Google Cloud

Blog

Google Cloud

Blog

Google Cloud

Blog

Google Cloud

Guide to common Cloud Dataflow use-case patterns, Part 2

August 21, 2017

Reza Rokni
Senior Staff Developer Advocate Dataflow

John LaBarge
Solution Architect

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

This open-ended series (see first installment) documents the most common patterns we've seen across production Cloud Dataflow deployments. In Part 2, we're bringing you another batch — including solutions and pseudocode for implementation in your own environment.

Let's dive in!

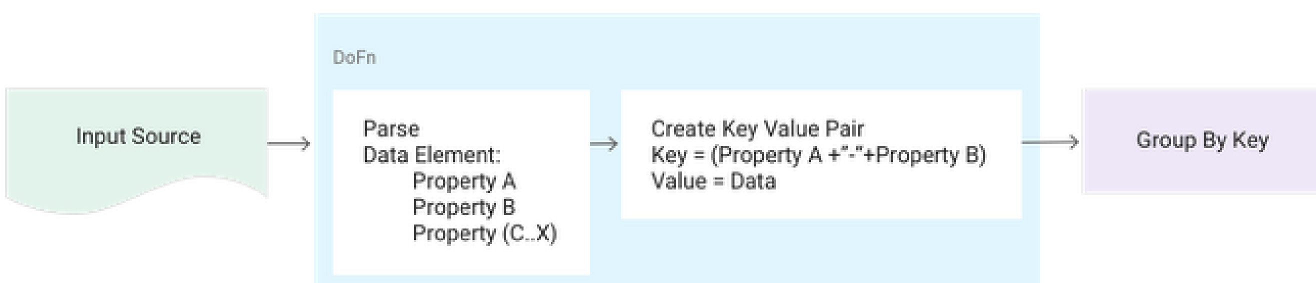
Data elements need to be grouped by multiple properties.

Example:

IoT data arrives with location and device-type properties. You need to group these elements based on both these properties.

Solution:

1. Create a composite key made up of both properties.
2. Use the composite key as input to the `KV.of()` factory method to create a KV object that consists of the composite key `(K)` and pertaining data element from which the composite key was derived (V).
3. Use a `GroupByKey` transform to get your desired groupings (e.g., a resulting `PCollection>>`).



Pseudocode:

```
PCollection<Data> data = p.apply(...).apply(parseData)
```

```
compositeKV.apply(GroupByKey())
)
```

Note: building a string using concatenation of "-" works but is not the best approach for production systems. Instead, we generally recommend creating a new class to represent the composite key and likely using @DefaultCoder. See "Annotating a Custom Data Type with a Default Coder" in the docs for Cloud Dataflow SDKs 1.x; for 2.x, see this.

Pattern: Joining two PCollections on a common key

Description:

Joining of two datasets based on a common key.

Example:

You want to join clickstream data and CRM data in batch mode via the user ID field.

Solution:

- For each dataset in the join, create a key-value pair using the utility KV class (see above).
- Create tags so that you can access the various collections from the result of the join.

right where the value for the left collection is null. Finally, to do an inner join, include in the result set only those items where there are elements for both the left and right collections.

Note: If possible, use `SideInputs` for any activity where one of the join tables is actually small — around 100MB in stream mode or under a 1GB in batch mode. This will perform much better than the join operation described here. This join is shuffle heavy and is best used when both collections being joined are larger than those guidelines. `SideInputs` are not precisely equivalent, however, as they're only read when data shows up on the main input. In contrast, a `CoGroupByKey` triggers if data shows up on either side. Think of `SideInputs` as an inner-loop join — with nested loops, the inner loop only runs if the outer loop runs first.

Pseudocode:

```
// Each shares a common key ("K").
  PCollection<KV<K, V1>> pt1 = ...;
  PCollection<KV<K, V2>> pt2 = ...;
  // Create tuple tags for the value types in each collection.
  final TupleTag<V1> t1 = new TupleTag<V1>();
  final TupleTag<V2> t2 = new TupleTag<V2>();
  //Merge collection values into a CoGbkResult collection
  PCollection<KV<K, CoGbkResult>> coGbkResultCollection =
    KeyedPCollectionTuple.of(t1, pt1)
                         .and(t2, pt2)
                         .apply(CoGroupByKey.<K>create());
  // Access results and do something.
  PCollection<T> finalResultCollection =
```

```
            KV<K, CoGbkResult> e = c.element();
            // Get all collection 1 values
            Iterable<V1> pt1Vals = e.getValue().getAll(t1);
            // Now get collection 2 values
// Assuming the results has 2 unique keys...
            V2 pt2Val = e.getValue().getOnly(t2);
            ... Do Something ....
            c.output(...some T...);
          }
        }));
```

Note: Consider using the new [service-side Dataflow Shuffle] (in public beta at the time of this writing) as an optimization technique for your CoGroupByKey.

