GCP ETL

Cloud Data Fusion & Cloud Dataflow hands-on

T1 2023

**Table des matières**

[Introduction et objectifs 4](#_Toc132368993)

[Définition du use case 4](#_Toc132368994)

[ETL avec Cloud Data Fusion 5](#_Toc132368995)

[Configuration dans GCP 5](#_Toc132368996)

[Développement de l'ETL 6](#_Toc132368997)

[Déploiement de l'ETL 12](#_Toc132368998)

[Test\_1: pipeline exécutée avec cluster personnalisé 12](#_Toc132368999)

[Test\_2: pipeline exécutée avec cluster prédéfinit 12](#_Toc132369000)

[Test\_2.b: Cluster prédéfini en autoscaling 12](#_Toc132369001)

[ETL avec Cloud Dataflow 14](#_Toc132369002)

[Développement de la pipeline avec Apache Beam 14](#_Toc132369003)

[S'agissant des imports 14](#_Toc132369004)

[S'agissant des classes 14](#_Toc132369005)

[S'agissant de l'initiation de la pipeline et de sa configuration 15](#_Toc132369006)

[S'agissant de la pipeline 15](#_Toc132369007)

[Déploiement de la pipeline avec Cloud Dataflow 15](#_Toc132369008)

[CLI - en ligne de commande 16](#_Toc132369009)

[Workbench - Jupyter Lab 17](#_Toc132369010)

[Templates 18](#_Toc132369011)

[Mise en œuvre avec passage à l'échelle 21](#_Toc132369012)

[1er test - échantillon 21](#_Toc132369013)

[2ème test – premier passage à l’échelle – 1 To 21](#_Toc132369014)

[3ème test – réajustement de la taille du je de données, considérant les limites d’un compte gratuit 22](#_Toc132369015)

[a) une table de 8GB (pour 36M de lignes) 22](#_Toc132369016)

[b) une table de 26GB (pour 111M de lignes) 23](#_Toc132369017)

[c) même table avec des machines plus performantes 24](#_Toc132369018)

[Conclusion comparative 25](#_Toc132369019)

[Facilité de développement vs. Flexibilité 25](#_Toc132369020)

[Data Fusion 25](#_Toc132369021)

[Dataflow 26](#_Toc132369022)

[Scalabilité 27](#_Toc132369023)

[Dataflow 27](#_Toc132369024)

[Data Fusion 27](#_Toc132369025)

[Coût 28](#_Toc132369026)

[Data Fusion 28](#_Toc132369027)

[Dataflow 30](#_Toc132369028)

[Bonus : consommation des données dans Looker 32](#_Toc132369029)

# Introduction et objectifs

Ce document décrit les travaux réalisés pour la mise en œuvre d’ETL en utilisant les solutions Cloud Data Fusion et Cloud Dataflow sur GCP.

L'objectif principal est d'éprouver ces solutions pour acquérir un recul suffisant sur leurs capacités et limites, notamment vis-à-vis d'autres solutions (*on-premise* et sur cloud), et selon les différents cas d'usage qui peuvent exister (en fonction des diverses situations clients).

# Définition du use case

Pour expérimenter ces deux solutions et les confronter, un cas d'usage est identifié ci-dessous. Il vise à réaliser un ETL sur une table à l'échelle (idéalement plusieurs dizaines de To).

1. **la Table** : la table retenue est disponible dans les tables publiques BigQuery. Il s'agit de la table *gdeltv2.webngrams*. Elle recense toutes les séquences de mots publiées sur internet, ainsi que le contexte associé. Elle fait environ 70 To.
2. **l'ETL** : il s'agit de filtrer la table dans un premier temps sur les séquences de mots en langue anglaise. Dans un deuxième temps, opérer une transformation/regroupement par séquence de mot et par date de manière à pouvoir suivre dans le temps la fréquence d'apparition des séquences de mots

*Bonus* : consommation des données dans une visualisation sur Looker Studio

# ETL avec Cloud Data Fusion

Data Fusion est la version managée de CDAP (Cask Data Application Platform) chez Google depuis 2018. CDAP est une plateforme de développement d’applications open source pour l’écosystème Hadoop. Elle fournit aux développeurs une virtualisation d’applications pour accélérer le développement d’applications, notamment les ETL en batch et en streaming.

L’objectif de ce pipeline est résumé dans ces trois étapes :

* Importation d’une table publique du BigQuery
* Effectuer des transformations sur les données
* Sauvegarder les données transformées dans BigQuery.

## Configuration dans GCP

Avant la création de l’ETL, il faut configurer le compte de services dans GCP et activer les APIs nécessaires.

Les APIs qui doivent être activées sont:

* Cloud Dataproc API
* Cloud Data Fusion API
* Compute Engine API
* Cloud Dataproc Control API
* BigQuery API
* Cloud Logging API

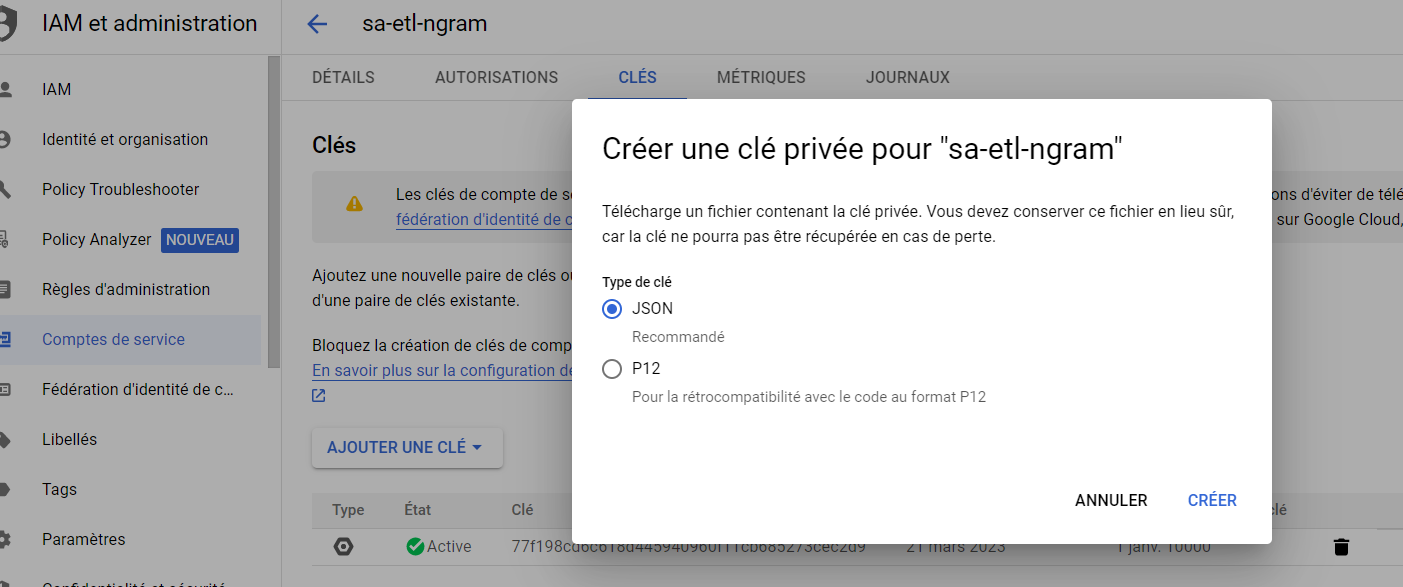
Le compte de service va permettre à l'ETL de se connecter aux ressources nécessaires pour exécuter le workflow. Pour ce faire, il faut suivre ces étapes :

IAM & Administration à Comptes de service à Créer --> Insérer le nom du compte de service --> sélectionner le bouton "créer et continuer".

Il faut par la suite, donner ces rôles nécessaires à ce compte de service. Ces rôles sont :

* BigQuery Admin
* BigQuery Data Editor
* Cloud Data fusion Runner
* Dataproc Administrator
* Dataproc Editor
* Dataproc Worker
* Editor
* Storage Admin

Une fois c'est fait, il faut enregistrer et valider la création. Ensuite, il faut générer une clé Json qu'on intègrera après dans la configuration des connections dans l'ETL.



Note: il faut enregistrer la clé pour l'utiliser après parce qu'une fois créée, on ne peut pas y accéder de nouveau! Si jamais elle est perdue, vous avez toujours la possibilité de créer une autre.

