Google Cloud

Blog

Google Cloud

Blog

Google Cloud

Blog

Google Cloud

Guide to common Cloud Dataflow use-case patterns, Part 1

June 16, 2017

Reza Rokni
Senior Staff Developer Advocate Dataflow

John LaBarge
Solution Architect

Editor's note: This is part one of a series on common Dataflow use-case patterns. You can find part two here.

As Google Cloud Dataflow adoption for large-scale processing of streaming and batch data pipelines has ramped up in the past couple of years, the Google Cloud solution architects team has been working closely with numerous Cloud Dataflow customers on everything from designing small POCs to fit-and-finish for large production deployments.

In this open-ended series, we'll describe the most common patterns across these customers that in combination cover an overwhelming majority of use cases (and as new patterns emerge over time, we'll keep you informed). Each pattern includes a

Pattern: Pushing data to multiple storage locations

Description:

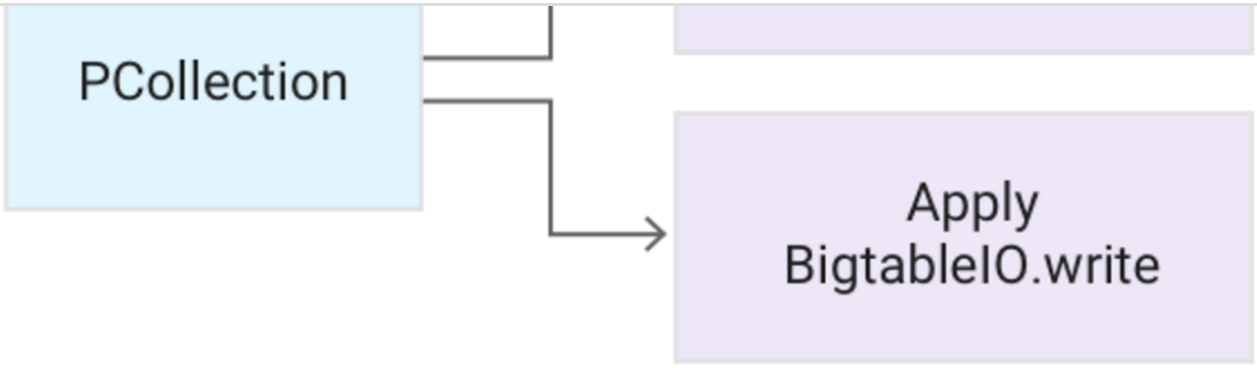
Covers the common pattern in which one has two different use cases for the same data and thus needs to use two different storage engines.

Example:

You have financial time-series data you need to store in a manner that allows you to:

1) run large-scale SQL aggregations, and 2) do small range-scan lookups, getting a small number of rows out of TBs of data. Given these requirements, the recommended approach will be to write the data to BigQuery for #1 and to Cloud Bigtable for #2. Solution:

A `PCollection` is immutable, so you can apply multiple transforms to the same one.



Pseudocode:

```
#Create branches by applying multiple transforms to the same
PCollection:
PCollection myCollection = {}..
myCollection.apply(BigQueryIO.write)
myCollection.apply(BigtableIO.write)
```

Pattern: Slowly-changing lookup cache

Description:

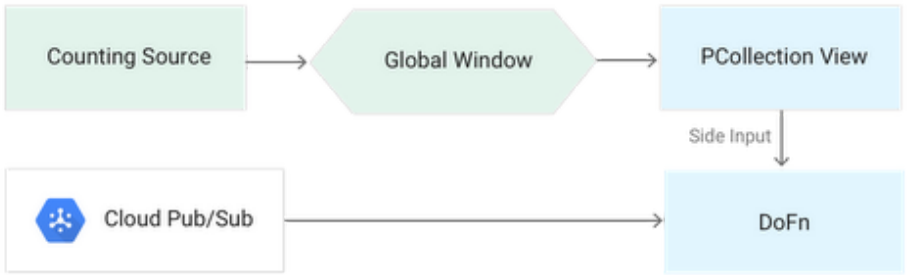
In streaming mode, lookup tables need to be accessible by your pipeline. If the lookup table never changes, then the standard Cloud Dataflow `SideInput` pattern reading from a bounded source such as BigQuery is a perfect fit. However, if the lookup data changes over time, in streaming mode there are additional considerations and options. The pattern described here

You have an ID field for the category of page type from which a clickstream event originates (e.g., Sales, Support, Admin). You want to enrich these elements with the description of the event stored in a BigQuery table.

