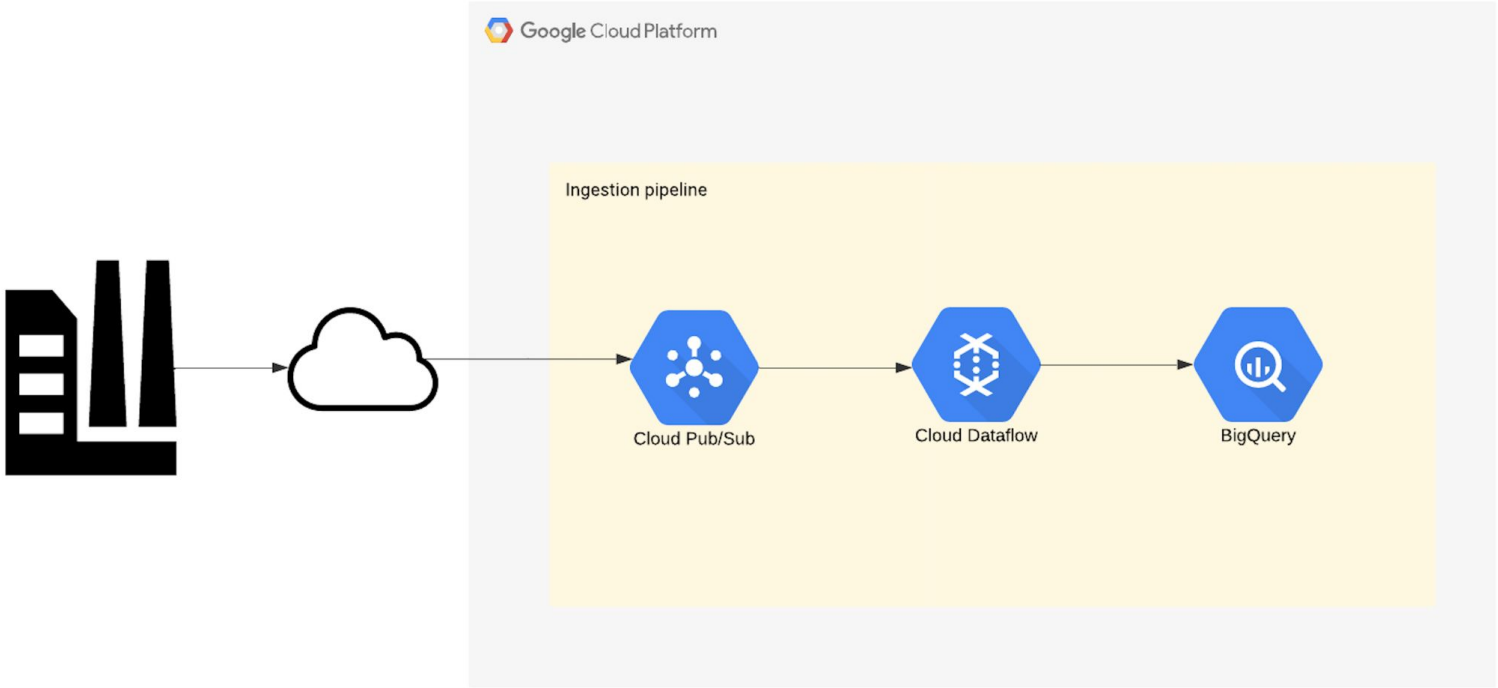
Exception management inside Beam

David Duarte
Data Architect (SFEIR)
https://www.linkedin.com/in/david-duarte-5b0627160/

Mazlum Tosun
Head of data (GroupBees) and tech lead GCP
https://www.linkedin.com/in/mazlum-tosun-900b1812/

Exception management

Basic streaming ingestion pipeline



Common errors when ingesting data

Input format is wrong

Output format is wrong

Data pipeline code contains bugs

Used libraries contains bugs

Failure when interacting with external systems

Risks to not manage these common errors

Data loss:

- If message is ack but not stored when error occurs

Non usable logs:

- Errors are not logged in a standardized manner
- Error analysis can be difficult

Monitoring issues:

- How to monitoring efficiently?

