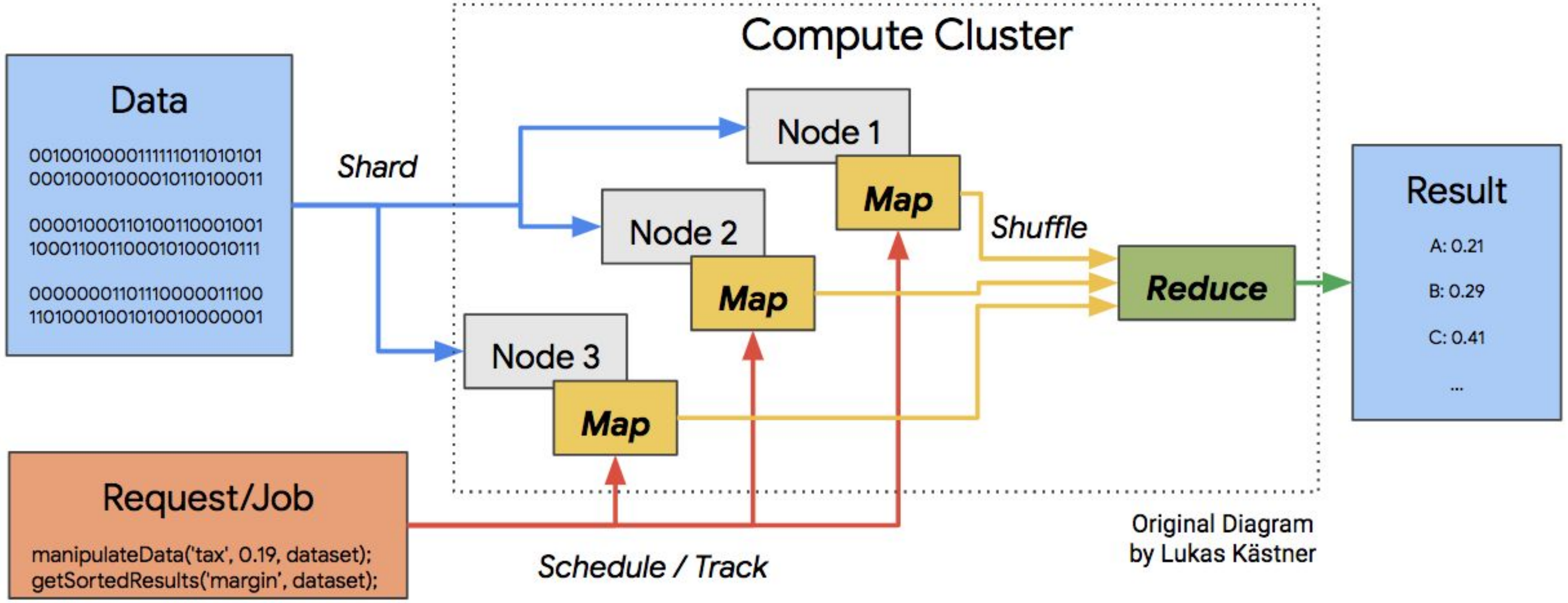


Original Diagram
by Lukas Kästner

2002 2004 2006 2008 2010 2012 2014 2016

MapReduce approach splits Big Data so that each compute node processes data local to it

MapReduce



2002 2004 2006 2008 2010 2012 2014 2016

ParDo allows for parallel processing

ParDo acts on one item at a time (like a Map in MapReduce)

- Multiple instances of class on many machines
- Should not contain any state

Useful for:

- Filtering (choosing which inputs to emit)
- Extracting parts of an input (e.g., fields of TableRow)
- Converting one Java type to another
- Calculating values from different parts of inputs

Python: Map vs. FlatMap

Use Map for 1:1 relationship between input & output

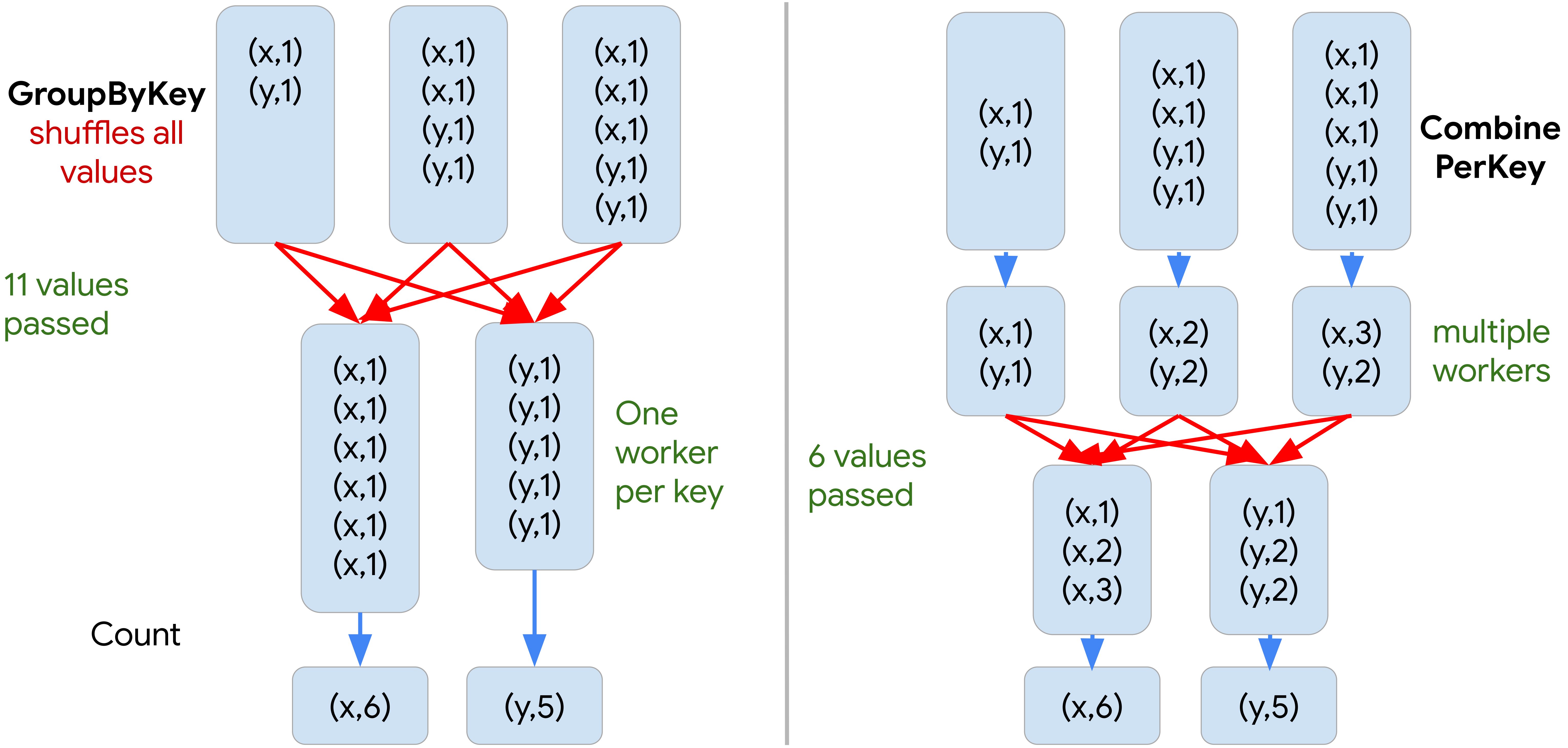
```
'WordLengths' >> beam.Map( lambda word: (word, len(word)) )
```