## Développement de l'ETL

Pour créer l'ETL, il est possible de suivre ces deux méthodes :

* Implémenter sur CDAP
* Implémenter sur Data Fusion

Pour qu'on puisse développer gratuitement notre ETL, il est conseillé d'utiliser CDAP. On a utilisé l'[image Docker de CDAP](https://hub.docker.com/r/caskdata/cdap-sandbox). Il faut tout d'abord importer l'image ensuite créer un container pour le run.

* importer l'image:

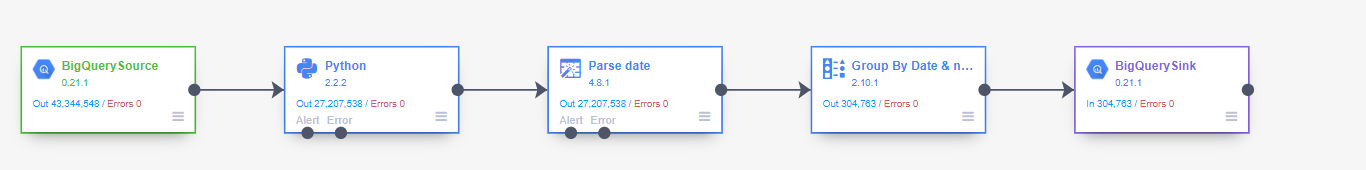
docker pull caskdata/cdap-sandbox

* run de l'image

docker run -it -p 11015:11015 -p 11011:11011 -v cdap-app:/opt/cdap/sandbox/my/own/datadir --name cdap-sandbox caskdata/cdap-sandbox cdap sandbox start --foreground

Pour se connecter à l'interface CDAP, se rendre sur le navigateur et taper: <http://localhost:11011/>. Choisir "Studio" dans la composante "Integrate". A partir de liste déroulante en haut à gauche, choisir "Data Pipeline - Batch".

l'ETL qu'on va créer est la suivante:



Voici le lien vers le fichier Json du pipeline. Il suffit de l’importer dans l’instance Data Fusion ou bien dans CDAP, pour pouvoir tester le pipeline. Les connexions vers bigquery sont à reconfigurer selon votre projet Google Cloud. [Data\_Fusion.json](https://github.com/polymoe/Inetum/tree/main/data_fusion_test)

Nous allons décrire les configurations sur les différentes composantes :

**BigQuery Source**

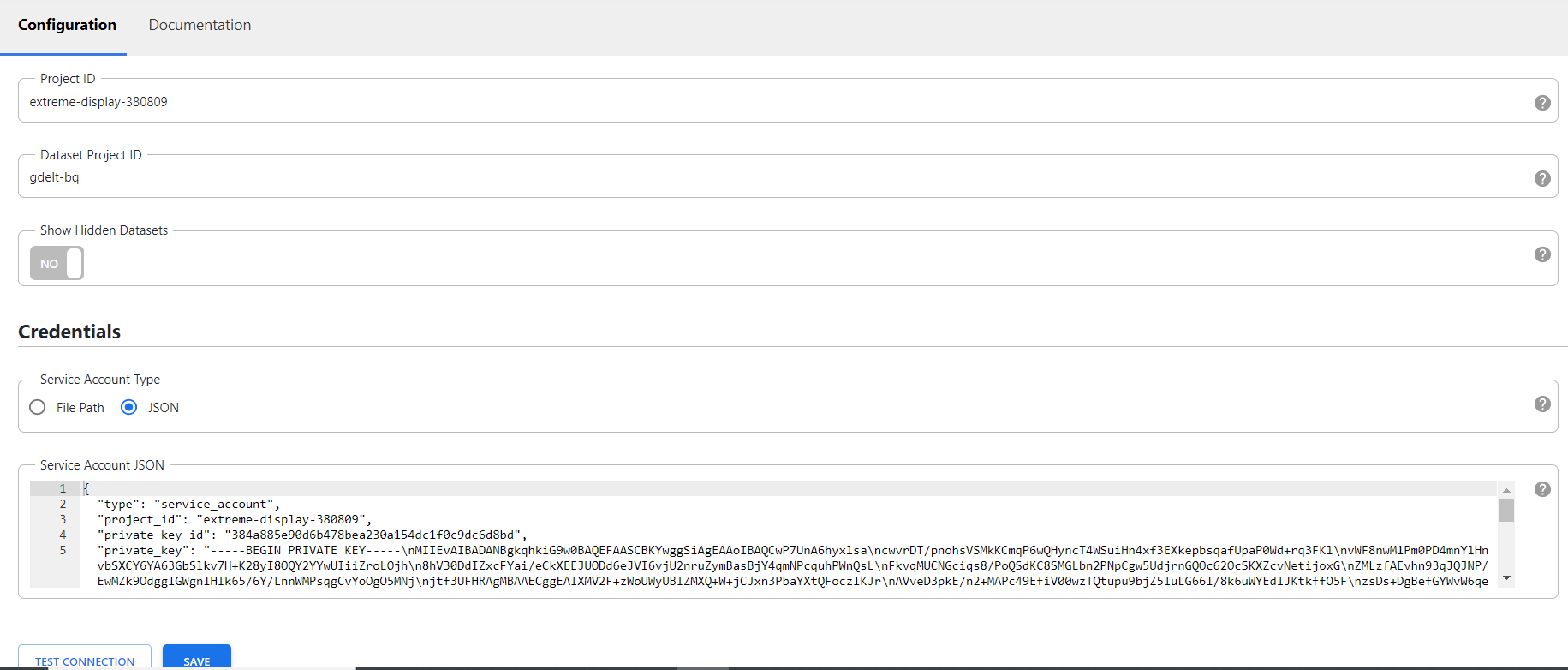
Dans l'onglet "Source", choisir "BigQuery". La première composante sera insérée.

Cliquer sur "properties". Renseigner les champs selon ce qui est décrit dans la documentation. Activer l'option "Use Connection" , ensuite dans "Connection", cliquer sur "BROWSE CONNECTIONS" --> "ADD CONNECTION".  Nous allons maintenant créer la connexion entre CDAP et Bigquery dans GCP. Choisir "Google Cloud Platform" --> BigQuery.

Note:

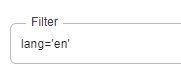
Dans notre use case, nous avons travaillé sur des données publiques assez volumineuse (70 Po) avec environ 400 milliard de lignes ([lien](https://console.cloud.google.com/bigquery?project=extreme-display-380809&ws=!1m5!1m4!4m3!1sgdelt-bq!2sgdeltv2!3swebngrams)). A noter que les ressources sont restreintes selon les [limitations des quotas](https://cloud.google.com/bigquery/quotas?hl=fr) préfixées par GCP ( Taille maximale par tâche de chargement est de 15 To). Les quotas quotidiens sont réapprovisionnées à intervalles réguliers tout au long de la journée, de sorte à inciter les utilisateurs à limiter leur taux de requêtes. Pour cette raison, on a travaillé avec un échantillon d'environ 112 millions de lignes (20 GO).

Dans "Dataset Project ID", il faut mettre l'identifiant du dataset contenant l'échantillon. Dans "Service Account JSON", il faut coller la clé Json de service account qu'on vient de créer et sauvegarder --> save --> cliquer sur "ADD CONNECTION".



On clique sur "BROWSE" pour choisir la base et la table souce  --> "cliquer sur "validate".

Dans le champ "filter", On peut mentionner le critère de filtrage pour n'extraire de la table source que les lignes dont on a besoin. Nous décidons de ne garder que les ngram des articles en langue anglaise.



**Python**

Nous avons besoin d'effectuer des opérations de nettoyage sur les données extraites. Notre but c'est de filtrer les ngram qui contiennent des stop-words tel que the, to , be, as, etc. Pour ce faire, on a besoin d'un plugin Python Transform, nous permettant d'écrire un code python. Il peut être déployé à partir de l'onglet "HUB".  [Documentation python transform](https://cdap.atlassian.net/wiki/spaces/DOCS/pages/694157596/Python+Evaluator+Transformation).

Dans ce plugin, il y a deux modes d'exécution: Interpreted et Native.