Pattern: Streaming mode large lookup tables

Description:

A large (in GBs) lookup table must be accurate, and changes often or does not fit in memory.

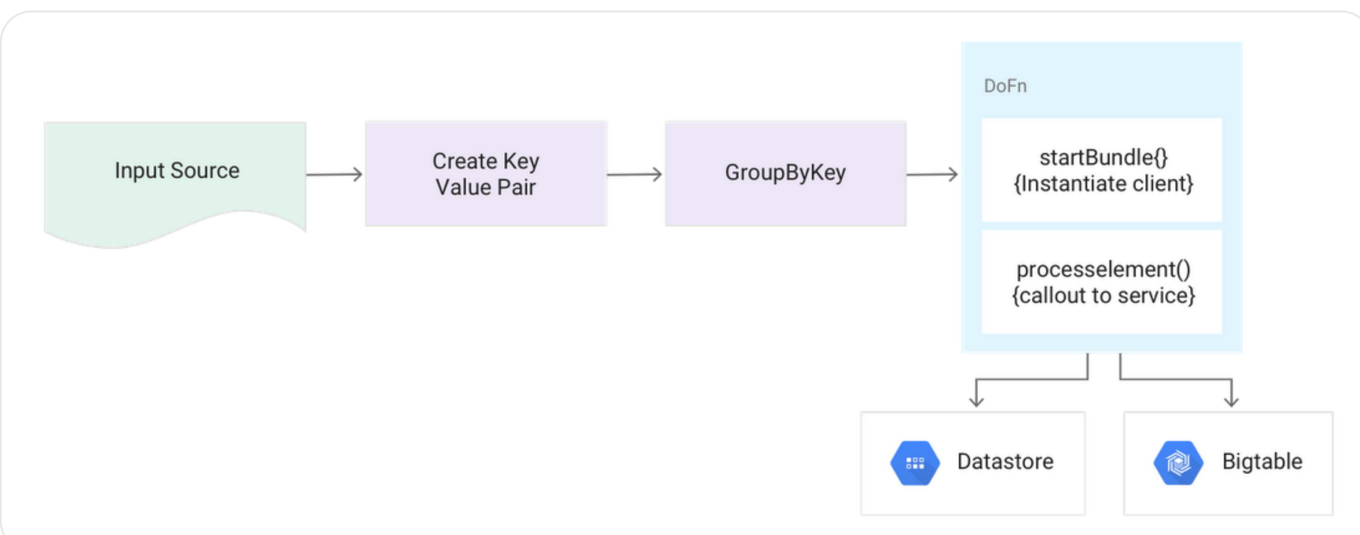
Example:

You have point of sale information from a retailer and need to associate the name of the product item with the data record which contains the `productID`. There are

Use the "Calling external services for data enrichment" pattern but rather than calling a micro service, call a read-optimized NoSQL database (such as Cloud Datastore or Cloud Bigtable) directly.

- For each value to be looked up, create a Key Value pair using the `KV` utility class.
- Do a `GroupByKey` to create batches of the same key type to make the call against the database.
- In the `DoFn`, make a call out to the database for that key and then apply the value to all values by walking through the iterable. Follow best practices with client instantiation as described in "Calling external services for data enrichment".