FlatMap for non 1:1 relationships, usually with generator

```
def vowels(word):
  for ch in word:
    if ch in ['a','e','i','o','u']:
      yield ch

'WordVowels' >> beam.FlatMap( lambda word: vowels(word) )
```

Java: Use apply(ParDo) for both cases

Reduce with GroupByKey or Combine.PerKey



GroupBy operation is akin to shuffle

In Dataflow, shuffle explicitly with a GroupByKey

- Create a Key-Value pair in a ParDo
- Then group by the key

```
cityAndZipcodes = p

    | beam.Map(lambda address: (address[1], address[3]) )
                                 CITY        ZIPCODE
  | beam.GroupByKey()
```

Combine.PerKey lets you aggregate

Can be applied to a PCollection of values:

```
totalAmount = salesAmounts | Combine.globally(sum)
```

And also to a grouped Key-Value pair:

```
totalSalesPerPerson = salesRecords | Combine.perKey(sum)
```

Many built-in functions: Sum, Mean, etc.

Lab

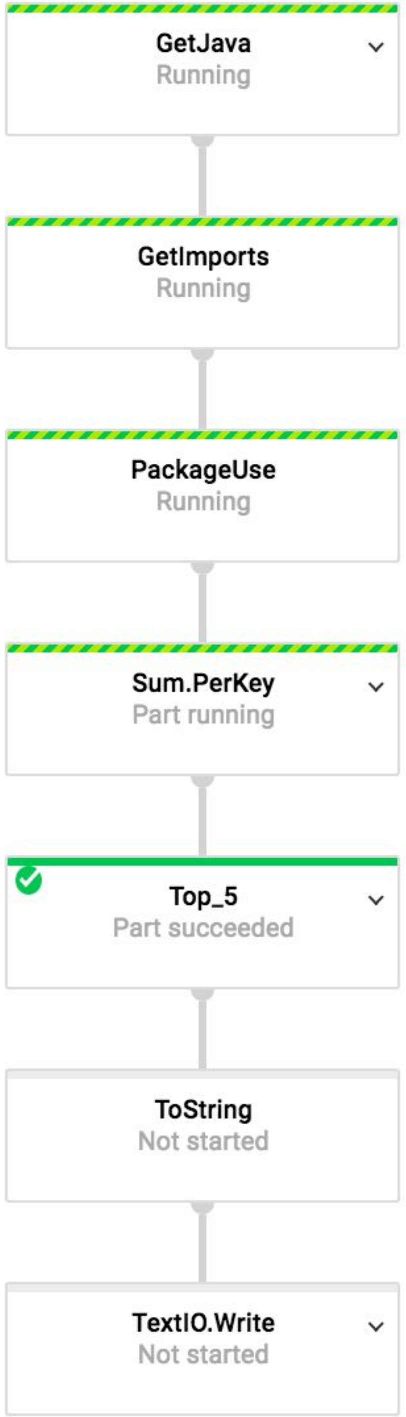
MapReduce in Dataflow

Carl Osipov

Lab: MapReduce in Dataflow (Java/Python)

In this lab, you will learn how to:

- Specify and use command-line options
- Carry out Map and Reduce operations



Use windows to specify how to aggregate unbounded collections

```
| 'Window' >> beam.WindowInto(
                 SlidingWindows(size = 120,
                                period = 30))
```

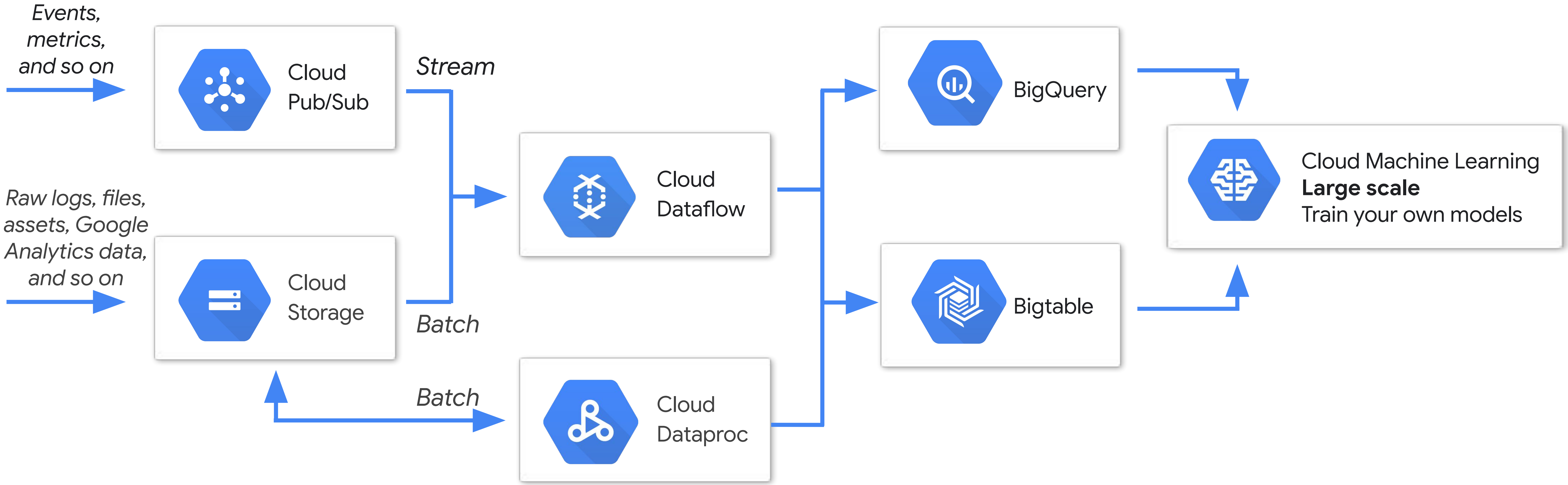
Subsequent Groups, aggregations, etc. are computed only within time window

Use windows to specify how to aggregate unbounded collections

```
.apply("window", Window.into(SlidingWindows//

    .of(Duration.standardMinutes(2))//

    .every(Duration.standardSeconds(30)))) //
```

Subsequent Groups, aggregations, etc.
are computed only within time window

In the Google Cloud reference architecture, preprocessing is repeatable between training and prediction because of Dataflow

