Dataflow job graphs

Read time: 6 minutes

When you select a specific Dataflow job, the monitoring interface provides a graphical representation of your job: the *job graph*. The job graph page in the console also provides a job summary, a job log, and information about each step in the pipeline.

A pipeline's job graph represents each transform in the pipeline as a box. Each box contains the transform name and information about the job status, which includes the following:

- **Running**: the step is running
- **Queued**: the step in a FlexRS job is queued
- **Succeeded**: the step finished successfully
- **Stopped**: the step stopped because the job stopped
- **Unknown**: the step failed to report status
- **Failed**: the step failed to complete

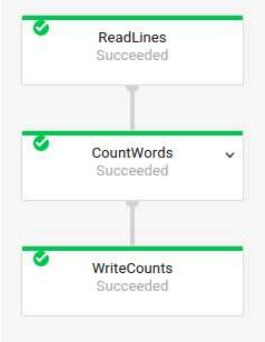
By default, the job graph page displays the **Graph view**. To view your job graph as a table, in **Job steps view**, select **Table view**. The table view contains the same information in a different format. The table view is helpful in the following scenarios:

- Your job has many stages, making the job graph difficult to navigate.
- You want to sort the job steps by a specific property. For example, you can sort the table by wall time to identify slow steps.

Basic job graph

Pipeline Code: **Job graph:**

```
  // Read the lines of the input text.
  p.apply("ReadLines",
TextIO.read().from(options.getInputFile()))
     // Count the words.
     .apply(new CountWords())
     // Write the formatted word counts to output.
     .apply("WriteCounts",
TextIO.write().to(options.getOutput()));
```




```
(
    pipeline
    # Read the lines of the input text.
    | 'ReadLines' >>
beam.io.ReadFromText(args.input_file)
    # Count the words.
    | CountWords()
    # Write the formatted word counts to output.
    | 'WriteCounts' >>
beam.io.WriteToText(args.output_path))
```

```
  // Create the pipeline.
  p := beam.NewPipeline()
    s := p.Root()
  // Read the lines of the input text.
  lines := textio.Read(s, *input)
  // Count the words.
  counted := beam.ParDo(s, CountWords, lines)
  // Write the formatted word counts to output.
  textio.Write(s, *output, formatted)
```

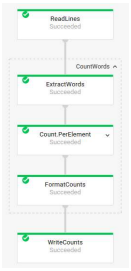
Figure 1: The pipeline code for a WordCount pipeline shown with the resulting execution graph in the Dataflow monitoring interface.

Composite transforms

In the job graph, composite transforms, transforms that contain multiple nested sub-transforms, are expandable. Expandable composite transforms are marked with an arrow in

the graph. To expand the transform and view the sub-transforms, click the arrow.

Pipeline Code: **Job graph:**




```
  // 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());return
wordCounts;
    }
  }
```

```
# The CountWords Composite Transform inside the
WordCount pipeline.
@beam.ptransform_fn
def CountWords(pcoll):
  return (
      pcoll
      # Convert lines of text into individual words.
      | 'ExtractWords' >> beam.ParDo(ExtractWordsFn())
      # Count the number of times each word occurs.
      | beam.combiners.Count.PerElement()
      # Format each word and count into a printable
string.
      | 'FormatCounts' >>
beam.ParDo(FormatCountsFn()))
```

```
  // The CountWords Composite Transform inside the
WordCount pipeline.
  func CountWords(s beam.Scope, lines
```

```
beam.PCollection) beam.PCollection {
    s = s.Scope("CountWords")// Convert lines of text
into individual words.
    col := beam.ParDo(s, &extractFn{SmallWordLength:
*smallWordLength}, lines)// Count the number of times
each word occurs.
    return stats.Count(s, col)
  }
```

Figure 2: The pipeline code for the sub-steps of the CountWords transform. Shown with the job graph expanded for the entire pipeline.

In your pipeline code, you might use the following code to invoke your composite transform:

```
result = transform.apply(input);
```

Composite transforms invoked in this manner omit the expected nesting and might appear expanded in the Dataflow monitoring interface. Your pipeline might also generate warnings or errors about stable unique names at pipeline execution time.