How to manage common errors

Implement dead letters queues:

- When error occurs:
  - catch the error
  - store the message, its context and the associated error message to an output topic

Monitoring and alerting

Add monitoring and alerting on dedicated output:

- If dedicated output grows
  - Trigger alert

Implementation guidelines

In each stage of the Apache Beam pipeline :

- Add Try / Catch block to handle error
- Get error
- Store message and error in side output
- Persist the side output

Implementation issues

- Try/Catch on each pipeline stage introduce technical code inside business code
  - Can be difficult to read and maintain

Asgarde

Simplify error handling with Apache Beam

https://github.com/tosun-si/asgarde

Native error handling with Beam : ParDo and DoFn

Beam recommends treating errors with a dead letter queue

It means catching errors in the flow and, using side outputs, sinking errors to a file, database or any other output…

Beam suggests handling side outputs with `TupleTags` in a `DoFn` class, example :

Native error handling with Beam : ParDo and DoFn

```java
// Failure object.
public class Failure implements Serializable {
    private final String pipelineStep;
    private final Integer inputElement;
    private final Throwable exception;

    public static <T> Failure from(final String pipelineStep,
                                   final T element,
                                   final Throwable exception) {
        return new Failure(pipelineStep, element.toString(), exception);
    }
}

// Word count DoFn class.
public class WordCountFn extends DoFn<String, Integer> {

     private final TupleTag<Integer> outputTag = new TupleTag<Integer>() {};
     private final TupleTag<Failure> failuresTag = new TupleTag<Failure>() {};

     @ProcessElement
     public void processElement(ProcessContext ctx) {
        try {
            // Could throw ArithmeticException.
            final String word = ctx.element();
            ctx.output(1 / word.length());
        } catch (Throwable throwable) {
            final Failure failure = Failure.from("step", ctx.element(), throwable);
            ctx.output(failuresTag, failure);
        }
    }

    public TupleTag<Integer> getOutputTag() {
       return outputTag;
    }

    public TupleTag<Failure> getFailuresTag() {
       return failuresTag;
    }
}
```

Native error handling with Beam : ParDo and DoFn

```
// In Beam pipeline flow.
final PCollection<String> wordPCollection....

final WordCountFn wordCountFn = new WordCountFn();

final PCollectionTuple tuple = wordPCollection
           .apply("ParDo", ParDo.of(wordCountFn)
                .withOutputTags(wordCountFn.getOutputTag(), TupleTagList.of(wordCountFn.getFailuresTag())));

// Output PCollection via outputTag.
PCollection<Integer> outputCollection = tuple.get(wordCountFn.getOutputTag());

// Failures PCollection via failuresTag.
PCollection<Failure> failuresCollection = tuple.get(wordCountFn.getFailuresTag());
```

With this approach we can, in all steps, get the output and failures result PCollections.

Native error with handling Beam : MapElements and FlatMapElement

Beam also allows handling errors with built-in components like `MapElements` and `FlatMapElements`

```
// In Beam pipeline flow.
final PCollection<String> wordPCollection....

WithFailures.Result<PCollection<Integer>, Failure> result = wordPCollection
   .apply("Map", MapElements
           .into(TypeDescriptors.integers())
           .via((String word) -> 1 / word.length())  // Could throw ArithmeticException
           .exceptionsInto(TypeDescriptor.of(Failure.class))
           .exceptionsVia(exElt -> Failure.from("step", exElt))
   );

PCollection<String> output = result.output();
PCollection<Failure> failures = result.failures();
```

Comparison between approaches : usual Beam pipeline

In a usual Beam pipeline flow, steps are chained fluently:

```
final PCollection<Integer> outputPCollection = inputPCollection
                .apply("Map",
                   MapElements .into(TypeDescriptors.strings()).via((String word) -> word + "Test"))
                .apply("FlatMap",
                   FlatMapElements
                      .into(TypeDescriptors.strings())
                      .via((String line) -> Arrays.asList(Arrays.copyOfRange(line.split(" "), 1, 5))))
                .apply("Map ParDo", ParDo.of(new WordCountFn()));
```