* *Native:*Avec ce mode, n’importe quelle version de Python et les bibliothèques installées sur les exécuteurs peuvent être utilisées. Le mode natif nécessite que python soit disponible sur chaque nœud de votre cluster. Il nécessite également que la bibliothèque py4j soit disponible sur chaque nœud.
* Interpreted: Le code Python est exécuté via jvm, donc les bibliothèques basées sur C (par exemple numpy) et la syntaxe Python3 ne sont pas prises en charge

Voici notre script Python:



*Note : Dans notre use case, nous avons besoin d'installer la bibliothèque  NLTK de ML pour effectuer le nettoyage. Nous avons donc essayer avec le mode "Native", mais on a rencontré des problèmes pour que le plugin connait l'interpréteur python qu'on a référencé dans ". Il y a avait très peu d'exemple sur Internet et selon des blogs, ce mode d'exécution dans le plugin présente un bug (à vérifier avec les prochaines versions. A noter que la version utilisée est 2.2.2)*

Nous avons ensuite essayé avec le mode "Interpreted".  Pour ce faire, nous devons personnaliser le cluster Dataproc, qui va exécuter le pipeline pour qu'il ne supporte que Python2. (image-version=1.5).

Il faut aussi créer un bucket qui contient le script d'initialisation des différents noeuds afin d'installer NLTK.

Le script "*myscript.sh"* est le suivant :

#!/bin/bash

if pip freeze | grep NLTK; then

    # NLTK is already installed

    true

else

    # nltk is not installed yet; install the latest from PyPI

    pip install NLTK==3.4.5

fi

Ce script doit être sauvegardé dans un bucket dans Google Cloud Storage. On l'a appelé : "*my\_dataproc\_init\_script"*

La commande pour créer ce cluster Dataproc est la suivante :

 gcloud dataproc clusters create mydfcluster \

    --image-version=1.5 \

    --service-account=sa-etl-ngram@extreme-display-380809.iam.gserviceaccount.com \

    --region=us-east1 \

    --metadata='MINICONDA\_VARIANT=2' \

    --metadata='MINICONDA\_VERSION=latest' \

    --initialization-actions=gs://goog-dataproc-initialization-actions-us-central1/conda/bootstrap-conda.sh,gs://my\_dataproc\_init\_script/myscript.sh \

    --properties=^#^dataproc:conda.packages='pytorch==1.0.1,visions==0.7.1'#dataproc:pip.packages='tokenizers==0.10.1,datasets==1.5.0'

Une fois notre cluster est créé, il doit être ajouté au System Compute Profiles de Data Fusion :

SYSTEM ADMIN --> Configurations --> System Compute Profiles --> Create new profile --> existing Dataproc --> remplir les données --> Create.

Le cluster personnalisé doit être affecté comme cluster par défaut.

**Wrangler**

Nous allons effectuer des transformations sur les données importées.

Cliquer sur "Properties" --> "WRANGLE" --> choisir les étapes de transformations.

Les transformations qu'on a fait sont les suivantes :

1) Filtrage sur la colonne "Lang": filtrer les lignes qui ne sont pas en anglais

2) Parse sur la colonne "date" : parser le type vers simple date() pour ne garder que l'année, mois et le jour et supprimer l'heure.

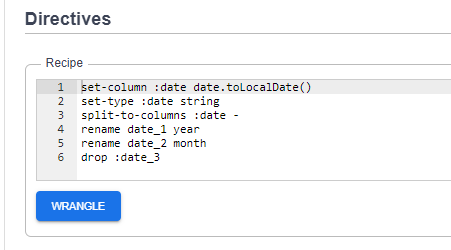
3) Parse sur la colonne "date" : modifier le type en string

4) Extract sur la colonne "date" : extraire les champs selon le délimiteur "-".

5) Renommer la colonne générée "date\_1" --> year

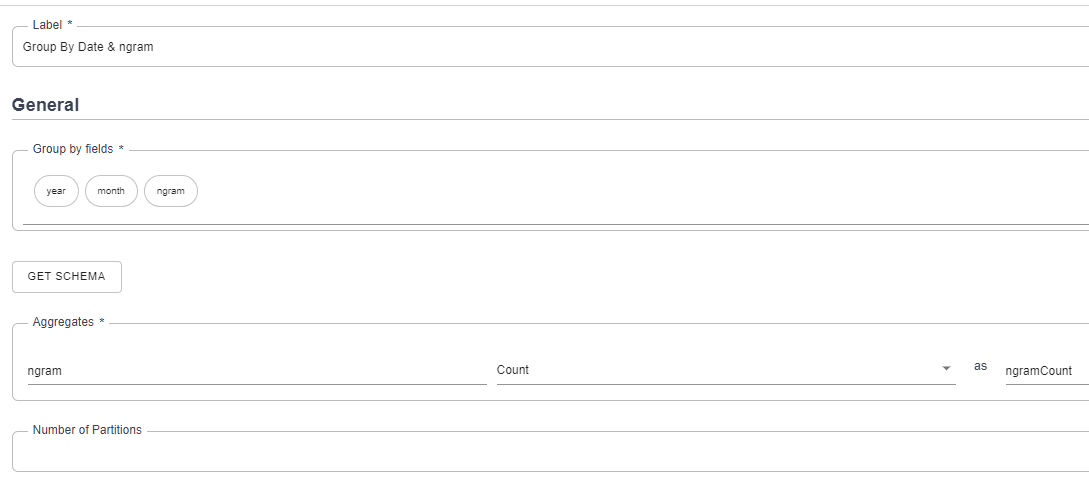
6) Renommer la colonne générée "date\_2é --> month

7) Supprimer la colonne date\_3 qui contient le jour



**Group by**

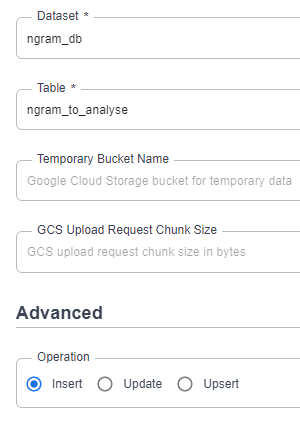
 Dans cette composante, nous allons regrouper les ngram (mots) par mois et par années. Ensuite nous allons calculer le nombre des ngram pour chaque mois de l'année.  L'objectif est d'étudier l'apparition de ces mots dans le temps. Voici la configuration qu'on a faite :

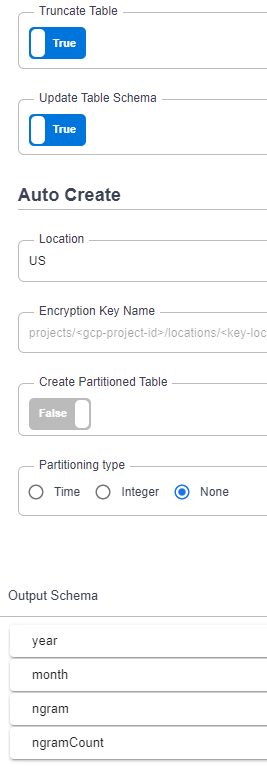


**BigQuery Sink**

 Dans cette dernière composante de l'ETL, nous allons spécifier la table dans laquelle on va stocker les données transformées. Pour ce faire, nous allons utiliser la même connexion que celle utilisée dans la composante "BigQuerySource". Il faut aussi spécier le nom de la table sink. Si elle n'existe pas, elle sera crée. Voici la configuration qu'on a faite :

|  |
| --- |
|  |



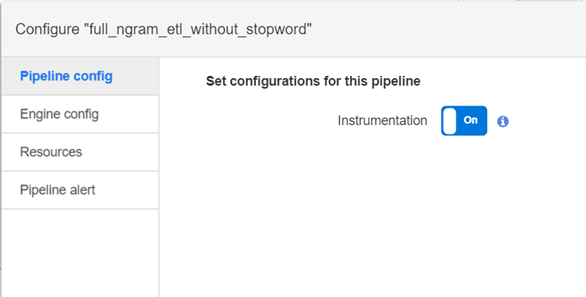


Note: Dans le but d'optimiser les traitements analytiques sur la table sink qui va être crée dans BigQuery, il est conseillé  d'activer l'option "Partitionnement de la table".

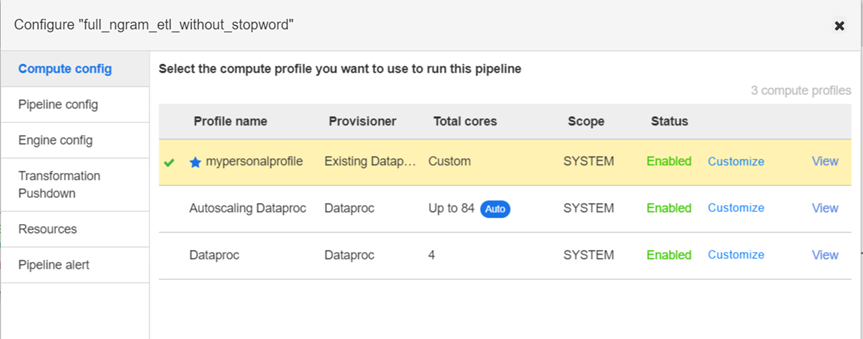
## Déploiement de l'ETL

### Test\_1: pipeline exécutée avec cluster personnalisé

Nous avons la possibilité, pour tester le fonctionnement de notre pipeline d'utiliser le mode "*preview*". Avec ce mode , on a accès à la configuration du cluster. Là on remarque qu'on a pas accès à l'option *Compute Config* qui permet de choisir avec quel cluster on va exécuter notre pipeline. Voici ce qu'on peut avoir en mode preview :



Pour pouvoir tester avec le cluster personnalisé, il faut déployer le pipeline avant. Ainsi, on  pourra avoir accès à ces configurations:



Une fois déployée, le pipeline a été planté à l'étape d'initialisation du cluster sans aucune erreur sortie dans le log!