Solution:

Use the Cloud Dataflow `Counting` source transform to emit a value daily, beginning on the day you create the pipeline.

- Pass this value into a global window via a data-driven trigger that activates on each element.
- In a DoFn, use this process as a trigger to pull data from your bounded source (such as BigQuery).
- Create your `SideInput` for use in downstream transforms.



Pseudocode

```
PCollection mainStream = p.apply(PubSubIO.read());
PCollection countingSource =
p.apply(CountingInput.unbounded().withRate(1,
Duration.standardDay(1))))
```

```
(new GlobalWindows())
.triggering(Repeatedly.forever(AfterProcessingTime.pastFirstElem
entInPane())
).apply(View.asMap())

mainstream.apply(DoFn().withsideinput(sideInput)
```

Note: It's important that you set the update frequency so that `SideInput` is updated in time for the streaming elements that require it. Because this pattern uses a global-window `SideInput`, matching to elements being processed will be nondeterministic. In most cases the `SideInput` will be available to all hosts shortly after update, but for large numbers of machines this step can take tens of seconds.

Pattern: Calling external services for data enrichment

Description:

This pattern will make a call out to an external service to enrich the data flowing through the system.

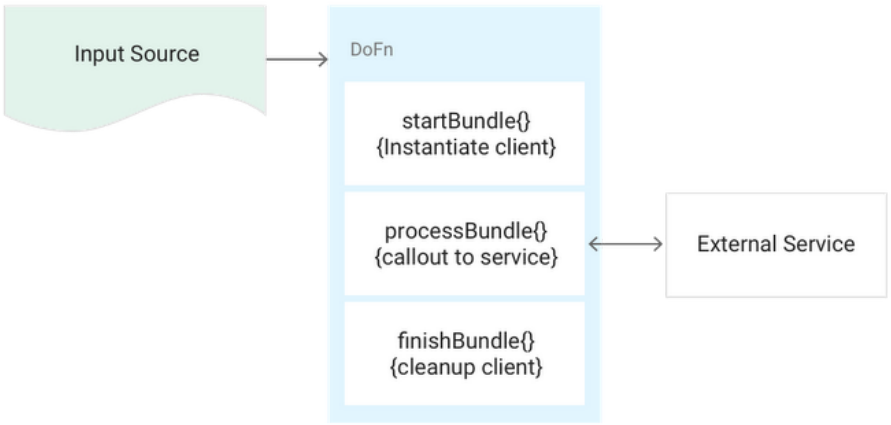
Example:

You need to give new website users a globally unique identifier using a service that takes in data points and returns a GUUID.

Solution:

control to make any type of connection that you choose, so long as the firewall rules you set up within your project/network allow it.

- If you're using a client in the `DoFn` that has heavy instantiation steps, rather than create that object in each `DoFn` call:
  - If the client is thread-safe and serializable, create it statically in the class definition of the `DoFn` .
  - If it's not thread-safe, create a new object in the `startBundle` method of `DoFn` . By doing so, the client will be reused across all elements of a bundle, saving initialization time.



Pseudocode

```
public class External extends DoFn {
@Override
public void startBundle(){
```

```
@Override
public void processElement(){
  Call out to external service
}
}

@Override
public void finishBundle(){
  Shutdown your external service client if needed
}
}
```

Note: When using this pattern, be sure to plan for the load that's placed on the external service and any associated backpressure. For example, imagine a pipeline that's processing tens of thousands of messages per second in steady state. If you made a callout per element, you would need the system to deal with the same number of API calls per second. Also, if the call takes on average 1 sec, that would cause massive backpressure on the pipeline. In these circumstances you should consider batching these requests, instead.