Comparison between approaches : Usual Beam pipeline with error handling

Here's the same flow with error handling in each step:

```
WithFailures.Result<PCollection<String>, Failure> result1 = input
        .apply("Map", MapElements
                        .into(TypeDescriptors.strings())
                        .via((String word) -> word + "Test")
                        .exceptionsInto(TypeDescriptor.of(Failure.class))
                        .exceptionsVia(exElt -> Failure.from("step", exElt)));

final PCollection<String> output1 = result1.output();
final PCollection<Failure> failure1 = result1.failures();

WithFailures.Result<PCollection<String>, Failure> result2 = output1
        .apply("FlatMap", FlatMapElements
                            .into(TypeDescriptors.strings())
                            .via((String line) -> Arrays.asList(Arrays.copyOfRange(line.split(" "), 1, 5)))
                            .exceptionsInto(TypeDescriptor.of(Failure.class))
                            .exceptionsVia(exElt -> Failure.from("step", exElt)));

final PCollection<String> output2 = result1.output();
final PCollection<Failure> failure2 = result1.failures();

final PCollectionTuple result3 = output2
        .apply("Map ParDo", ParDo.of(wordCountFn)
                                .withOutputTags(wordCountFn.getOutputTag(),
                                                TupleTagList.of(wordCountFn.getFailuresTag())));

final PCollection<Integer> output3 = result3.get(wordCountFn.getOutputTag());
final PCollection<Failure> failure3 = result3.get(wordCountFn.getFailuresTag());

final PCollection<Failure> allFailures = PCollectionList
        .of(failure1)
        .and(failure2)
        .and(failure3)
        .apply(Flatten.pCollections());
```

Comparison between approaches : Usual Beam pipeline with error handling

Problems with this approach:

- We loose the native fluent style on apply chains, because we have to handle output and error for each step.
- For MapElements and FlatMapElements we have to always add exceptionsInto and exceptionsVia (can be centralized).
- For each custom DoFn, we have to duplicate the code of TupleTag logic and the try catch block (can be centralized).
- The code is verbose.
- There is no centralized code to concat all the errors, we have to concat all failures (can be centralized).

Comparison between approaches : Usual Beam pipeline with error handling using Asgarde

Here's the same flow with error handling, but using Asgarde library instead:

```
final WithFailures.Result<PCollection<Integer>, Failure> resultComposer = CollectionComposer.of(input)
        .apply("Map", MapElements.into(TypeDescriptors.strings()).via((String word) -> word + "Test"))
        .apply("FlatMap", FlatMapElements
                .into(TypeDescriptors.strings())
                .via((String line) -> Arrays.asList(Arrays.copyOfRange(line.split(" "), 1, 5))))
        .apply("ParDo", MapElementFn.into(TypeDescriptors.integers()).via(word -> 1 / word.length()))
        .getResult();
```

Comparison between approaches : Usual Beam pipeline with error handling using Asgarde

Purpose of the library:

- Wrap all error handling logic in a composer class.
- Wrap exceptionsInto and exceptionsVia usage in the native Beam classes MapElements and FlatMapElements.
- Keep the fluent style natively proposed by Beam in apply methods while checking for failures and offer a less verbose way of handling errors.
- Expose custom DoFn classes with centralized try/catch blocks (loan pattern) and Tuple tags.
- There is no centralized code to concat all the errors, we have to concat all failures (can be centralized).
- Expose an easier access to the @Setup, @StartBundle, @FinishBundle, @Teardown steps of DoFn classes.
- Expose a way to handle errors in filtering logic (currently not available with Beam's Filter.by).

Comparison between approaches : Usual Beam pipeline with error handling using Asgarde Kotlin