J'ai arrêté le déploiement après deux heures d'attente.

### Test\_2: pipeline exécutée avec cluster prédéfinit

En utilisant le cluster prédéfinit, on ne peux pas y importer les bibliothèques dont on a besoin. Même en l'initialisant comme on a fait dans le test précédent, le pipeline a planté à l'étape d'initialisation. Peut être c'est dû aux limites de quotas pour le compte gratuit.

*Note: Test à refaire avec la version payante de la console Google Cloud.*

### Test\_2.b: Cluster prédéfini en autoscaling

Dans l'option d'autoscaling, c'est à nous de gérer la scalabilité verticale (performance de la machine) et c'est Google qui gère la scalabilité horizontale (nombre de workers). Donc, nous devons faire des tests pour trouver la meilleure configuration pour le master et les workers.

Nous pouvons personnaliser la configuration des ressources pour le cluster prédéfinit, toujours dans l'étape de déploiement et non pas en mode "preview". Nous avons testé les configurations suivantes :

* Machine n1-standard-1 : nombre Cores=1 ; Mémoire: 5 GB

Pipeline failed --> Requested resource=<memory:2432, vCores:3>, maximum allowed allocation=<memory:4095, vCores:1>

* Machine n1-standard-2 :  nombre Cores=2; Mémoire: 8 GB

pipeline failed/  Requested resource type=[vcores], Requested resource=<memory:2432, vCores:3>, maximum allowed allocation=<memory:6554, vCores:2>

* Machine n1-standard-4 :

Notre pipeline a été exécutée avec succès après 23 mins 38 secs.

# ETL avec Cloud Dataflow

Il est tout d'abord indiqué qu'un repo Github dédié centralise les scripts et fichiers utilisés dans les travaux sur Dataflow. Il se trouve [ici](https://github.com/polymoe/Inetum/tree/main/dataflow_test).

Dataflow n'est en réalité qu'un type (parmi d'autres) de *runner* de la pipeline rédigée avec le framework Apache Beam.

Par conséquent, ci-dessous, nous allons :

1. dans un premier temps présenter le script Python (utilisant la librairie *apache\_beam*) qui implémente notre pipeline d'ETL.
2. Nous allons ensuite voir les différentes manières d'exécuter cette pipeline dans Dataflow.

## Développement de la pipeline avec Apache Beam

Nous décrivons ici le script python qui implémente notre ETL. Celui-ci est disponible dans le repository Github consacré, [ici](https://github.com/polymoe/Inetum/blob/main/dataflow_test/gcp/main/main.py).

Le script se compose des parties suivantes :

* les imports
* les classes qui définissent les transformations appliquées dans l'ETL
* l'initiation de la pipeline et la préparation de sa configuration
* la pipeline

### S'agissant des imports

Ils concernent principalement :

* *argparse*: pour *parser* les arguments de la commande d'exécution de la pipeline (en ligne de commande), et déterminer les options de la pipeline
* *apache\_beam*: pour importer la librairie Apache Beam pour écrire notre pipeline
* ‘*Coder*’ de *apache\_beam.coders* : pour convertir les entrées avec le bon codage qui peut être géré par apache beam
* *datetime* : pour les conversions des dates au bon format
* *nltk*: pour filtrer les mots en input (anglais) en retirant les stopwords
* *re* : pour utiliser les "regex" pour appliquer des transformations aux mots en entrée (ex: suppression des caractères spéciaux)
* *PipelineOptions* : pour spécifier les options de la pipeline
* *bigquery* de *apache\_beam.io.gcp.internal.clients* : pour pouvoir requêter BigQuery depuis Apache Beam

### S'agissant des classes

Les classes suivantes sont implémentées :

* *ISOCoder* : pour lire et écrire des strings au format ISO-8859-1
* *FilterByLang* : pour filtrer les inputs et ne garder que les inputs en langue anglaise
* *FilterStopwords* : pour filtrer les stopwords en utilisant la librairie nltk et des transformations utilisant les regex
* *Format\_to\_dict* : pour transformer les output au bon format (dictionnaire) pour transfert vers BigQuery

### S'agissant de l'initiation de la pipeline et de sa configuration

Elle passe par l'utilisation de *argparse.ArgumentParser()*.

On définit alors les paramètres (connus) que nous utilisons explicitement dans notre code, et ceux (les autres) qui sont passés en argument de la PipelineOptions, et définissent donc les options de la pipeline. Parmi les paramètres explicites, nous avons:

* *--project* : l'identifiant du projet
* *--dataset* : l'identifiant du dataset (bigquery)
* *--tablein* : l'identifiant de la table (bigquery) des inputs
* *--tableout* : l'identifiant de la table (bigquery) à créer en output de la pipeline

Nous définissons ensuite les spécifications des tables d'input et d'output (et le schéma de cette dernière).

Enfin, nous définissons les stopwords en anglais.

### S'agissant de la pipeline

Nous écrivons enfin notre pipeline ETL, en réalisant successivement :

* la lecture des inputs dans la table d'input Bigquery
* le filtrage des lignes de la table qui ne sont pas en langue anglaise
* le filtrage des stopwords en langue anglaise (et caractères spéciaux)
* le comptage des mots
* la conversion des données de sortie de l'ETL au format dictionnaire
* l'écriture dans une table Bigquery de sortie

## Déploiement de la pipeline avec Cloud Dataflow

Une fois la pipeline rédigée dans le framework Apache Beam, il est possible de l'exécuter de différentes manières dans GCP Dataflow.

Au préalable, il est nécessaire de :

* créer un projet dans GCP
* activer les différentes API nécessaires à l'exécution du projet
* octroyer les droits nécessaires aux ressources mobilisées

Pour ce qui est des API, il faut activer :

* Dataflow API
* Compute Engine API
* loud Logging API
* loud Storage
* Google Cloud Storage JSON API
* BigQuery API
* Cloud Datastore API
* Cloud Resource Manager API

Ensuite, dans IAM, octroyer les rôles suivants pour le compte de service Compute Enginer par défaut :

* Administrateur Dataflow
* Administrateur des objets Storage
* Administrateur Storage
* Éditeur
* Éditeur de données BigQuery
* Lecteur Artifact Registry
* Nœud de calcul Dataflow

Une fois ces prérequis satisfaits, il est possible d'utiliser l'une des manières suivantes pour exécuter la pipeline dans Dataflow.

### CLI - en ligne de commande

Avec cette méthode, on utilise les lignes de commande du Cloud Shell GCP pour :

* créer un environnement virtuel python
* installer dedans le SDK Apache Beam
* exécuter le job avec un runner dataflow

Installez le module virtualenv, créez un environnement virtuel, puis activez-le :

pip3 install virtualenv python3 -m virtualenv env source env/bin/activate

Installez le SDK Apache Beam :

pip3 install apache-beam[gcp]

Il ne faut pas oublier de créer un Bucket dans Cloud Storage

Dans Cloud Shell exécuter le script python, en lui indiquant dans les paramètres :

* le nom du fichier des données d'entrée à traiter, que l'on aura préalablement mis dans Cloud Storage (dans le cas d'un script qui prend un fichier csv dans Cloud Storage et qui crée un fichier CSV traité en sortie également dans cloud storage. Nous verrons ultérieurement le scénario qui manipule des tables BigQuery en entrée et en sortie de pipeline)
* le nom du fichier de sortie à créer dans cloud storage (sans mettre .csv)

python3 -m main \  
--input gs://dflowtestbucket/raw\_data\_webngrams.csv \  
--output gs://dflowtestbucket/filtered\_data\_webngrams\  
--runner DataflowRunner \  
--project dataflowtest-380909 \  
--temp\_location gs://dflowtestbucket/tmp/