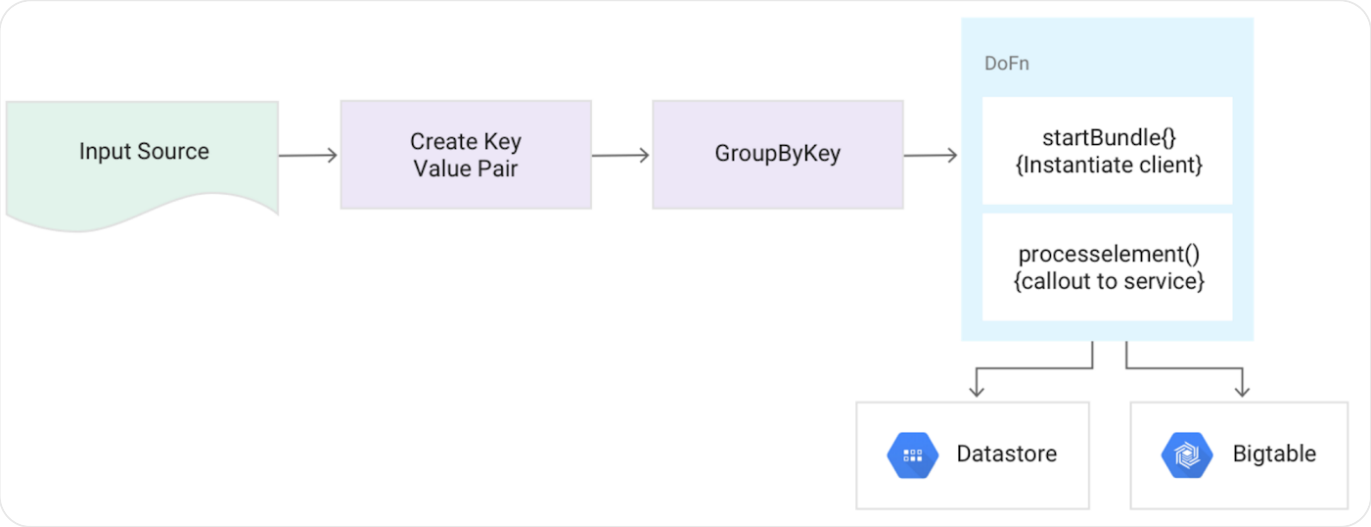
Pattern: Dealing with bad data

Description:

You should always defensively plan for bad or unexpectedly shaped data. A production system not only needs to guard against invalid input in a try-catch block but also to preserve that data for future re-processing.

Solution:

- Within the `DoFn` always use a try-catch block around activities like parsing data. In the exception block, rather than just log the issue, send the raw data out as a `SideOutput` into a storage medium such as BigQuery or Cloud Bigtable using a String column to store the raw unprocessed data.
  - Use Tuple tags to access multiple outputs from the resulting `PCollection` .



Pseudocode

```
final TupleTag successTag ;
final TupleTag deadLetterTag;
PCollection input = /* … */;
PCollectionTuple outputTuple = input.apply(ParDo.of(new DoFn() {
```

```
    // Logging
c.sideOutput(deadLetterTag, c.element());
  }
}).withOutputTags(successTag, TupleTagList.of(deadLetterTag)));
// Write the dead letter inputs to a BigQuery table for later
analysis
outputTuple.get(deadLetterTag).apply(BigQueryIO.write(...));
// Retrieve the successful elements...
PCollection success = outputTuple.get(successTag);
```

Pattern: Starting jobs using a simple REST endpoint

Description:

