## **Desenvolvimento inicial**

É possível utilizar o repositório X como start do desenvolvimento com Apache Beam. Dentro do projeto, contém as bibliotecas iniciais para desenvolvimento e os runners para desenvolvimento local e no Dataflow.

Dentro do projeto contém 3 exemplos de job.

* Um seria o exemplo fornecido pela Apache, o WordCount. Caso queira dar os primeiros passos com o Apache Beam, leia o tutorial fornecido pela Apache <https://beam.apache.org/get-started/wordcount-example/>
* O segundo seria um dos templates do Google para pegar mensagens em um tópico do Pub/Sub e salvar no Cloud Storage.
* O terceiro, é um exemplo para lidar com dados inválidos.

## **Casos de uso comuns**

É possível ver exemplos de casos de uso nos links a seguir

* <https://cloud.google.com/blog/products/data-analytics/guide-to-common-cloud-dataflow-use-case-patterns-part-1>
* <https://cloud.google.com/blog/products/data-analytics/guide-to-common-cloud-dataflow-use-case-patterns-part-2>
* <https://beam.apache.org/documentation/patterns/overview/>
* <https://cloud.google.com/dataflow/docs/samples/reference-patterns>

## **Boas práticas de desenvolvimento com Dataflow**

### **Lidando com dados inválidos**

Um caso de uso muito comum é lidar algum dado inválido ou inesperado. É necessário estar preparado contra esses tipos de dados e preservar o dado para um futuro reprocessamento.

A solução para esse cenário seria em 2 passos:

* Dentro do DoFn, usar o bloco try/catch sobre parse de dados, dentro do bloco catch realizar alguma ação como log, enviar o dado para algum tópico etc.
* Utilizar as turple tag para acessar múltiplos resultados do PCollection

Um exemplo básico seria

final *TupleTag* successTag ;

final *TupleTag* deadLetterTag;

*Pcollection* input = /\* … \*/;

*PcollectionTuple* outputTuple = input.apply(ParDo.of(**new** DoFn() {

  @*Override*

*void* processElement(*ProcessContext* *c*) {

  try {

    c.output(process(c.element());

  } catch (*Exception* *e*) {

      // 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);

Dentro do repo X, tem um exemplo chamado DealingWithBadData. É possível utilizar ele como base.

<https://cloud.google.com/architecture/building-production-ready-data-pipelines-using-dataflow-developing-and-testing#development_best_practices>

### **SideInputs x CoGroupByKey**

* O Dataflow suporta o uso de SideInputs persistindo os dados em um armazenamento persistente, ficando disponível para todos os Workers. Portanto, o uso de side inputs é recomendável em buscas/junções de dados de proporções menores e em operações que são utilizadas múltiplas vezes.
* Para melhor otimização, fazer o cache do side input em memória melhora a velocidade em operações de busca
* O CoGroupByKey é recomendável em casos de busca de grande proporção de dados de um PCollection, que poderia exceder a memória do Worker

## **Limitar o tamanho de batch e chamadas simultâneas**

Ao chamar serviços externos, é possível reduzir o número de chamadas e agrupar o número de elementos ao utilizar a função [GroupIntoBatches](https://beam.apache.org/releases/javadoc/2.34.0/org/apache/beam/sdk/transforms/GroupIntoBatches.html). Com essa função, é possível o envio de um payload único, ao invés de individual.

É possível também limitar o número de chamadas paralelas para um serviço externo com base em uma chave para particionar os dados de entrada.

Considere utilizar algumas das abordagens para produzir chaves para elementos:

* Escolha um atributo do conjunto de dados para usar como uma chave, por exemplo como o ID de um conjunto de elementos;
* Gerar chaves de dados para dividir elementos aleatoriamente em um número fixo de partições.

O exemplo a seguir mostra como dividir elementos aleatoriamente em 10 partições:

// PII or classified data which needs redaction.

*PCollection*<String> sensitiveData = ...;

*int* numPartitions = 10; // Number of parallel batches to create.