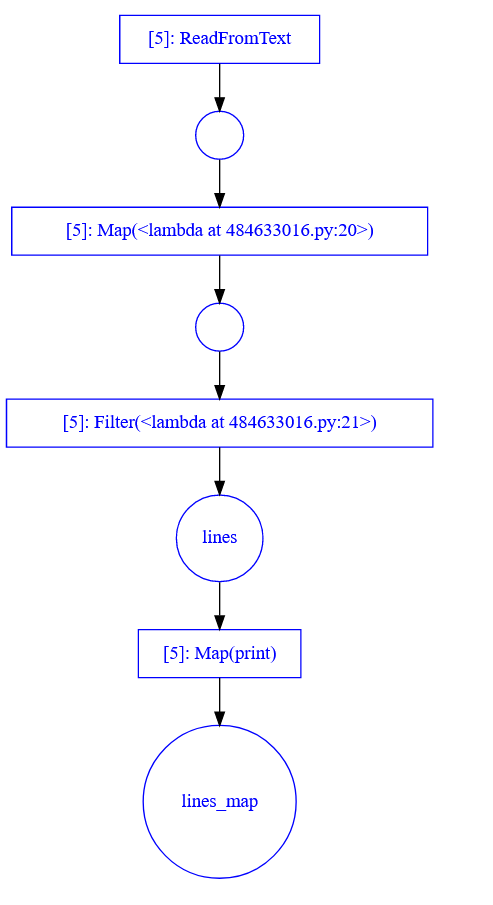
On peut alors vérifier que le fichier output filtered\_data\_webngrams.csv a bien été créé dans le bucket dans Cloud Storage.

### Workbench - Jupyter Lab

En allant dans la section "Workbench" de Dataflow sur GCP, il est possible de créer un nouveau Notebook Jupyter Lab.

Il suffit alors de copier coller notre script dans le notebook et de l'exécuter.

Une fois exécuté, il est possible d'ouvrir un onglet "Open Inspector" pour visualiser le graphe de la pipeline:



### Templates

En allant dans la section "Pipelines" de Dtaflow sur GCP, il est possible de créer une pipeline en se basant sur des templates préexistants.

Ces templates ne fournissent qu'une structure macro généralement de cas d'usages fréquents avec GCP. Quelques exemples :

|  |
| --- |
|  |

Il est néanmoins possible de créer son propre template adapté à son ETL. C'est ce que nous avons fait.

L'approche est la suivante :

* créer une image docker de notre pipeline et des metadata associées (le format des paramètres que l'on passerait en ligne de commande pour exécuter la pipeline : i.e les noms des fichiers/tables d'entrée sortie de la pipeline)
* stocker l'image docker dans Artifact Registry
* génération du fichier json de notre modèle (FLEX) de transformation, qui sera exécuté lors du job Dataflow

Il faut veiller à ce que les API suivantes soient activées :

* Dataflow
* Compute Engine
* Logging
* Cloud Storage
* Cloud Storage JSON
* BigQuery
* Resource Manager
* App Engine
* Artifact Registry
* Cloud Scheduler
* Cloud Build

Attribuer les rôles adéquats avec les commandes suivantes :

gcloud projects add-iam-policy-binding dataflowtest-380909 --member="user:toumi.msh@gmail.com" --role=roles/iam.serviceAccountUser

gcloud projects add-iam-policy-binding dataflowtest-380909 --member="serviceAccount:823307351858-compute@developer.gserviceaccount.com" --role=roles/dataflow.admin  
gcloud projects add-iam-policy-binding dataflowtest-380909 --member="serviceAccount:823307351858-compute@developer.gserviceaccount.com" --role=roles/dataflow.worker  
gcloud projects add-iam-policy-binding dataflowtest-380909 --member="serviceAccount:823307351858-compute@developer.gserviceaccount.com" --role=roles/bigquery.dataEditor  
gcloud projects add-iam-policy-binding dataflowtest-380909 --member="serviceAccount:823307351858-compute@developer.gserviceaccount.com" --role=roles/storage.objectAdmin  
gcloud projects add-iam-policy-binding dataflowtest-380909 --member="serviceAccount:823307351858-compute@developer.gserviceaccount.com" --role=roles/artifactregistry.reader

Ensuite, il faut créer un repository dans Artifact Registry, pour y déposer notre image docker.

gcloud artifacts repositories create dataflowflexmodelrepo \  
    --repository-format=docker \  
    --location=us    \  
    --async

Pour créer notre image docker, nous allons créer un Dockerfile. Le script est disponible [ici](https://github.com/polymoe/Inetum/tree/main/dataflow_test/gcp/template) sur le Github.

Il embarque notre fichier python qui définit la pipeline avec apache beam, ainsi qu'un fichier requirements.txt

Pour des raisons d'efficacité/vitesse d'exécution, il est recommandé de ne pas inclure les packages apache\_beam dans requirements.txt, mais de l'installer via les commandes appropriées dans le Dockerfile.

Ensuite, pour configurer l'authentification auprès des dépôts Docker, exécutez la commande suivante :

gcloud auth configure-docker us-docker.pkg.dev

Ensuite, il faut créer l'image Docker à l'aide du fichier Dockerfile avec Cloud Build :

gcloud builds submit --tag us-docker.pkg.dev/dataflowtest-380909/dataflowflexmodelrepo/dataflow/flex\_dataflow\_model\_test:latest .

Il est ensuite possible d'ajouter au modèle des métadonnées afin de valider des paramètres personnalisés lors de l'exécution du modèle. Si vous souhaitez créer des métadonnées pour votre modèle, il faut créer un fichier metadata.json comme indiqué sur le [github](https://github.com/polymoe/Inetum/tree/main/dataflow_test/gcp/template" \t "_blank" \o "https://github.com/polymoe/inetum/tree/main/dataflow_test/gcp/template).

Enfin, avec ces éléments, nous créons notre modèle FLEX (fichier json) dans Cloud Storage. Ce modèle (fichier json) contiendra l'information sur l'image docker à utiliser ainsi que sur les medatas associées à notre pipeline.

gcloud dataflow flex-template build gs://dflowtestbucket/samples/dataflow/templates/beam-dataflow-model-test.json \  
--image "us-docker.pkg.dev/dataflowtest-380909/dataflowflexmodelrepo/dataflow/flex\_dataflow\_model\_test:latest" \  
--sdk-language "PYTHON" \  
--metadata-file "metadata.json"

Finalement, pour exécuter notre pipeline en faisant appel à notre modèle:

gcloud dataflow flex-template run "flex-dataflow-template-test-`date +%Y%m%d-%H%M%S`" \  
--template-file-gcs-location "gs://dflowtestbucket/samples/dataflow/templates/beam-dataflow-model-test.json" \  
--parameters input='gs://dflowtestbucket/raw\_data\_webngrams.csv' \  
--parameters output='gs://dflowtestbucket/filtered\_data\_webngrams' \  
--region "us"

### Mise en œuvre avec passage à l'échelle

Après avoir testé la pipeline en utilisant des csv en input et output dans un bucket Cloud Storage, la partie finale des travaux dataflow consiste à appliquer l'ETL à BigQuery, en lisant une table en input, et en créant une table transformée en output.

#### 1er test - échantillon

Un premier test a été réalisé avec un échantillon de table de 200 lignes. Pour rappel, la table originale fait 70To pour  320 milliards de lignes.

**Résultat : OK**

#### 2ème test – premier passage à l’échelle – 1 To

Après la réussite de cette première étape, nous avons essayé un premier passage à l'échelle avec une échantillon de, 1To de notre table.

Résultat : pipeline OK - **scalabilité KO**

La scalabilité automatique n'a pas fonctionné, après de nombreuses tentatives de manipulation des options de la pipeline (forcer la scalabilité sur on/off, forcer le nombre de workers, forcer le nombre max de workers, etc.)

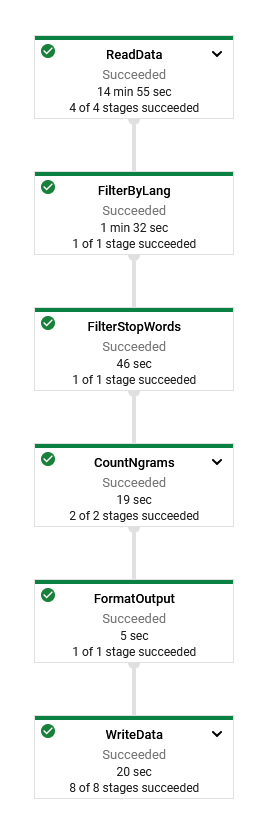
En effet, bien que le target workers (nombre requis pour exécuter le travail) calculé par Dataflow était de l'ordre de 400 workers, le current worker (le nombre de workers effectivement mobilisés) est resté égal à 1. Ainsi, au bout d 1,5h d'exécution, seulement 4% de notre échantillon a pu être lu par la pipeline (soit environ 150M de lignes).