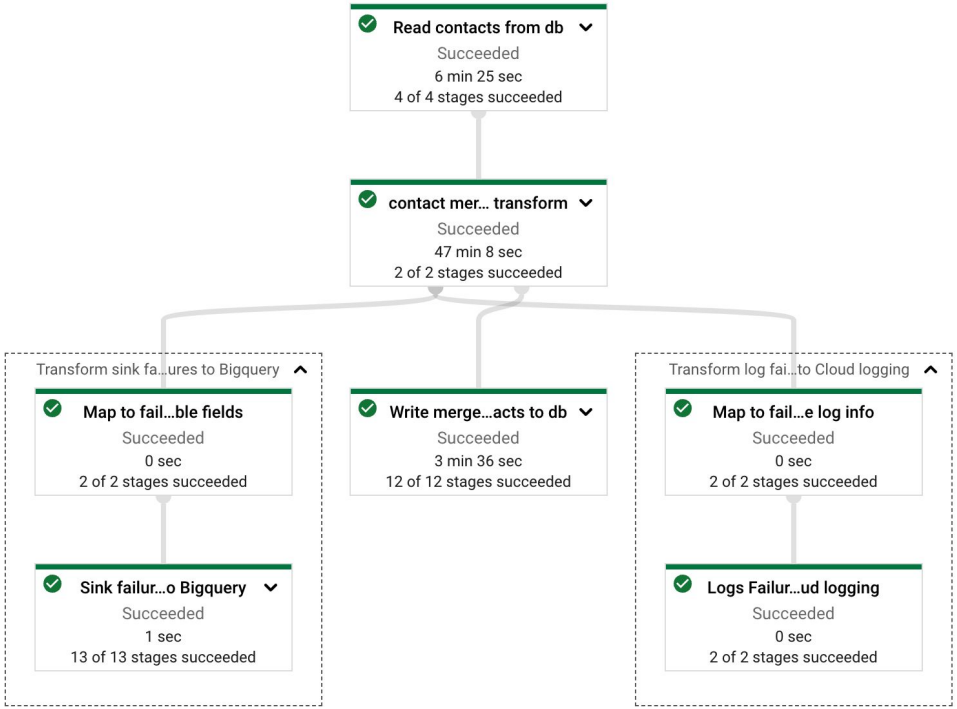
Extensions are proposed to use Asgarde with Kotlin:

```
import fr.groupbees.asgarde.*

val result: Result<PCollection<Int>, Failure> = CollectionComposer.of(words)
      .map("Map") { word -> word + "Test" }
      .flatMap("FlatMap") { Arrays.asList(*Arrays.copyOfRange(it.split(" ").toTypedArray(), 1, 5)) }
      .mapFn("ParDo", { word -> 1 / word.length })
      .result
```

Example of bad sinks steps in a Dataflow DAG

Bad sinks with DLQ to BigQuery and Cloud Logging



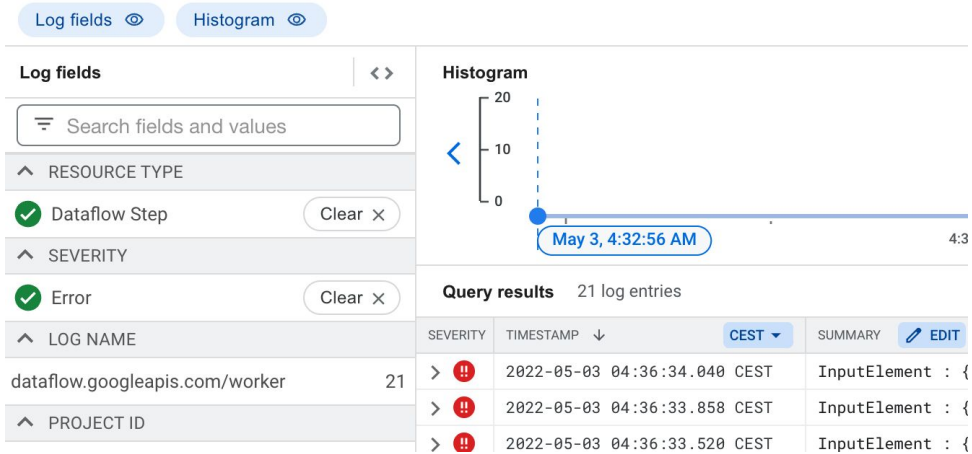
Log based metric and alerting based on bad sinks

We can create a log based metric and alerting policy based on the bad sink targeted to cloud logging

Policy documentation

Alert fired when Dataflow jobs have errors caught by our error handling system. In our Dataflow jobs, we store errors without stopping jobs. In this case, specific info (input element and the linked stack trace) are logged to Stackdriver.

VIEW INCIDENT VIEW LOGS



Thank you!