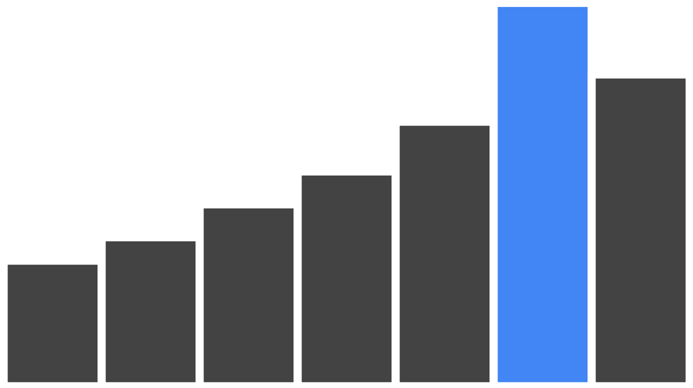


Google Cloud

Preprocessing with Cloud Dataprep

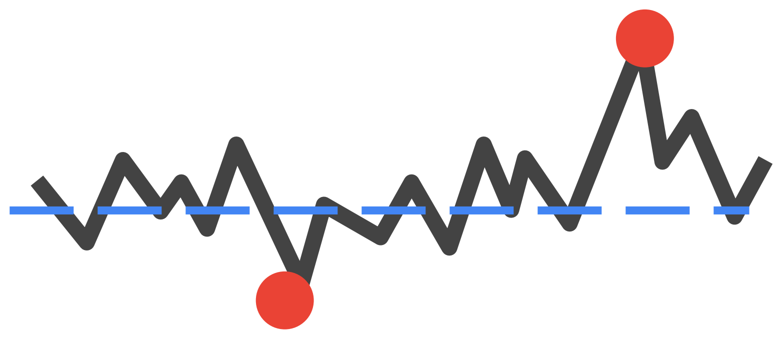
Carl Osipov

Exploring and knowing your data is essential



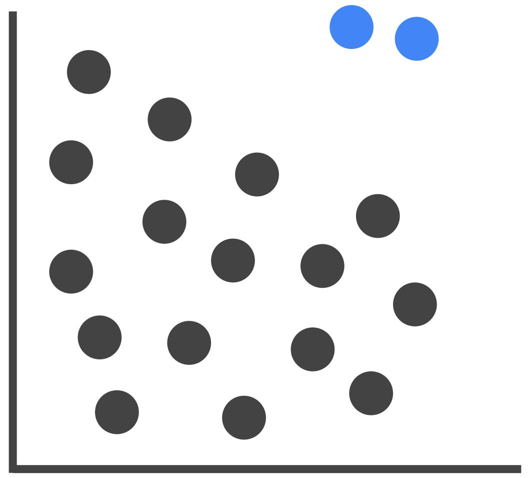
Explore and Visualize
Common Values

Exploring and knowing your data is essential



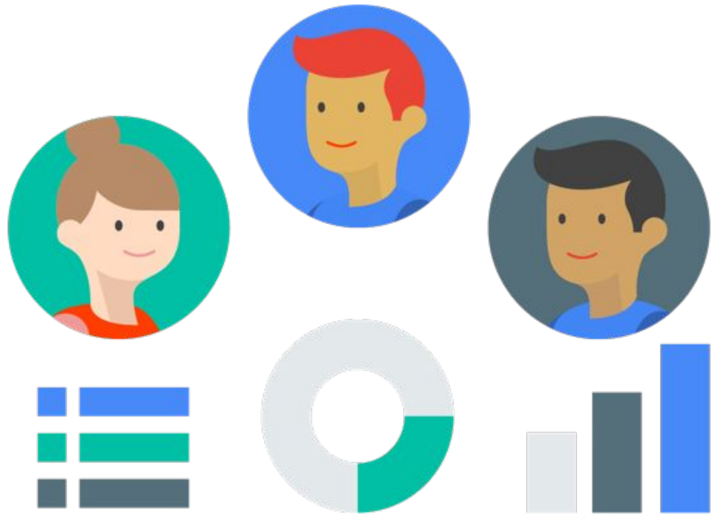
Analyze Key Statistics (min, max, avg, std dev)

Exploring and knowing your data is essential



Explore the distributions

Exploring and knowing your data is essential



Collaborate with
Domain Experts

There are two general approaches to designing preprocessing

There are two general approaches to designing preprocessing

1. Explore in Cloud Datalab

2. Write code in BigQuery / Dataflow / TensorFlow to transform data

Cloud Dataflow

TensorFlow

Cloud Datalab

BigQuery

There are two general approaches to designing preprocessing

1. Explore in Cloud Datalab
2. Write code in BigQuery / Dataflow / TensorFlow to transform data

Cloud Dataflow

TensorFlow

Cloud Datalab

BigQuery

1. Explore in Cloud Dataprep
2. Design Recipe in UI to Preprocess Data
3. Apply generated Dataflow transformations to all data
4. Reuse Dataflow transformation in real-time pipeline

Cloud Dataprep

Cloud Dataflow

There are two general approaches to designing preprocessing

1. Explore in Cloud Datalab
2. Write code in BigQuery / Dataflow / TensorFlow to transform data

Cloud Dataflow

TensorFlow

Cloud Datalab

BigQuery

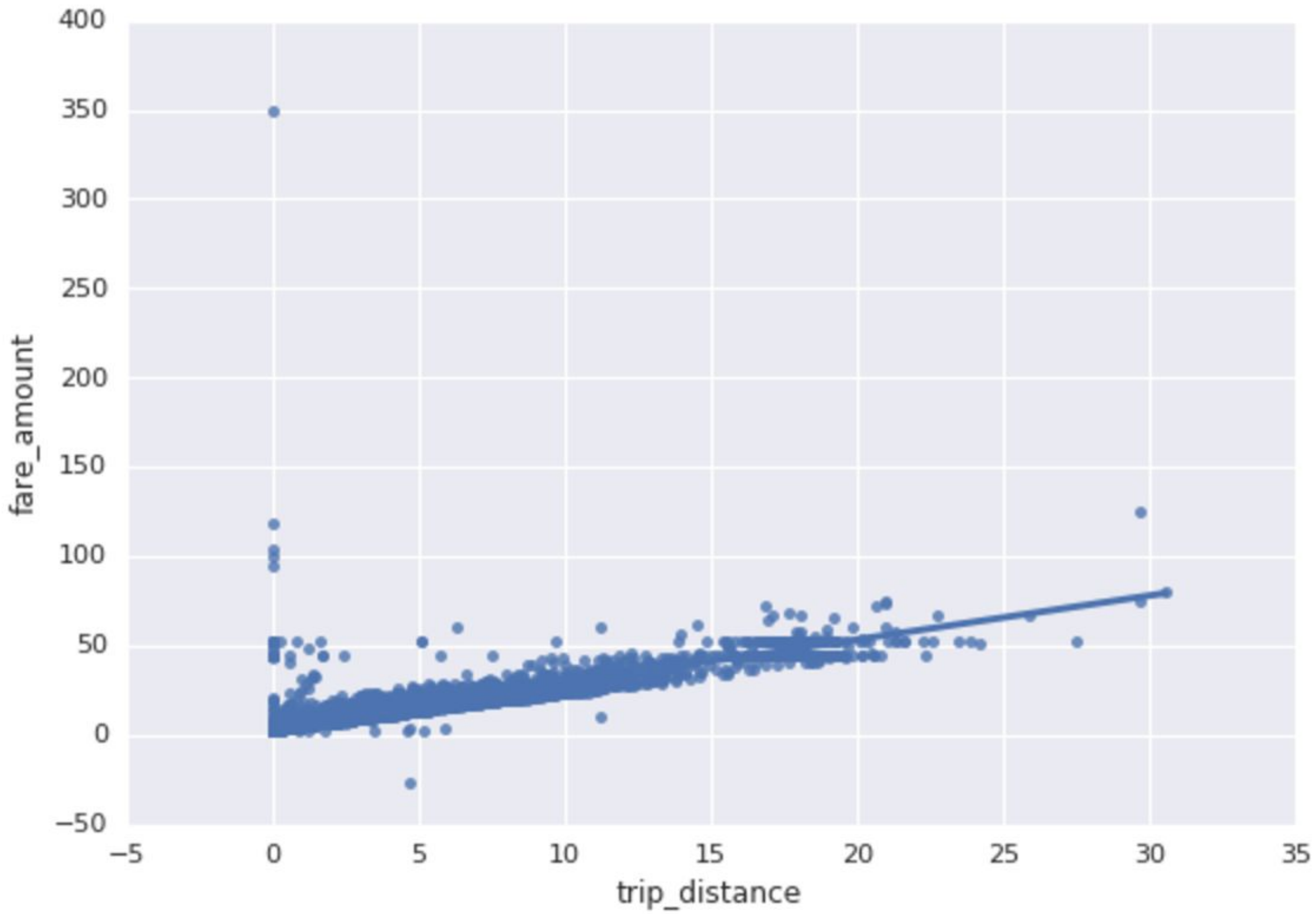
1. Explore in Cloud Dataprep
2. Design Recipe in UI to Preprocess Data
3. Apply generated Dataflow transformations to all data
4. Reuse Dataflow transformation in real-time pipeline