#### 3ème test – réajustement de la taille du je de données, considérant les limites d’un compte gratuit

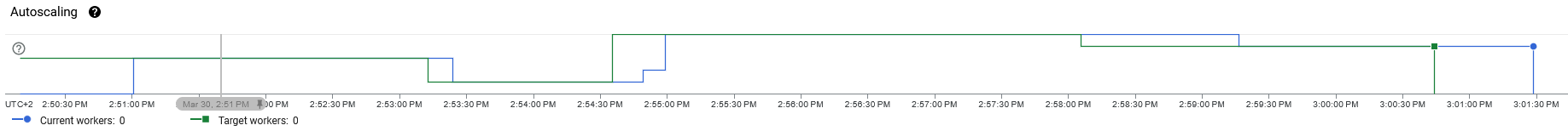
Après l'échec de la précédente tentative, nous avons été contraints de revoir la taille de notre échantillon à la baisse, afin de tenir compte (semble-t-il des limitations inhérentes à l’utilisation d’un compte gratuit). Trois essais ont été réalisés :

##### a) une table de 8GB (pour 36M de lignes)

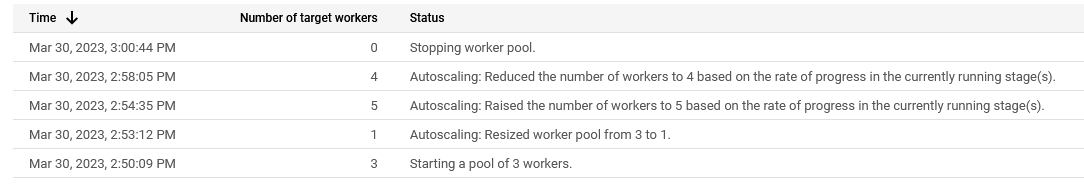
**Résultat OK**



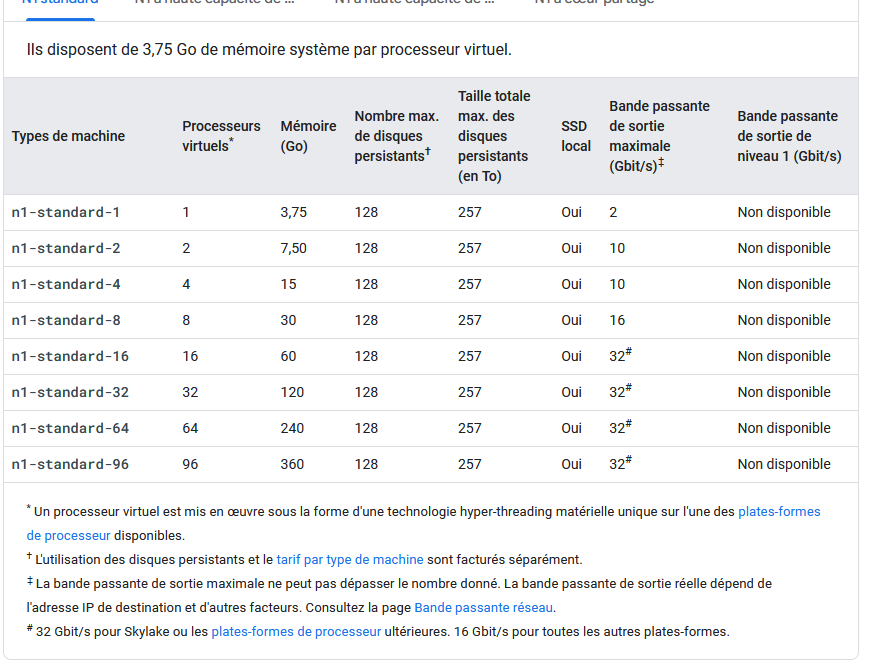
la pipeline a été exécutée avec succès, au bout de 11min30.



la pipeline a mobilisé (à son maximum) 5 workers (sur 5 workers demandés).



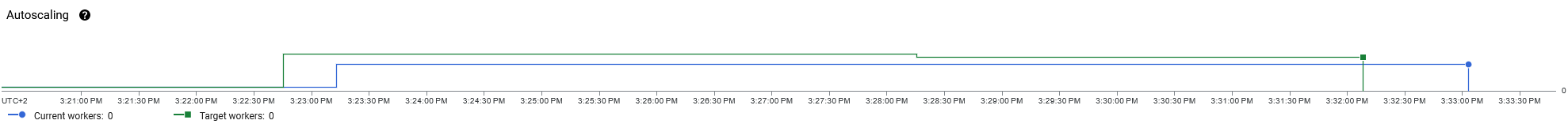
La configuration des workers déployés est : n1-standard-1 in us-west1-b.



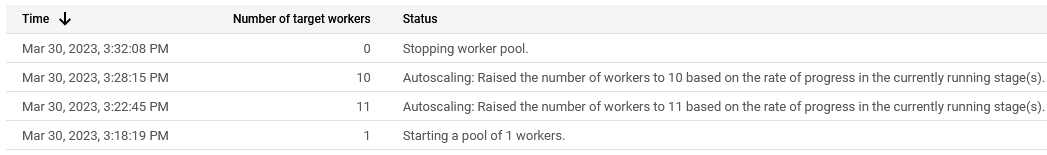
##### b) une table de 26GB (pour 111M de lignes)

**Résultat : Pipeline OK – Scalabilité KO**

**la pipeline a été exécutée avec succès, au bout de 14 min 55 sec.**

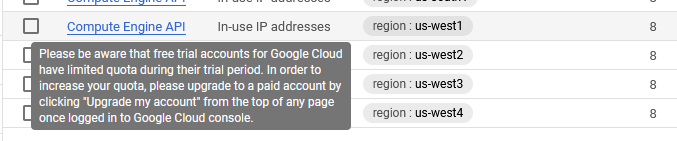


la pipeline a mobilisé (à son maximum) 8 workers (sur 11 workers demandés).



La configuration des workers déployés est : n1-standard-1 in us-west1-b (cf. table ci-dessus).

Il est apparu qu'un certain nombre de quotas son définis pour les différentes ressources utilisées. En particulier, un quotas de 8 adresses IP externes seulement est disponibles (pour les comptes gratuits). Il est possible de demander une extension, modulo passage à un compte payant.



##### c) même table avec des machines plus performantes

**Résultat : Pipeline OK – Scalabilité OK**

Nous avons dans ce cas forcé la famille de machines (plus performante) utilisée par la pipeline.

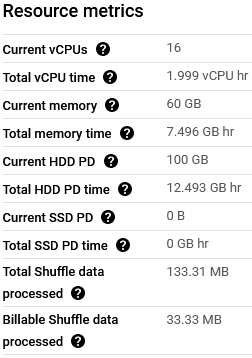
La pipeline s'est bien exécutée en 12 minutes.

Cette fois, au lieu de laisser Dataflow choisir la famille de workers à n1-standard-1, on le force à choisir des workers plus formants n1-standard-4, en passant ce paramètre en argument de PipelineOptions en ligne de commande.

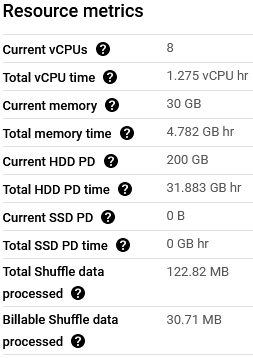
Cette fois, la pipeline n'a eu besoin que de 4 workers à son maximum, s'affranchissant ainsi de la contrainte du quotas de 8 adresses IP externes.