To avoid these issues, invoke your transforms by using the recommended format:

```
result = input.apply(transform);
```

Transform names

Dataflow has a few different ways to obtain the transform name that's shown in the monitoring job graph. Transform names are used in publicly-visible places, including the Dataflow monitoring interface, log files, and debugging tools. Don't use transform names that include personally identifiable information, such as usernames or organization names.

- **Dataflow can use a name that you assign** when you apply your transform. The first argument you supply to the `apply` method is your transform name.

- **Dataflow can infer the transform name**, either from the class name, if you build a custom transform, or the name of your `DoFn` function object, if you use a core transform such as `ParDo`.

- **Dataflow can use a name that you assign** when you apply your transform. You can set the transform name by specifying the transform's `label` argument.
- **Dataflow can infer the transform name**, either from the class name, if you build a custom transform, or the name of your `DoFn` function object, if you use a core transform such as `ParDo`.

- **Dataflow can use a name that you assign** when you apply your transform. You can set the transform name by specifying the `Scope`.
- **Dataflow can infer the transform name**, either from the struct name if you're using a structural `DoFn` or from the function name if you're using a functional `DoFn`.

Understand the metrics

This section provides details about the metrics associated with the job graph.

Wall time

When you click a step, the **Wall time** metric is displayed in the **Step info** panel. Wall time provides the total approximate time spent across all threads in all workers on the following actions:

- Initializing the step
- Processing data
- Shuffling data
- Ending the step

For composite steps, wall time tells you the sum of time spent in the component steps. This estimate can help you identify slow steps and diagnose which part of your pipeline is taking more time than required.

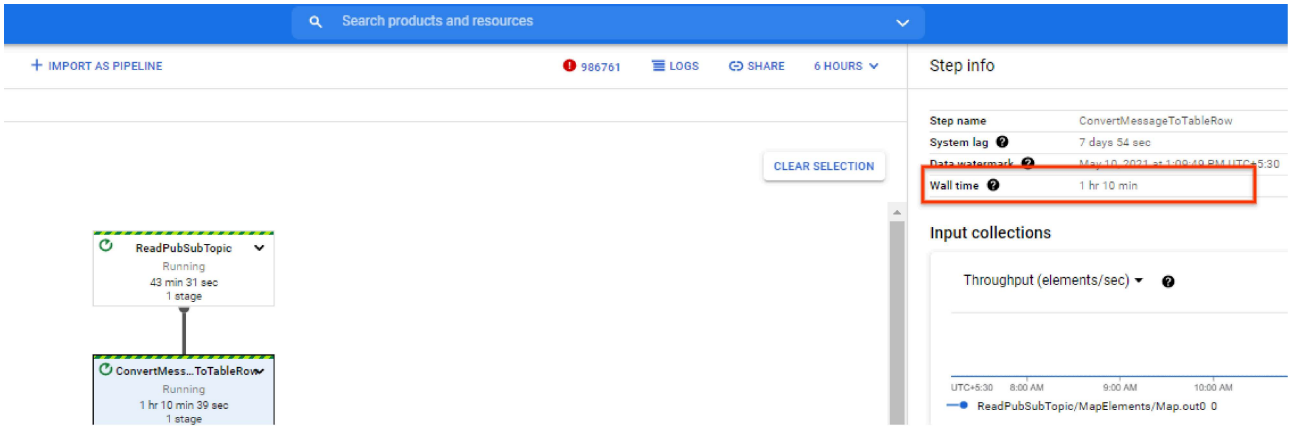


*Figure 3: The **Wall time** metric can help you ensure your pipeline is running efficiently.*

Side input metrics

Side input metrics show you how your side input access patterns and algorithms affect your pipeline's performance. When your pipeline uses a side input, Dataflow writes the collection to a persistent layer, such as a disk, and your transforms read from this persistent collection. These reads and writes affect your job's run time.

The Dataflow monitoring interface displays side input metrics when you select a transform that creates or consumes a side input collection. You can view the metrics in the **Side Input Metrics** section of the **Step info** panel.