Cloud Dataprep

Cloud Dataflow

Example of exploring in Datalab: is there something wrong?

```
ax = sns.regplot(x="trip_distance", y="fare_amount", ci=None, truncate=True, data=trips)
```



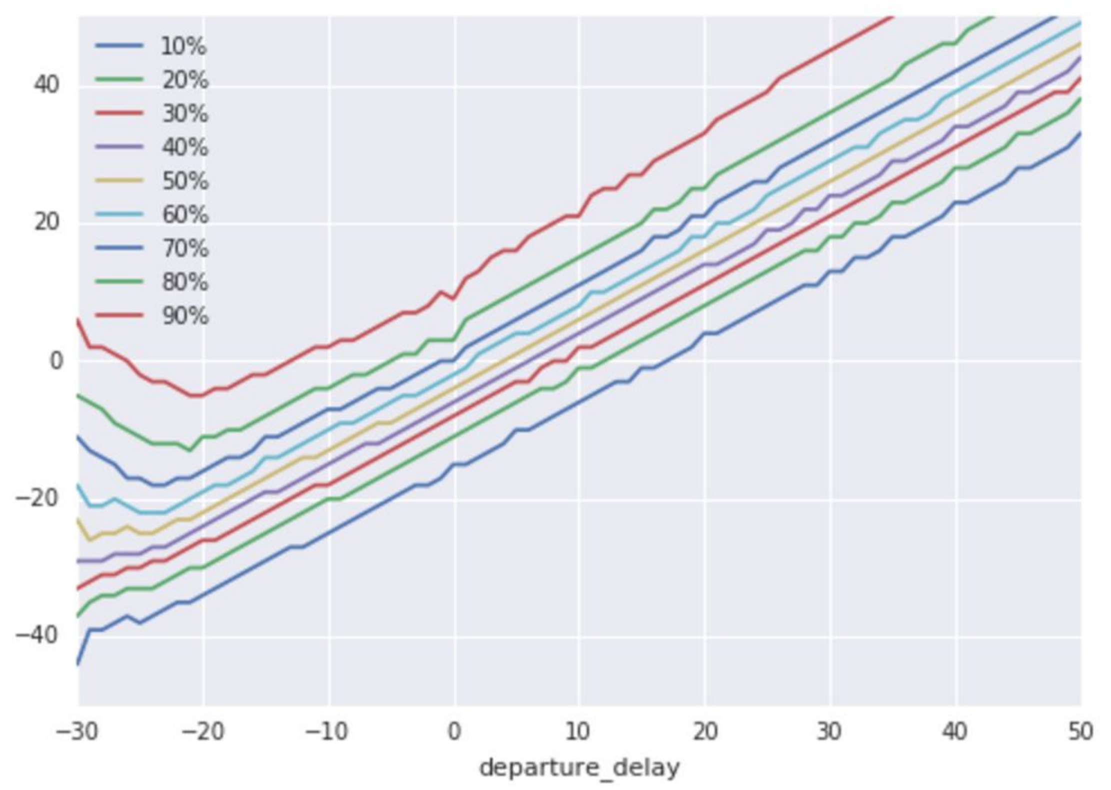
Scatterplots and in-memory pandas dataframes don’t scale

… is there a better way?

Best to aggregate in BigQuery and plot in Datalab

```
query="""
SELECT
  departure_delay,
  COUNT(1) AS num_flights,
  APPROX_QUANTILES(arrival_delay, 10) AS arrival_delay_deciles
FROM
  `bigquery-samples.airline_ontime_data.flights`
GROUP BY
  departure_delay
HAVING
  num_flights > 100
ORDER BY
  departure_delay ASC
"""

import google.datalab.bigquery as bq
df = bq.Query(query).execute().result().to_dataframe()
df.head()
```

```
without_extremes = df.drop(['0%', '100%'], 1)
without_extremes.plot(x='departure_delay', xlim=(-30,50), ylim=(-50,50));
```