Comparaison des ressources entre ce cas et le cas précédent :

*Nouvelle configuration*



*Précédente configuration*

**

# Conclusion comparative

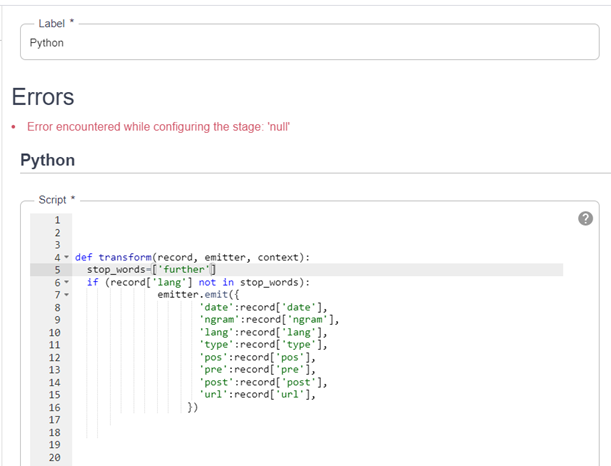
Dans cette partie, nous comparons de manière synthétique les deux solutions au regard des trois clés suivantes :

1. La facilité de développement vs. la flexibilité
2. La scalabilité
3. Le coût

## Facilité de développement vs. Flexibilité

### Data Fusion

* Avec Data Fusion, le développement d'un pipeline est très simple, ce sont des composantes graphiques qu'on va glisser et ensuite configurer. Pour les transformations à effectuer, il suffit de sélectionner le champ à transformer et choisir lui affecter un wrangler. Le code sera automatiquement généré. Donc c'est une solution très adaptée pour des clients qui ne disposent pas, parmi leurs ressources humaines, de développeurs et de spécialistes de domaine, et qui envisage créer des pipelines avec des transformations simples.
* Dans le cas où on veut appliquer une transformation personnalisée, c'est très compliqué de la mettre en place. Par exemple, Dans notre cas, lorsqu'on a utilisé le plugin Python Transformer, qui est le seul plugin jusqu'à maintenant supportant Python. Selon la documentation, il ne supporte pas Python3 et les bibliothèques basées sur C, ce qui est contraignant avec plusieurs use case.
* Dans notre cas d'étude, on n'a pas réussi à initialiser un cluster, lors du test de déploiement, avec des bibliothèques autres que celles déjà installées dans l'image. A noter qu'on a choisi la version d'image 1.5 qui ne contient que Python2 et on y a installé la version de NLTK adaptée (NLTK==3.4.5). Aucune erreur ne nous ai affichée, le pipeline a été planté à l'étape de l'initialisation. Test à réessayer avec la version payante GCP.
* Lorsqu'on a écrit un code Python, le plugin Python Transformer n'arrive pas à débugger le script. Il nous indique seulement le nombre d'erreur à fixer, sans préciser le type d'erreur ou sa localisation, ceci ne fait que perdre du temps ! Voici à quoi ressemble le message d'erreur :



### Dataflow

Par opposition, nous remontons les points suivants pour Dataflow :

* Au contraire de Data Fusion, la création de pipeline avec Dataflow ne se fait pas en passant par une interface graphique. Elle nécessite la mise en œuvre d'une pipeline développée avec Apache Beam. A ce titre, il apparait qu'en première approche Dataflow est un outil moins accessible à court terme pour un public non-développer. Le cas échéant, il nécessite une montée en compétence sur le framework Apache Beam.
* Néanmoins, il est possible de recourir à des Template de pipeline permettant la mise en oeuvre d'ETL entre services GCP ou/et avec des services partenaires. Exemples:
  + [Apache Kafka to BigQuery](https://cloud.google.com/dataflow/docs/guides/templates/provided/kafka-to-bigquery)
  + [Change Data Capture from MySQL to BigQuery (Stream)](https://cloud.google.com/dataflow/docs/guides/templates/provided/mysql-change-data-capture-to-bigquery)
  + [Apache Cassandra to Bigtable](https://cloud.google.com/dataflow/docs/guides/templates/provided/cassandra-to-bigtable)
  + [Cloud Storage Avro to Bigtable](https://cloud.google.com/dataflow/docs/guides/templates/provided/avro-to-bigtable)
* En outre, cette approche (basée sur le développement) a l'avantage de permettre le développement de pipelines facilement personnalisables, en s'appuyant sur les bibliothèques disponibles dans les différents langages implémentant le framework Apache Beam (Python, Java, Go), et donc apporte beaucoup de flexibilité.
* Autre flexibilité notable : concerne les différentes approches pour exécuter un job sur Dataflow
  + en passant par le Cloud Shell (i.e en lignes de commandes)
  + en passant par le workbench (jupyter notebook)
  + en recourant à des templates (il est ainsi possible de créer son propre template de pipeline, et de l'exécuter ensuite rapidement avec l'exécuteur de templates)
  + en implémentant une pipeline avec Dataflow SQL : s'affranchissant ainsi des contraintes de développement/montée en compétence sur Apache Beam. Dataflow convertit ensuite la pipeline SQL en pipeline Apache Beam et l'exécute.

## Scalabilité

### Dataflow

* Le caractère entièrement scalable et serverless de Dataflow permet de s'affranchir entièrement des considérations de configuration de l'infrastructure sous-jacente pour le traitement distribué des données. Dataflow se charge de choisir la configuration la plus appropriée pour la réalisation des tâches.
* Il est néanmoins possible de "forcer" Dataflow à utiliser une certaine typologie de machines, si on l'estime préférable.
* N.B : dans le cadre d'un compte gratuit (free tier), Dataflow présente quelques limitations, notamment concernant le nombre max d'adresses ip utilisables par le cluster, limitant ainsi à 8 machines au maximum, ce qui limite donc les possibilités de scalabilité. Il est néanmoins possible de forcer Dataflow à recourir à des machines plus performantes, et ainsi réduire le besoin de machines supplémentaires, et donc en adresses IP.

### Data Fusion

Nous avons testé Data Fusion en mode autoscaling. A rappeler que ce mode d'exécution assure uniquement la scalabilité horizontale (nombre de workers). C'est à nous de gérer la scalabilité verticale (type de machine). Nous avons choisi les mêmes configuration des VM utilisées pour Dataflow. A noter qu'avec Dataflow, la scalabilité verticale ainsi que l'horizontale, sont par défaut entièrement gérées en mode autoscaling, et il est possible néanmoins de lui spécifier (forcer) par exemple la nature de la machine à utiliser pour l'exécution du travail.

*tableau comparatif*

|  |  |  |
| --- | --- | --- |
| type de machine | Dataflow | Data fusion |
| n1-standard-1 | Pipeline: Success\*TE: 14 min 55 sec | Pipeline: Failed(uniquement 300 lignes sont traitées) |
| n1-standard-2 | - | Pipeline: Failed (uniquement 430 lignes sont traitées) |
| n1-standard-4 | Pipeline: Success\*TE: 12 min | Pipeline: Success\*TE: 23 min 38 sec |

\*TE: Temps d'exécution

Nous remarquons que Data fusion, même en mode autoscaling, nécessite des machines assez performantes pour pouvoir exécuter le pipeline (n1-standard-4). En le comparant à Dataflow, elle consomme beaucoup plus de ressources pour qu'elle puisse être exécutée. Même da sa version gratuite Dataflow a réussi à exécuter l'ETL avec toutefois certaines limites :

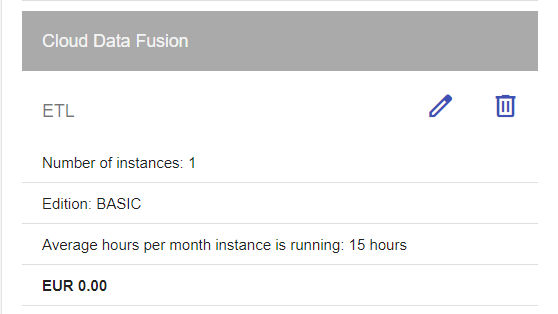
* le nombre d'adresses IP externes (pour les machines) est limité à 8, et donc, les ressources de machines mobilisables pour le travail est limité à 8 également, même si Dataflow estime avoir besoin de davantage de machines pour mener à bien son travail dans les meilleures conditions (forcer dataflow à utiliser des machines plus puissantes, ex: n1-standard-4 a permis de contourner ce problème)
* la pipeline ne s'est pas exécutée lorsque la table d'entrée était très grande (1To en input a fonctionné. En revanche, avec une table de plusieurs dizaines de To, la pipeline ne s'est pas exécutée). La scalabilité de dataflow semble dysfonctionner dans ces conditions, Dataflow n'approvisionnant alors qu'une seule machine pour le travail (qui risque de prendre plusieurs dizaines d'heures pour être exécuté).