*PCollection*<*KV*<Long, *Iterable*<String>>> batchedData =

    sensitiveData

        .apply("Assign data into partitions",

            ParDo.of(**new** *DoFn*<String, *KV*<Long, String>>() {

*Random* random = **new** Random();

              @*ProcessElement*

              public *void* assignRandomPartition(*ProcessContext* *context*) {

                context.output(

                  KV.of(randomPartitionNumber(), context.element()));

              }

              private static *int* randomPartitionNumber() {

                return random.nextInt(numPartitions);

              }

            }))

        .apply("Create batches of sensitive data",

            GroupIntoBatches.<Long, String>ofSize(100L));

// Use batched sensitive data to fully utilize Redaction API

// (which has a rate limit but allows large payloads)

batchedData

    .apply("Call Redaction API in batches", callRedactionApiOnBatch());

## **Identificar problemas de performance causadas pela fusão de atividades pelo Dataflow**

O Dataflow constrói um gráfico de passos que representam sua pipeline, baseado nas transformações e atividades que foram construídas.

Ao realizar o deploy da sua pipeline, o Dataflow pode modificar sua pipeline e realizar fusões de algumas operações para melhorar a performance. Esse processo é conhecido como [fusion optimization](https://cloud.google.com/dataflow/docs/guides/deploying-a-pipeline#fusion-optimization).

Em alguns casos o Dataflow pode de forma incorreta, escolher uma maneira de fundir operações da sua pipeline. Segue o exemplo abaixo.

Diagrama

Descrição gerada automaticamente

Ao observar a imagem acima, é esperado que a etapa Increment Number seja realizada de forma rápida. Porém, as 2 etapas foram unificadas e não vai ser possível executar as etapas de forma independente.

É possível verificar quais passos foram realizados a fusão, com o comando abaixo

gcloud dataflow jobs describe --full ***job-id*** --format json

Os objetos que foram realizados a fusão, estão dentro de ExecutionStageSummary e no array de ComponentTransform.

Uma das formas de evitar a fusão, é o uso da transformação [GroupByKey](https://beam.apache.org/releases/javadoc/current/org/apache/beam/sdk/transforms/GroupByKey.html) e desagrupar antes de cada etapa, com o exemplo a seguir:

sequence

    .apply("Map Elements", MapElements.via(**new** *SimpleFunction*<Long, *KV*<Long, Void>>() {

      public *KV*<Long, Void> apply(*Long* *n*) {

        return KV.of(n, null);

      }

    }))

    .apply("Group By Key", GroupByKey.<Long, Void>create())

    .apply("Emit Keys", Keys.<Long>create())

    .apply("Find Primes Less-than-N", ParDo.of(**new** FindLowerPrimesFn()));

Outro método é o uso do [Reshuffle](https://beam.apache.org/releases/javadoc/2.34.0/org/apache/beam/sdk/transforms/Reshuffle.html), porém ele não é reutilizável em outros runners.

## **Métricas do Beam**

As métricas do Beam é uma classe utilitária que produz várias métricas para reportar algumas características da sua pipeline. Ao utilizar o Cloud Monitoring, é possível visualizar as métricas como uma [custom metrics](https://cloud.google.com/dataflow/docs/guides/using-cloud-monitoring#custom_metrics) (também chamado de User Counters)

Segue um exemplo de uso dessas métricas:

final *TupleTag*<String> errorTag = **new** *TupleTag*<String>(){};

final *TupleTag*<MockObject> successTag = **new** *TupleTag*<MockObject>(){};

final class ParseEventFn extends *DoFn*<String, MyObject> {

  private final *Counter* malformedCounter = Metrics.counter(ParseEventFn.class, "malformedJson");

  private final *Counter* emptyCounter = Metrics.counter(ParseEventFn.class, "emptyPayload");

  private *Gson* gsonParser;

  @*Setup*

  public setup() {

    gsonParser = **new** Gson();

  }

  @*ProcessElement*

  public *void* processElement(*ProcessContext* *c*) {

    try {

*MyObject* myObj = gsonParser.fromJson(c.element(), MyObject.class);

      if (myObj.getPayload() != null) {

        //  Output the element if non-empty payload

        c.output(successTag, myObj);

      }

      else {

        // Increment empty payload counter

        emptyCounter.inc();

      }

    }

    catch (*JsonParseException* *e*) {

      // Increment malformed JSON counter

      malformedCounter.inc();

      // Output the element to dead letter queue

      c.output(errorTag, c.element());

    }

  }

}

Referências

<https://beam.apache.org/documentation/programming-guide/>

<https://cloud.google.com/pubsub/docs/pubsub-dataflow>

<https://beam.apache.org/documentation/pipelines/design-your-pipeline/>

<https://beam.apache.org/documentation/pipelines/test-your-pipeline/>