Write DataFlow code to do any transformations

Using Cloud Dataprep for exploring and preprocessing data

1. Explore in Cloud Datalab
2. Write code in BigQuery / Dataflow / TensorFlow to transform data

Cloud Dataflow

TensorFlow

Cloud Datalab

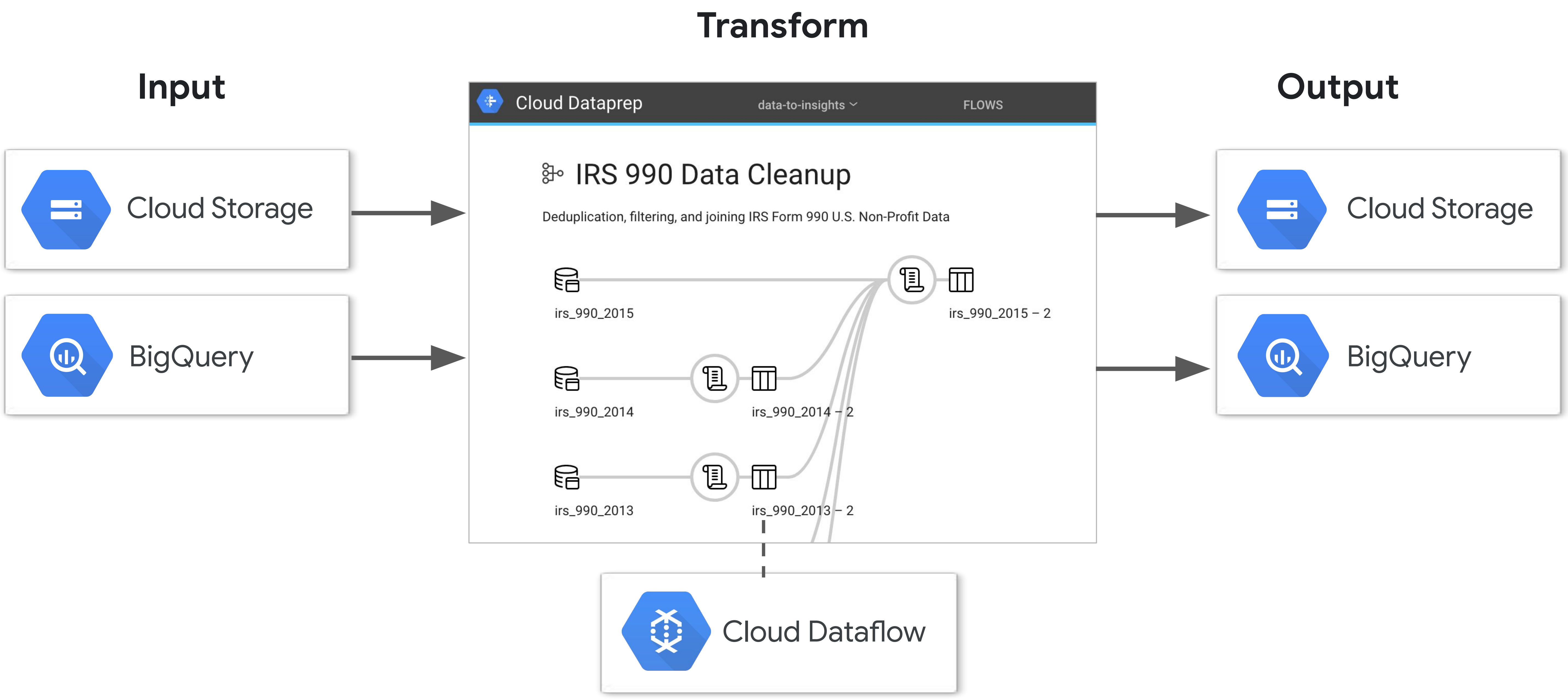
BigQuery

1. Explore in Cloud Dataprep
2. Design Recipe in UI to Preprocess Data
3. Apply generated Dataflow transformations to all data
4. Reuse Dataflow transformation in real-time pipeline

Cloud Dataprep

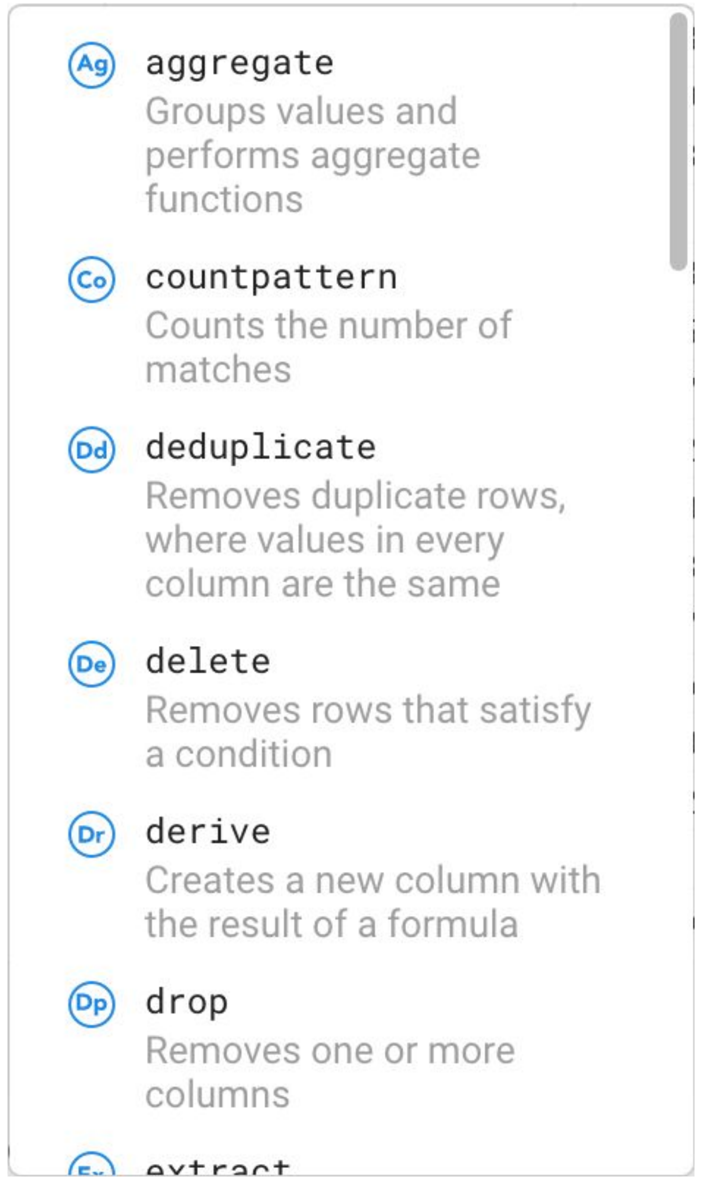
Cloud Dataflow

Cloud Dataprep supports the full preprocessing lifecycle



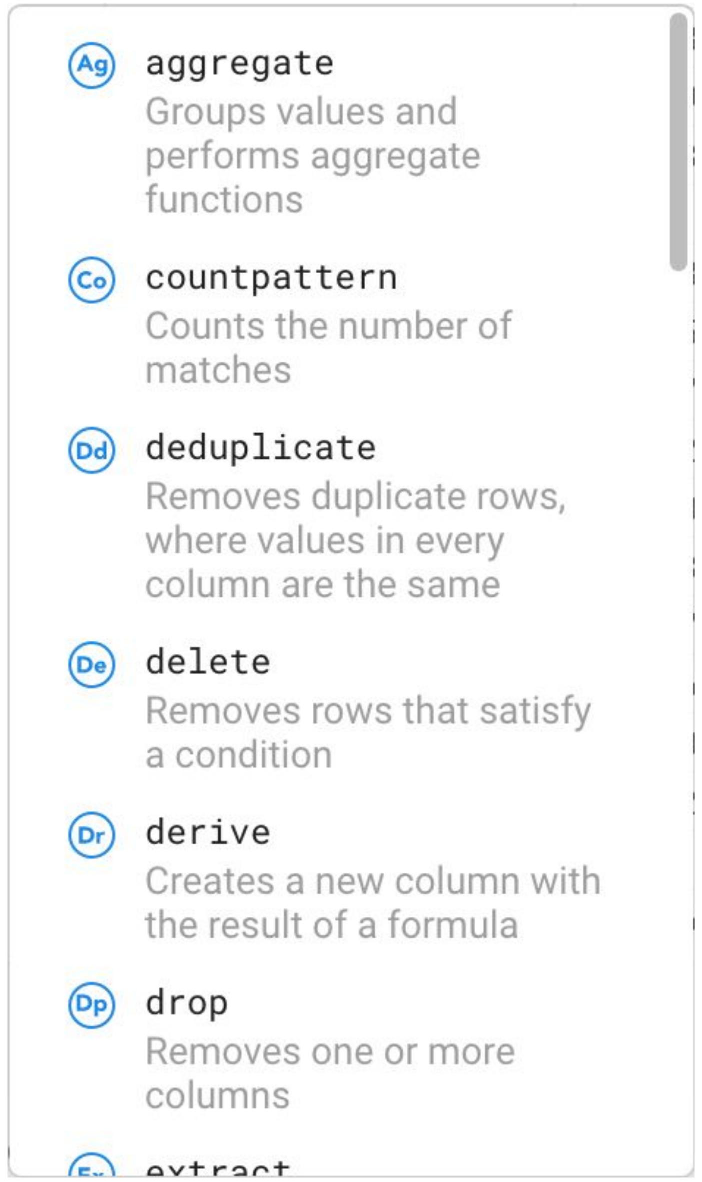
Cloud Dataprep wranglers write beam code in Dataflow