Note: We recommend that you cache the result of the external lookups to reduce the number of lookups. Pipelines can get extremely bottlenecked if they do a lookup on every element. (Remember, you're replacing a process that usually takes nanoseconds with one that could take 100s of milliseconds.)



```
p.apply(...).apply(parseProducts)
PCollection<KV<String,Product>> identifiedProducts =
data.apply(KV.of(...)}
PCollection<KV<String,Iterable<Product>> productsGroupedById =
identifiedProducts.apply(GroupByKey());
groupedProducts.apply(ParDo.of(new DoFn(){
LookupClient lookupClient;
public startBundle(ProcessContext c) {
  //create lookup client
lookupClient = new LookupClient(/*important parameters*);
}
public processElement(ProcessContext c){
    KV<String,Iterable<Product>> groupedProducts = c.element();
    String productId = groupedProducts.getKey();
    String productNameForKey = lookupClient.getValueForKey(key);

    for (product : groupedData.getValue()) {

c.output(Product.fromProduct(product).withProductName(productNam
e))
    }

});
public finishBundle(){
   lookupClient.cleanup();
}
```

Two streams are windowed in different ways — for example, fixed windows of 5 mins and 1 min respectively — but also need to be joined.

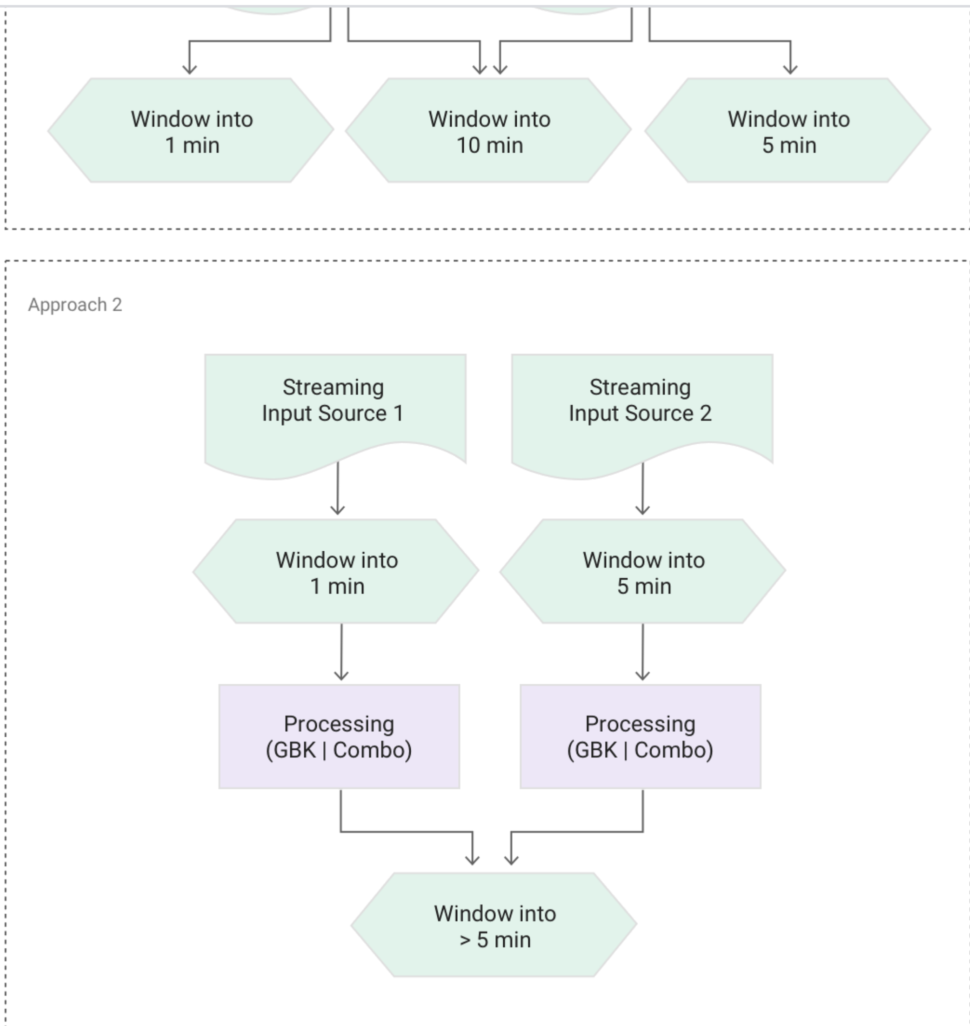
Example:

You have multiple IoT devices attached to a piece of equipment, with various alerts being computed and streamed to Cloud Dataflow. Some of the alerts occur in 1-min fixed windows, and some of the events occur in 5-min fixed windows. You also want to merge all the data for cross-signal analysis.