Transforms that create a side input

If the selected transform creates a side input collection, the **Side Input Metrics** section displays the name of the collection, along with the following metrics:

- **Time spent writing:** The time spent writing the side input collection.
- **Bytes written:** The total number of bytes written to the side input collection.
- **Time & bytes read from side input:** A table that contains additional metrics for all transforms that consume the side input collection, called *side input consumers*.

The **Time & bytes read from side input** table contains the following information for each side input consumer:

- **Side input consumer:** The transform name of the side input consumer.
- **Time spent reading:** The time this consumer spent reading the side input collection.
- **Bytes read:** The number of bytes this consumer read from the side input collection.

If your pipeline has a composite transform that creates a side input, expand the composite transform until you see the specific subtransform that creates the side input. Then, select that subtransform to view the **Side Input Metrics** section.

Figure 4 shows side input metrics for a transform that creates a side input collection.

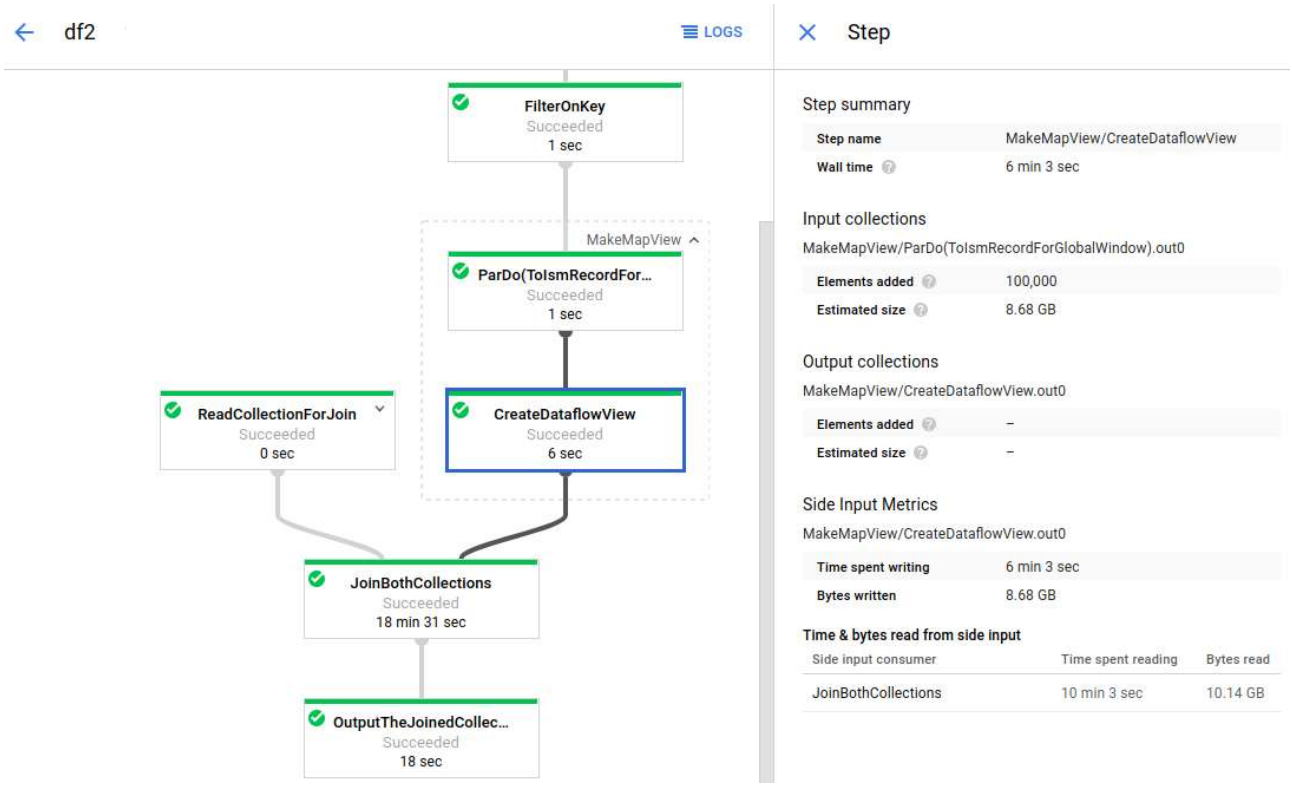


*Figure 4: The job graph has an expanded composite transform (`MakeMapView`). The subtransform that creates the side input (`CreateDataflowView`) is selected, and the side input metrics are visible in the **Step info** side panel.*