Build Recipes in Cloud Dataprep UI

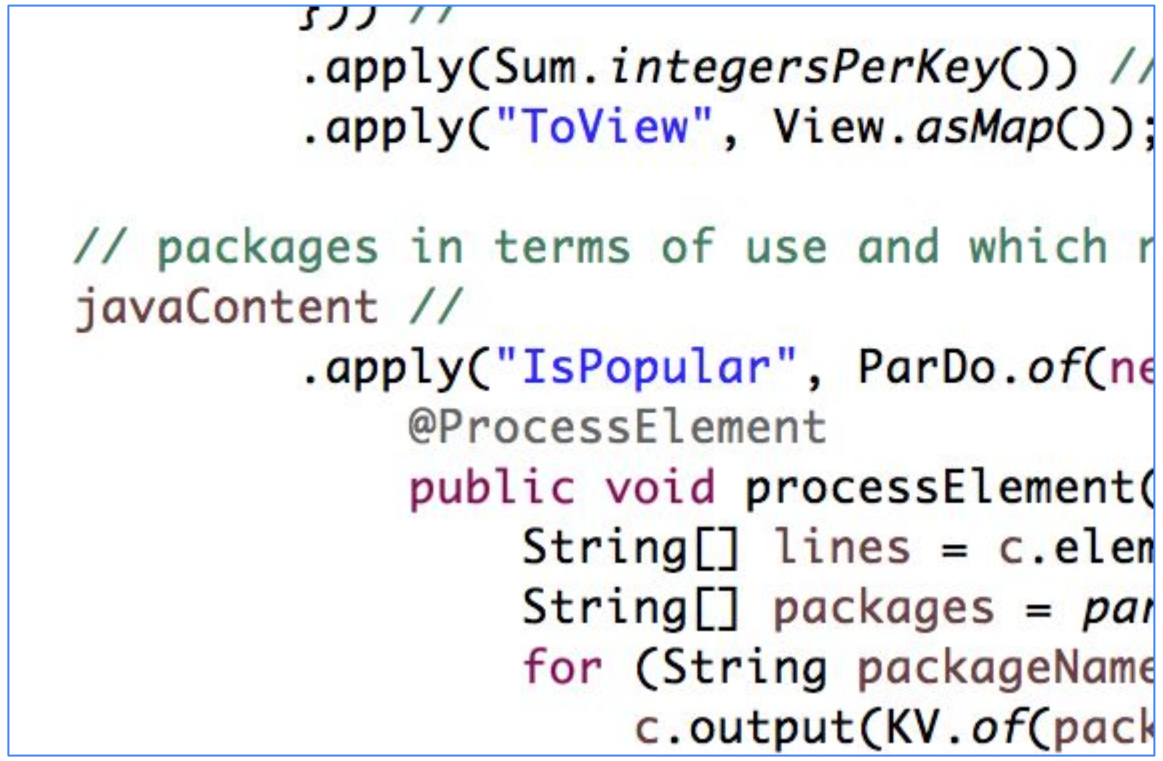


Cloud Dataprep wranglers write beam code in Dataflow

Build Recipes in Cloud Dataprep UI



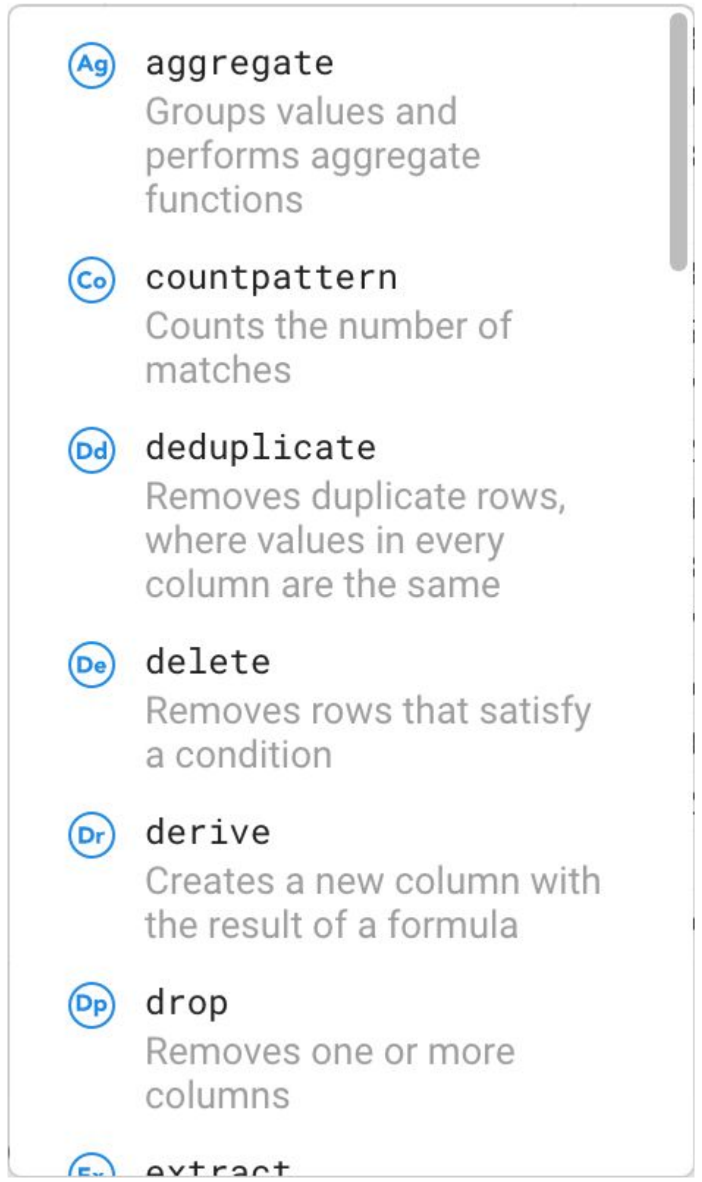
Dataprep Converts Recipes to Beam



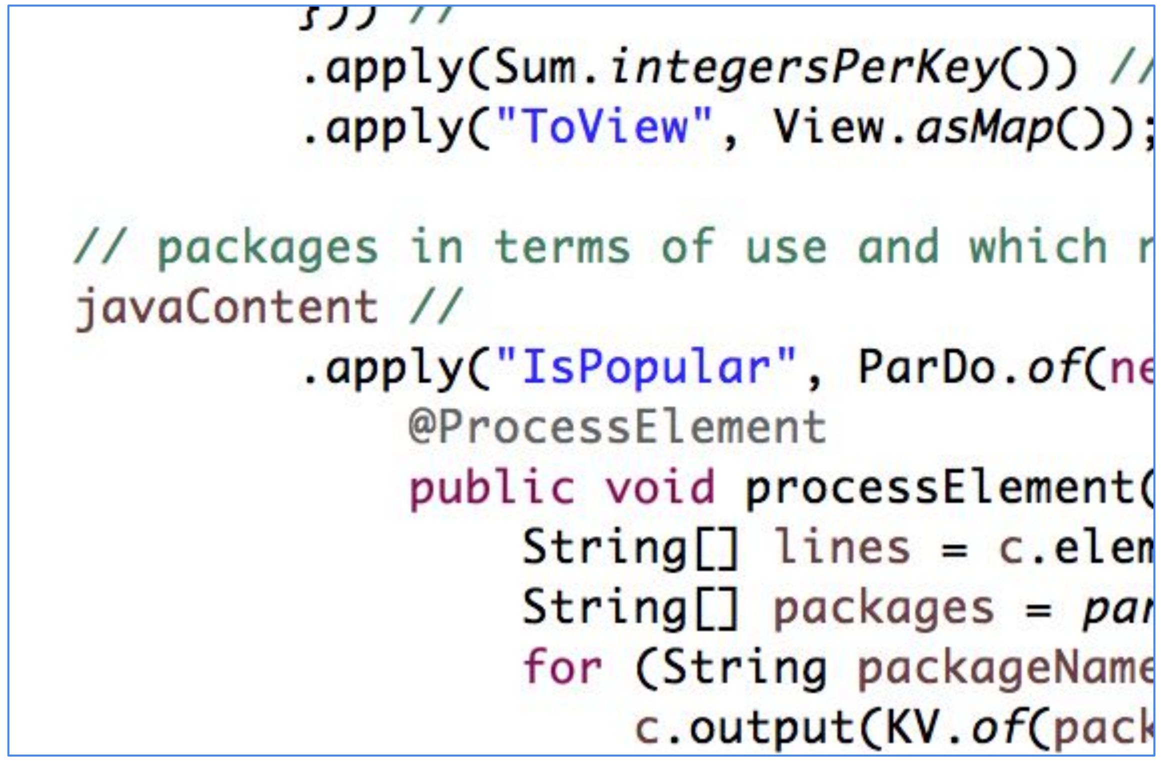
```
        .apply(Sum.integersPerKey()) //
        .apply("ToView", View.asMap());

// packages in terms of use and which r
javaContent //
        .apply("IsPopular", ParDo.of(ne
            @ProcessElement