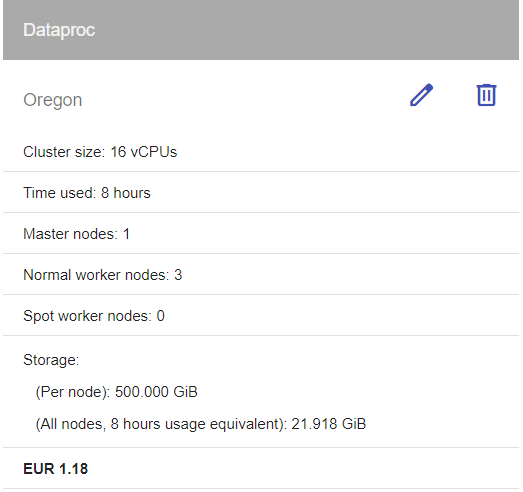
## Coût

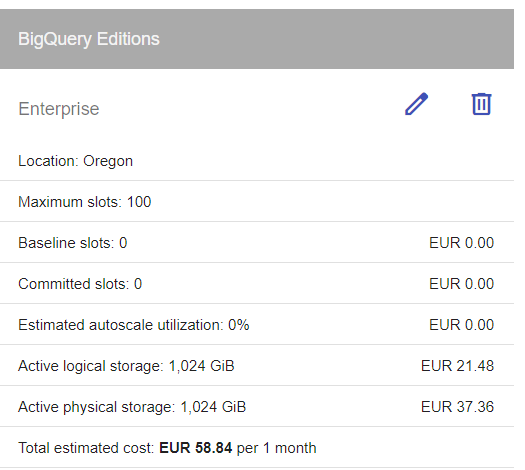
### Data Fusion

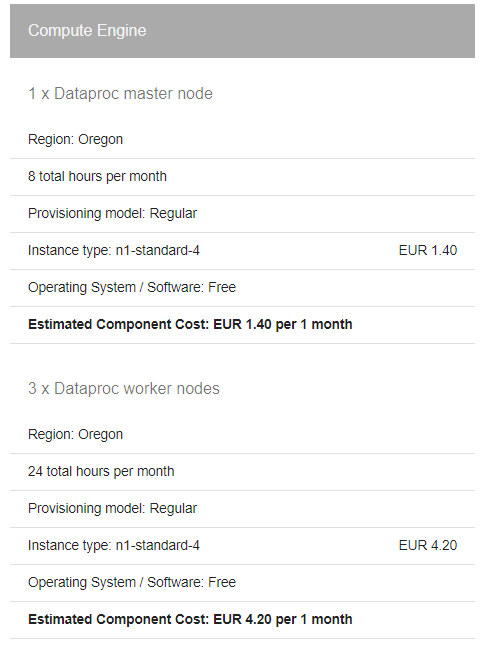
* Dans notre use case, nous avons utilisé le plugin Python Transformer. Pour pouvoir le tester, il a fallut déployer/exécuter le pipeline. Ceci consomme des ressources (temps, mémoire et services GCP) en instanciant l'image du cluster Dataproc et la lancer juste pour faire des tests !
* En plus, pour appliquer les actions d'initialisations (package supplémentaires à installer), il fallait créer un bucket dans Google Storage contenant le script d'initialisation. Ceci a un coût supplémentaire.
* Dans le cas où on veut utiliser un cluster personnalisé, il fallait créer une instance Dataproc. Le lancement et l'arrêt de ce cluster se fait manuellement. Donc si ça ne se fait pas rapidement, il y aura un coût à payer pour rien. A noter que lorsqu'on utilise un cluster prédéfinit dans Data Fusion, le lancement et l'arrêt du cluster se fait automatiquement.
* Les heures de développement sur Data Fusion, sont calculées à l'heure. Nous avons choisi comme type d'instance: "Développeur" facturé à 0.35$/h. Nous avons passé 7 jours à développer, ce qui nous a coûté (0.35\*24\*7 = 58.8$)

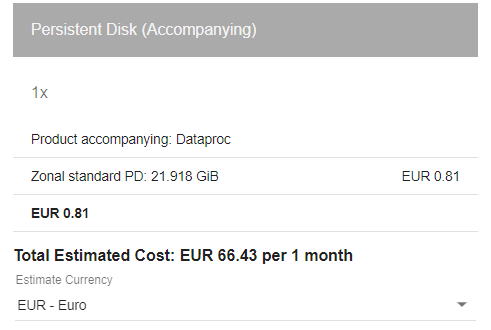
Voici ce que donne la simulation sur cet outil de Google : [Outil de simulation](https://cloud.google.com/products/calculator#id=ecfe328b-dc67-46a6-88e2-d6137779a5f5)









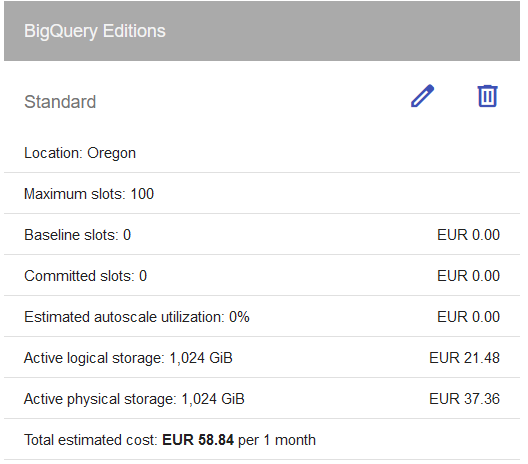


Il se rajoute à ceci les frais de développement, le total = 66.43 + 58.8 = 125.23 $ par mois.

### Dataflow

Pour la simulation des coûts avec Dataflow, nous avons pris les hypothèses suivantes :

* Nous manipulons une table de l'ordre du To dont des coûts de stockage sont à prévoir sur BigQuery
* Nous sommes localisés dans la région us-west-1 (et multiregions us pour Cloud Storage)
* nous utilisons pour Bigquery un faible (le plus faible) nombre max de slots : 100
* En revanche, nous considérons que nous ne recourons jamais à l'autoscaling (i.e que nous ne dépassons jamais 100 slots). Autrement, le prix est nettement plus cher (plusieurs ordres de grandeurs)
* Pour Dataflow, nous utilisons 4 machines de type n1-standard-4

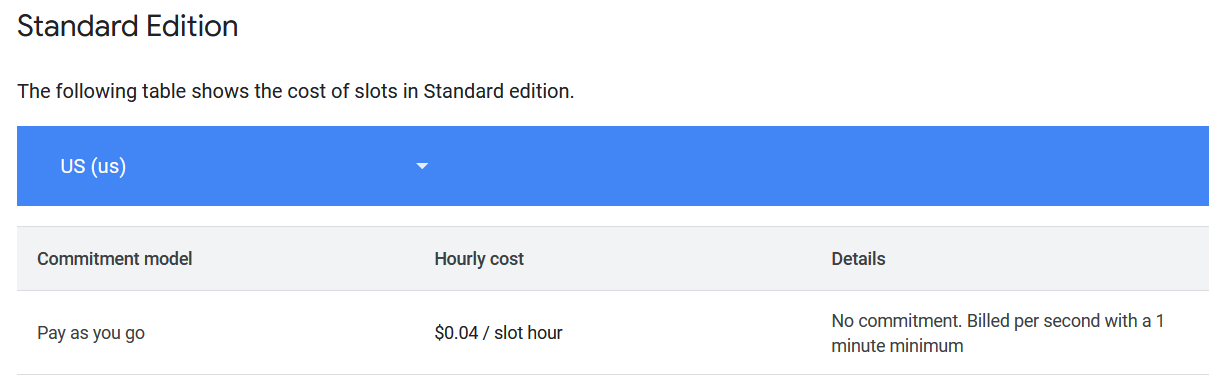


Une image contenant texte

Description générée automatiquement

Ainsi, le coût total estimé pour un mois est de l'ordre de 64€ environ.

**Toutefois**, il est à noter qu'il semblerait que certains coûts ne soient pas inclus, notamment en ce qui concerne les coûts des slots de l'édition Standard de BigQuery, comme indiqué ci-dessous (mais pas comptabilisé dans le tableau supra)



On pourrait ainsi considérer que les 100 slots soient utilisés à chaque fois, et qu'au total, nous avons utilisé BigQuery pendant 1h sur le mois ==> ce qui revient à ajouter : $0.04 \* 100 slots \* 1h = 4$

**Pour un total de l'ordre de 68$ donc mensuel.**

Il est également à noter que Dataflow comporte un avantage important du fait qu'il n'implique pas de coût de développement (lié à la création d'une instance et au temps de développement). Le développement peut en effet se faire simplement en local sur la machine du développeur, qui exécute simplement son code ensuite en ligne de commande dans le Cloud Shell sur GCP.

En synthèse :

# Bonus : consommation des données dans Looker

En attente de pouvoir utiliser Looker