Transforms that consume one or more side inputs

If the selected transform consumes one or more side inputs, the **Side Input Metrics** section displays the **Time & bytes read from side input** table. This table contains the following information for each side input collection:

- **Side input collection:** The name of the side input collection.
- **Time spent reading:** The time the transform spent reading this side input collection.
- **Bytes read:** The number of bytes the transform read from this side input collection.

If your pipeline has a composite transform that reads a side input, expand the composite transform until you see the specific subtransform that reads the side input. Then, select that subtransform to view the **Side Input Metrics** section.

Figure 5 shows side input metrics for a transform that reads from a side input collection.

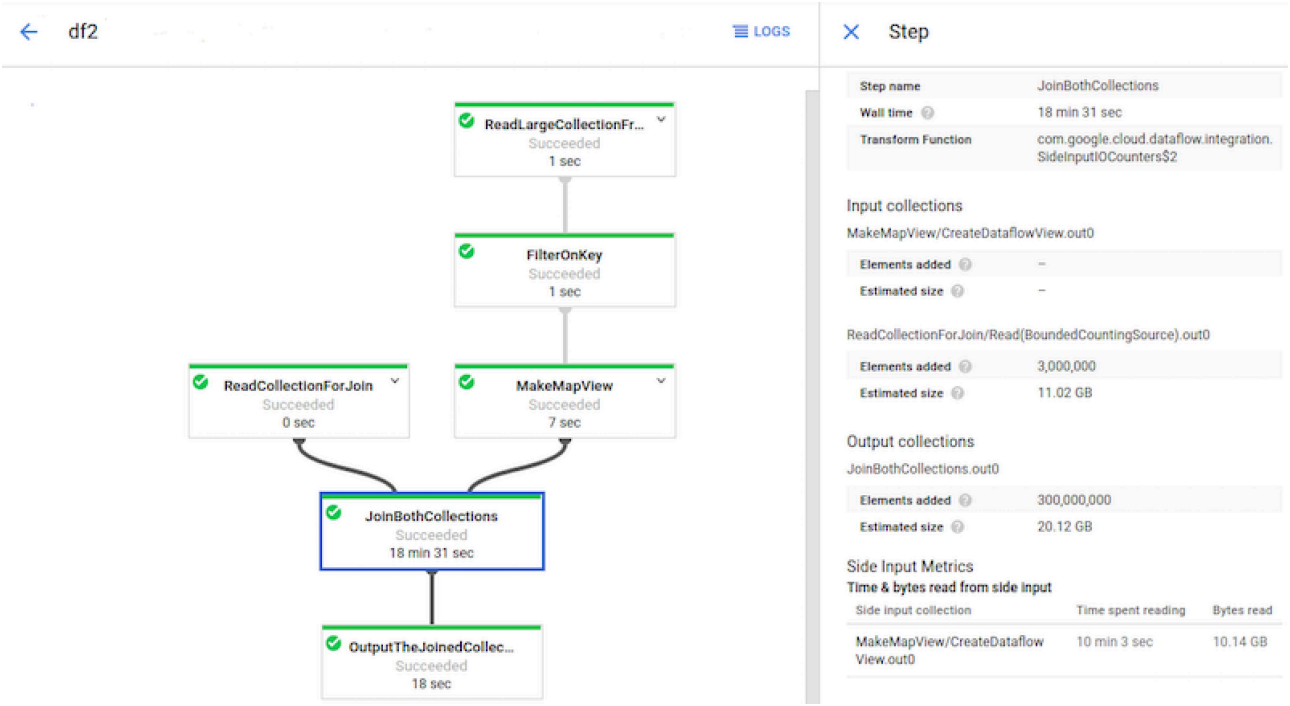


*Figure 5: The `JoinBothCollections` transform reads from a side input collection. `JoinBothCollections` is selected in the job graph and the side input metrics are visible in the **Step info** side panel.*

Identify side input performance issues

Reiteration is a common side input performance issue. If your side input `PCollection` is too large, workers can't cache the entire collection in memory. As a result, the workers must repeatedly read from the persistent side input collection.