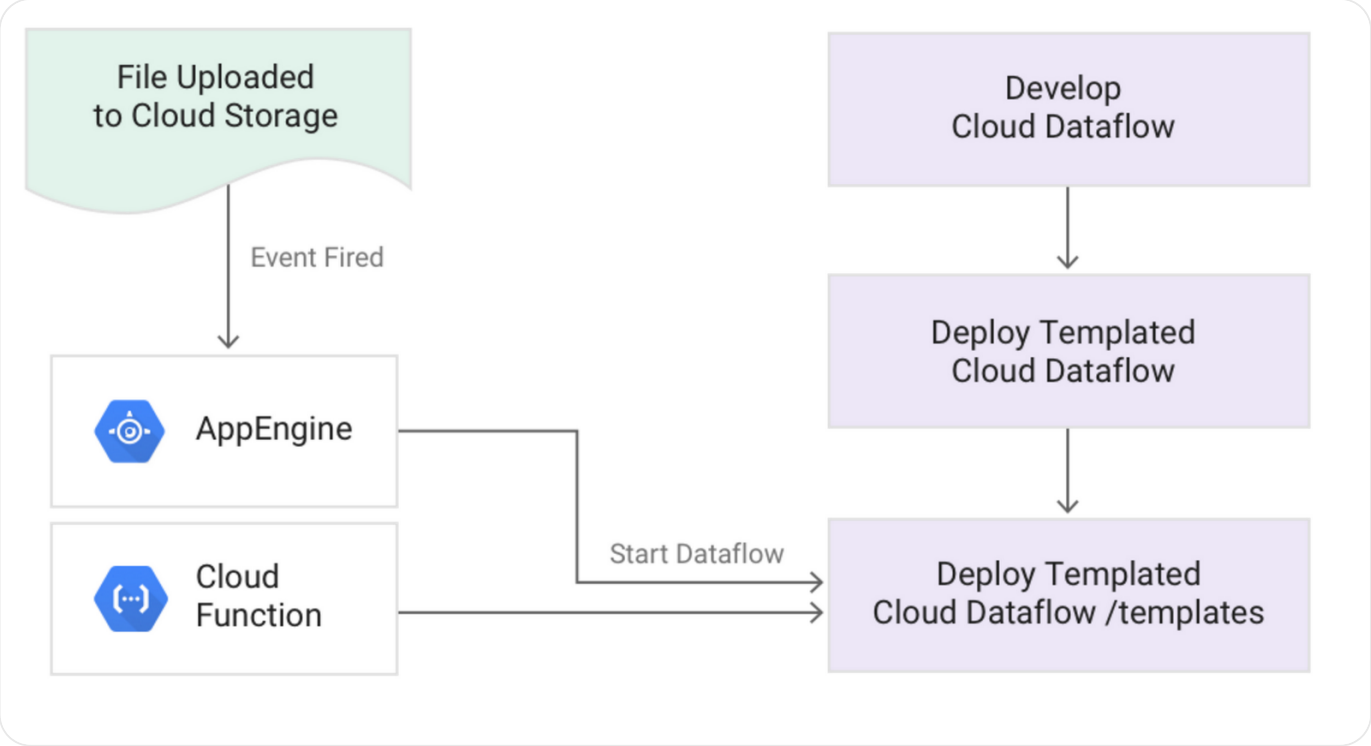
Many Cloud Dataflow jobs, especially those in batch mode, are triggered by real-world events such as a file landing in Google Cloud Storage or serve as the next step in a sequence of data pipeline transformations. One common way to implement this approach is to package the Cloud Dataflow SDK and create an executable file that launches the job. But a better option is to use a simple REST endpoint to trigger the Cloud Dataflow pipeline.

Example:

Your retail stores upload files to Cloud Storage throughout the day. Each file is processed using a batch job, and that job should start immediately after the file is uploaded.

Google Cloud

Blog



Pseudocode:

```
POST
https://dataflow.googleapis.com/v1b3/projects/{YOUR_PROJECT_ID}/
templates
    {
        "jobName": "myjobname",
        "gcsPath":
"gs://{YOUR_BUCKET_NAME}/templates/TemplateName"
        "parameters": {
            "inputFile" :
"gs://{YOUR_BUCKET_NAME}/input/my_input.txt",
```

```
            "zone": "us-central1-f"
        }
    }
```

Next steps

This initial list of Cloud Dataflow user patterns should hopefully help you think about new and different ways to get value from the service. Learn more in the second post in this series.

Google Cloud

Google Cloud

Blog

Posted in Google Cloud—Data Analytics

Related articles

Data Analytics

Automate data governance, extend your data fabric with Dataplex-BigLake integration

By Uri Gilad • 2-minute read

Training and Certifications

How HSBC is upskilling at scale with Google Cloud

By Adrian Phelan • 3-minute read

Data Analytics

BigQuery Omni: solving cross-cloud challenges by bringing analytics to your data

By Vidya Shanmugam • 6-minute read

AI & Machine Learning

Efficient PyTorch training with Vertex AI

By Xiang Xu • 7-minute read