            public void processElement(
                String[] lines = c.elem
                String[] packages = par
                for (String packageName
                    c.output(KV.of(pack
```

Cloud Dataprep wranglers write beam code in Dataflow

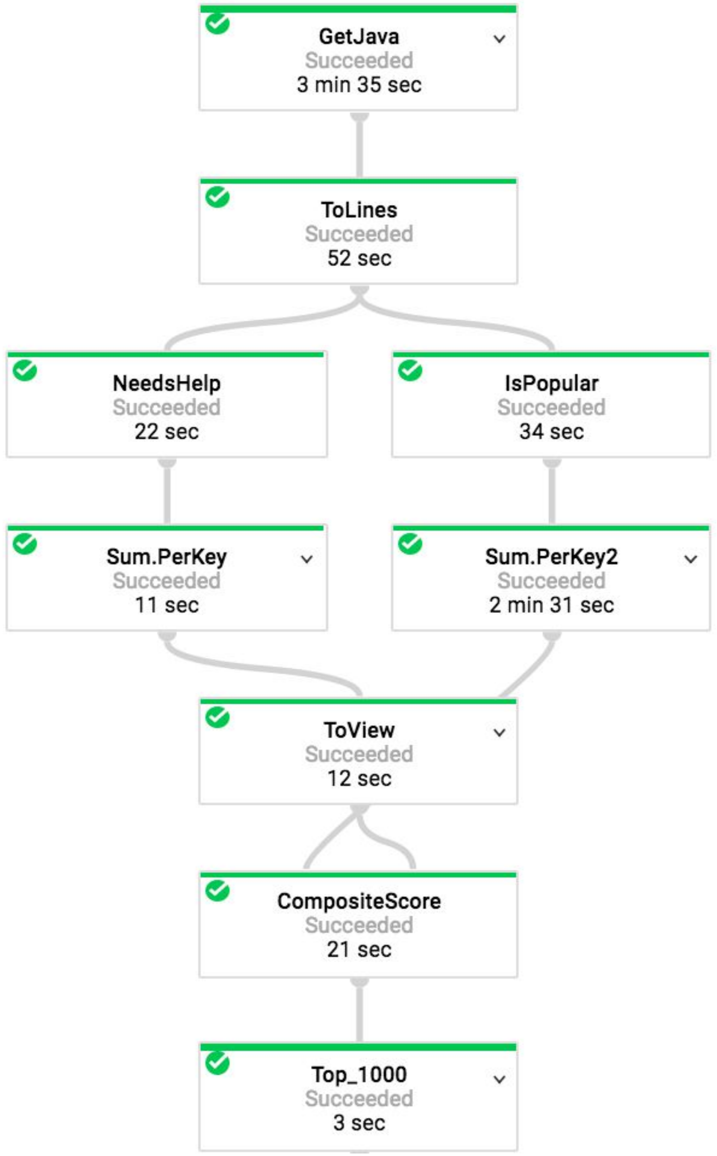
Build Recipes in Cloud Dataprep UI



Dataprep Converts Recipes to Beam



Dataprep Runs a Dataflow Job



Wide array of transformation wranglers available

Data Ingestion (Upload, GCS, BigQuery)