In figure 6, side input metrics show that the total bytes read from the side input collection are much larger than the collection's size, total bytes written.

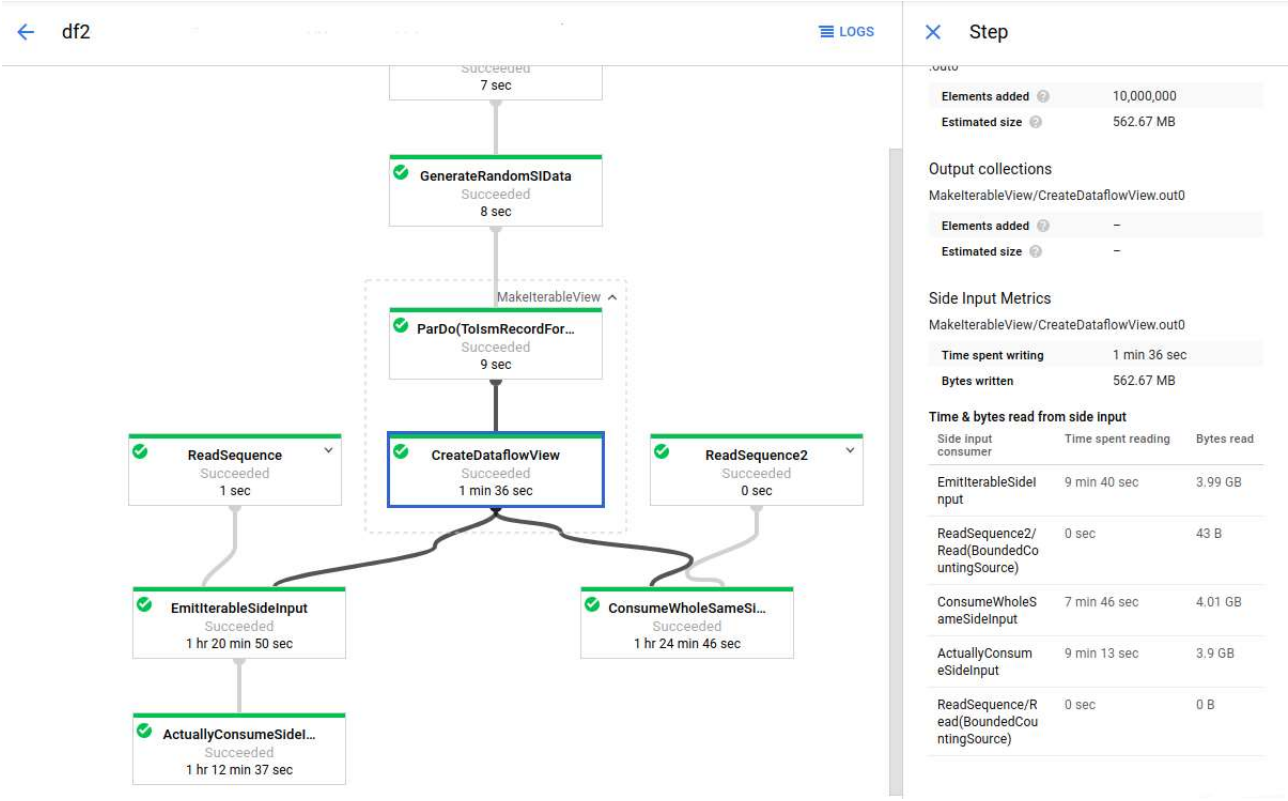


Figure 6: An example of reiteration. The side input collection is 563 MB, and the sum of the bytes read by consuming transforms is almost 12 GB.