Solution:

To join two streams, the respective windowing transforms have to match. Two options are available:

- Similar to the "Pushing data to multiple storage locations" pattern, create multiple branches to support three different windowing strategies.
- Re-window the 1-min and 5-min streams into a new window strategy that's larger or equal in size to the window of the largest stream.



Pseudocode:

```
-- ************************** Approach 1
PCollection streamA = p.apply(StreamingSource A);
PCollection streamB = p.apply(StreamingSource B);
PCollection StreamAWindow1Min = streamA.
(WindowInto(FixedWindow(1minute));
```

```
(WindowInto(FixedWindow(8minute));
-- ************************** Approach 2
PCollection StreamAWindow1Min = p.apply(StreamingSource A)
.apply(WindowInto(FixedWindow(1minute))
.apply(some aggregation);
PCollection StreamAWindow5Min = p.apply(StreamingSource B)
.apply(WindowInto(FixedWindow(5minute))
.apply(some aggregation);
-- re-window
p.(Flatten(
StreamAWindow1Min.apply(WindowInto(FixedWindow(10minute)),
StreamAWindow5Min.apply(WindowInto(FixedWindow(10minute)),
)
```

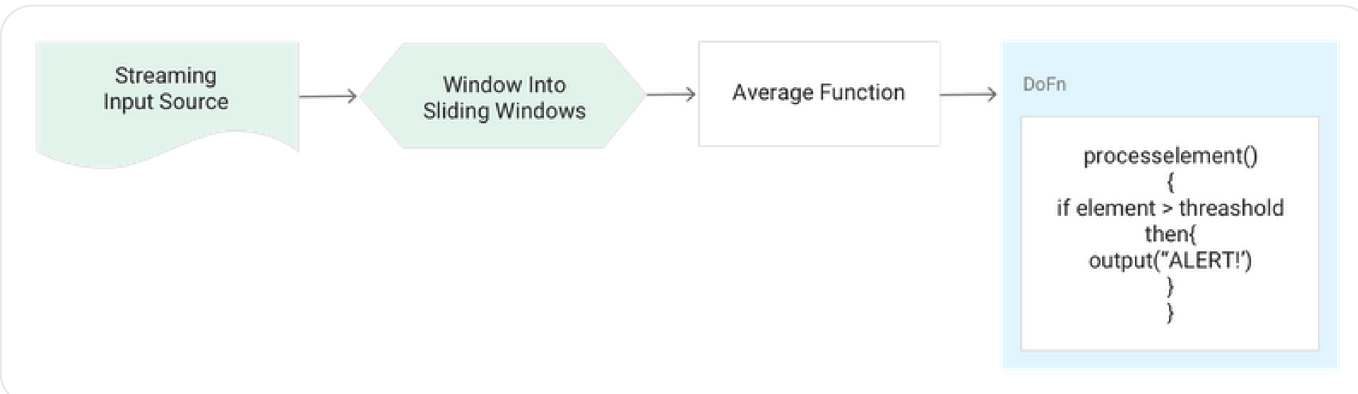
Pattern: Threshold detection with time-series data

Description:

This use case — a common one for stream processing — can be thought of as a simple way to detect anomalies when the rules are easily definable (i.e., generate a moving average and compare that with a rule that defines if a threshold has been reached).

Example:

You normally record around 100 visitors per second on your website during a promotion period; if the moving average over 1 hour is below 10 visitors per second, Consume the stream using an unbounded source like `PubSubIO` and window into sliding windows of the desired length and period. If the data structure is simple, use one of Cloud Dataflow's native aggregation functions such as `AVG` to calculate the moving average. Compare this `AVG` value against your predefined rules and if the value is over / under the threshold, and then fire an alert.



Pseudocode:

```
PCollection stream = p.apply(StreamingSource);
PCollection movingAverage =
stream.(WindowInto(
SlidingWindows.of(Duration).every(Duration)
.ParDo.of(AverageFunction);
movingAverage.apply(ParDo.of(DoFn(){Check if value >
Threshold}))
```

Next steps

Google Cloud

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

In this series, we'll describe the most common Dataflow use-case patterns, including description, example, solution and pseudocode.

By Reza Rokni • 5-minute read

Posted in [Google Cloud](#)—[Data Analytics](#)

Related articles

Data Analytics

Automate data governance, extend your data fabric with Dataplex-

Google Cloud

Blog

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

Follow us

Google Cloud Google Cloud Products Privacy Terms

Help English