Data Cleansing

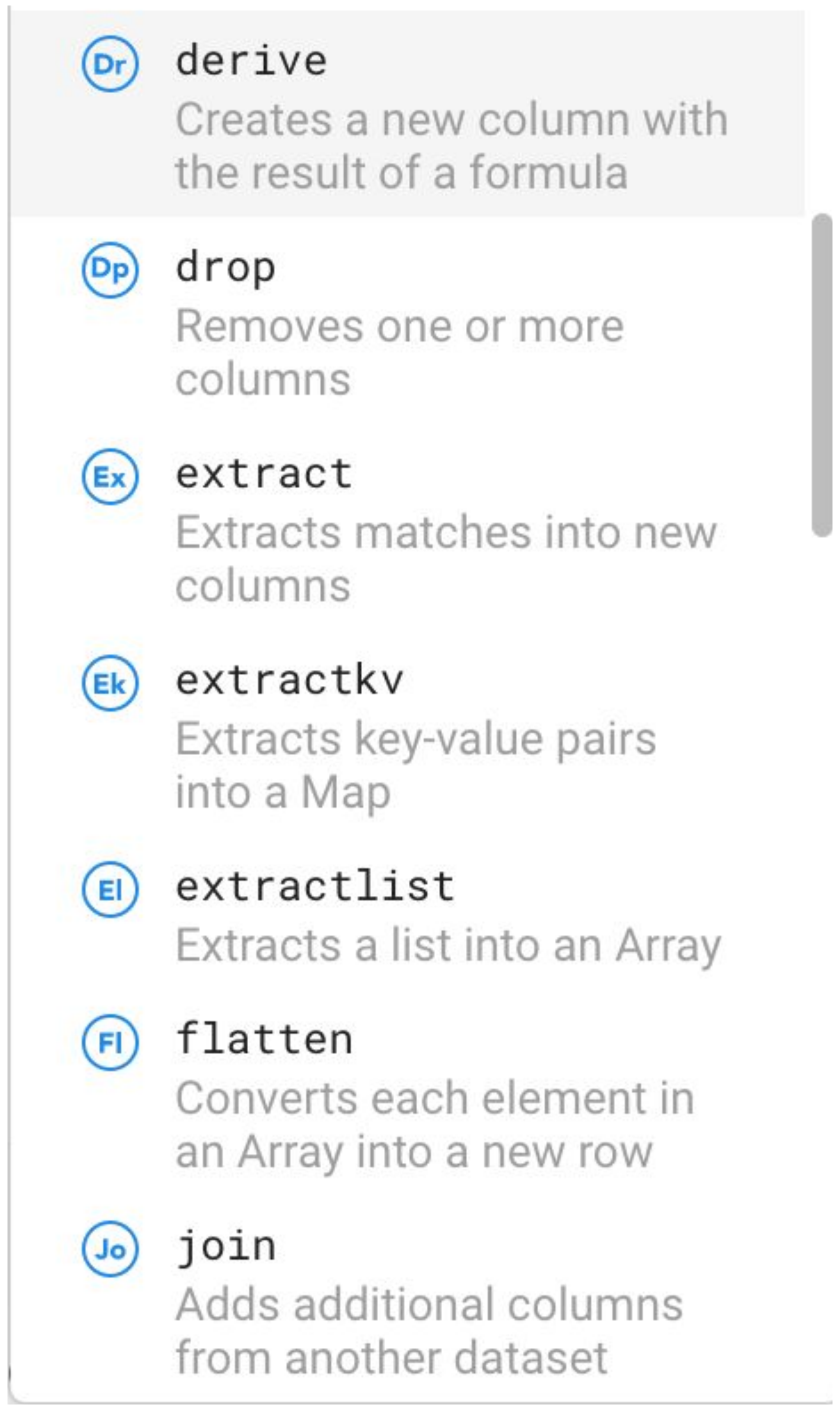
Aggregations

Joins, Unions

Transformations

Type Conversions

Dataprep Wranglers

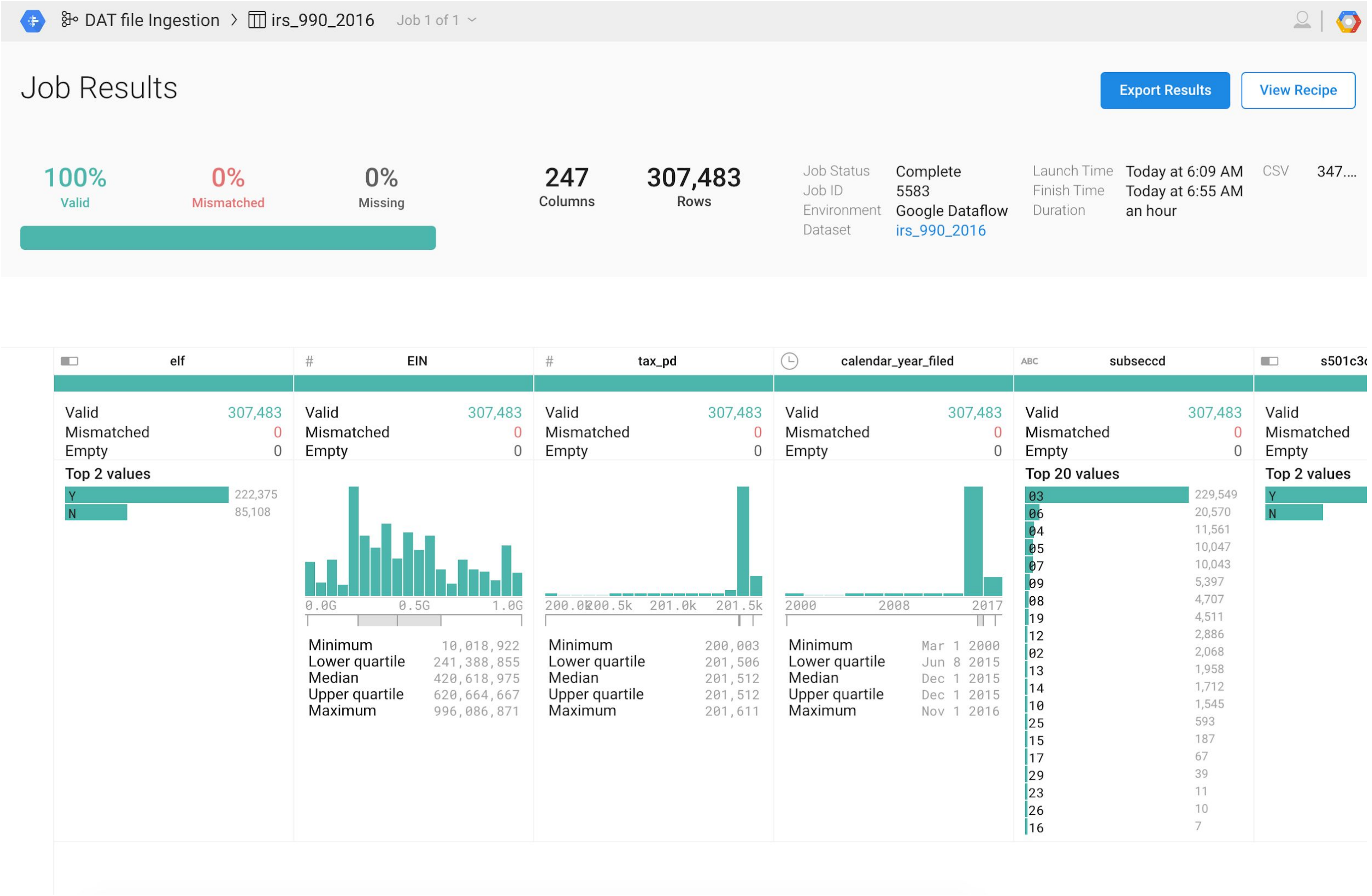


Monitor Dataprep jobs and output results to BigQuery or GCS

Track completed and ongoing jobs

See the data quality metrics for transformed datasets

View histograms with summary statistics for each field



Lab

Computing Time-Windowed Features in Cloud Dataprep

Carl Osipov

Lab: Computing Time-Windowed Features in Cloud Dataprep

In this lab, you will learn how to:

- Build a new Flow using Cloud Dataprep
- Create and chain transformation steps with recipes
- Running Cloud Dataprep jobs

cloud.google.com

Google Cloud

Feature Engineering

Carl Osipov

Preprocessing and feature creation

Carl Osipov