To improve the performance of this pipeline, redesign your algorithm to avoid iterating or refetching the side input data. In this example, the pipeline creates the Cartesian product of two collections. The algorithm iterates through the entire side input collection for each element of the main collection. You can improve the access pattern of the pipeline by batching multiple elements of the main collection together. This change reduces the number of times workers must re-read the side input collection.

Another common performance issue can occur if your pipeline performs a join by applying a `ParDo` with one or more large side inputs. In this case, workers spend a large percentage of the processing time for the join operation reading from the side input collections.

Figure 7 shows example side input metrics for this issue:

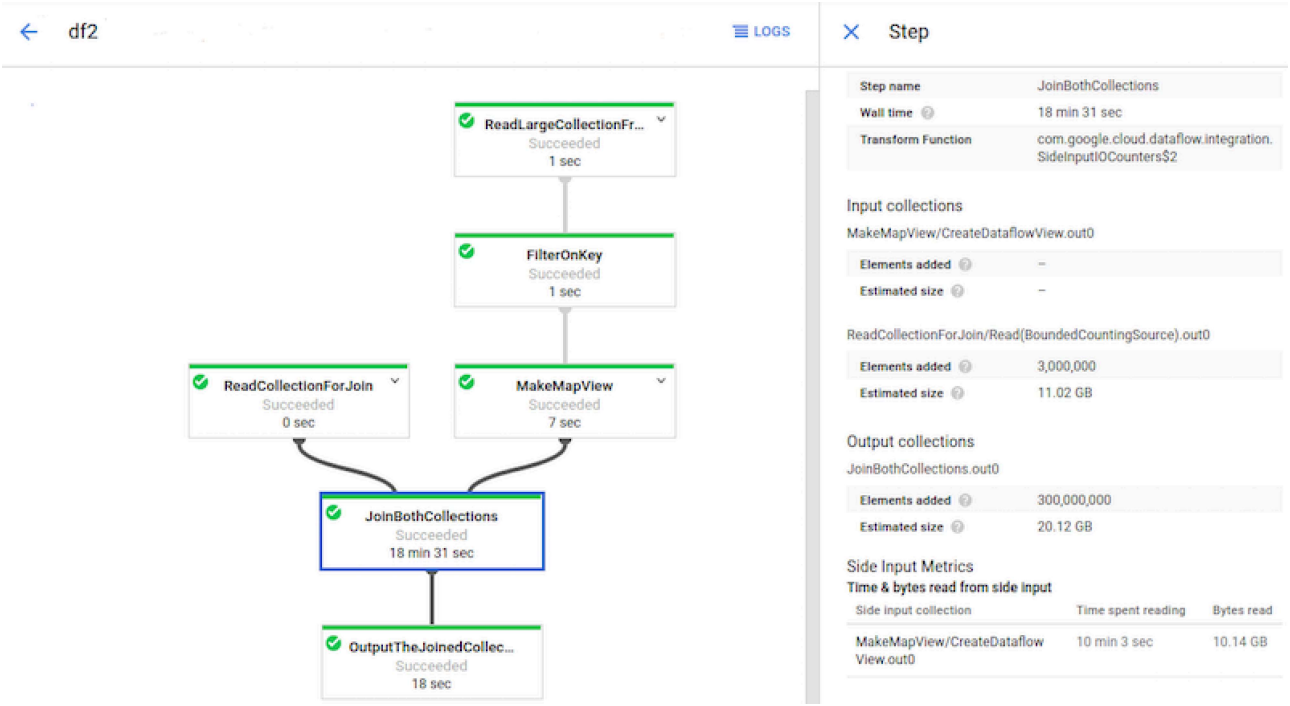


Figure 7: The `JoinBothCollections` transform has a total processing time of 18 min 31 sec. Workers spend the majority of the processing time (10 min 3 sec) reading from the 10 GB side input collection.

To improve the performance of this pipeline, use CoGroupByKey instead of side inputs.