

TP 3+4

“Shazam”

Ce double TP noté va nous occuper toute la journée. Vous le réalisez en binôme.

Il contient 3 parties :

- Une grosse partie 1 avec des manipulations sous Big Query
- Une partie 2 de discussion, sans manipulations
- Une partie 3 de manipulation de Cloud Composer, la version GCP d'Airflow

L'évaluation se fera sur les 3 parties, dont vous devrez stocker les réponses dans l'environnement du TP (voir plus loin). Les notes seront calibrées en fonction du point atteint par l'ensemble des binômes.

La partie 2 peut être commencée sans avoir fini la partie 1, mais il faut au moins avoir lu celle-ci et compris la logique qu'il y a derrière pour répondre à toutes les questions. Elle est indépendante de la partie 3.

La partie 3 peut être nécessiter d'avoir pratiqué le début de la partie 1.

Consignes générales

Le TP se déroule sur le cloud Google, comme les précédents, mais en utilisant des outils différents. Vous serez guidés pour la connexion et l'utilisation de ces outils. Pour vous faire gagner du temps sans devoir lire les documentations en ligne, ce TP est accompagné d'un autre document : un **kit de référence** des outils, qui reprend les éléments de syntaxe utiles pour le TP, et des réponses partielles sur lesquelles vous pourrez vous appuyer (ce n'est pas obligatoire, il y a toujours plusieurs manières de faire). N'hésitez pas à vous y référer ; les questions concernées y sont mises en évidence.

Veillez à bien lire les énoncés des questions, car il contiennent les informations de guidage pour les outils employés. Il décrivent aussi la manière dont les résultats sont attendus (exemple : noms des champs de table à produire), merci de respecter ces consignes. L'ordre des colonnes de table n'est pas important, mais leurs noms le sont.

Les requêtes et les réponses aux questions seront stockées dans un script que nous créerons au début. Attention à sauver régulièrement ce script, y compris avant de partir, pour ne pas perdre des réponses. Vous êtes libres de mettre ce que vous voulez dedans en plus des réponses, comme des notes de travail ; veillez simplement à bien mettre en évidence les n° de questions auxquelles vous répondez.

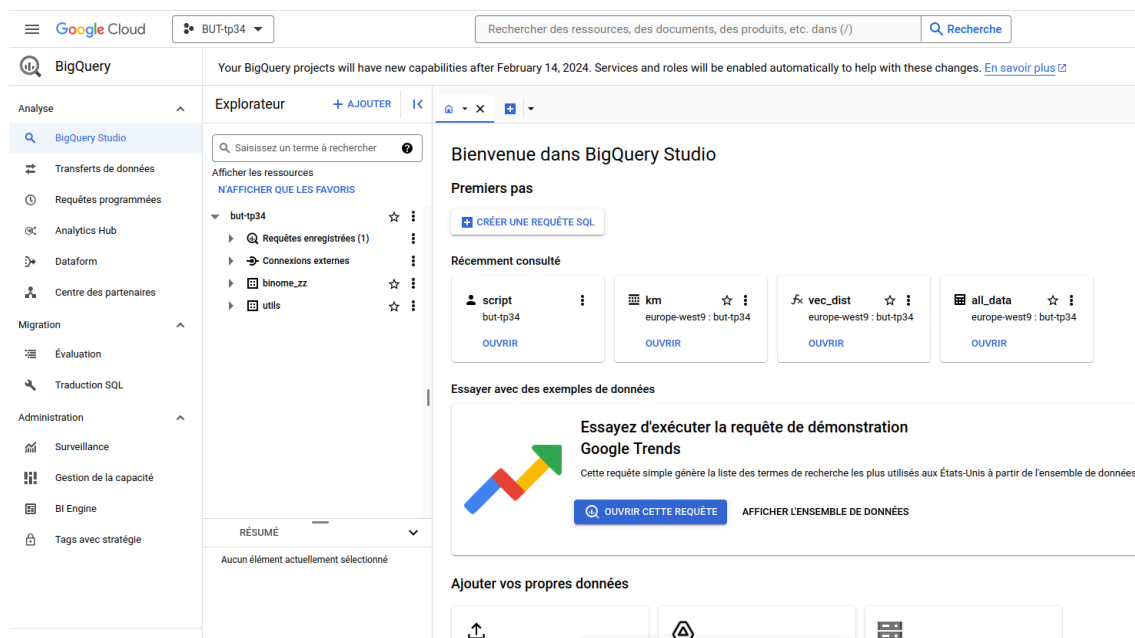
Connexion

Il faut ouvrir une session de navigateur (de préférence privée), et aller sur <https://console.cloud.google.com>.

Le login est de la forme binome_XNN@thomasvial.fr, XXN étant le n° de binôme (une lettre et 2 chiffres), et le mot de passe vous est fourni par ailleurs. Le mot de passe est composé de chiffres et de lettres en minuscules de a à f (clef hexadécimale aléatoire).

Après avoir accepté les conditions d'utilisation et quand la console s'affiche, sélectionner le projet BUT-tp34.

Naviguer ensuite vers Big Query en tapant par exemple "big query" dans la barre de recherche. Vous devez arriver sur un écran qui ressemble au suivant :



Commencez par créer un nouveau script en cliquant sur le + bleu dans l'en-tête de la partie droite, puis taper deux lignes dans l'éditeur : `/* et */`. Cliquez ensuite sur "ENREGISTRER", puis "Enregistrer la requête sous", et donnez un nom explicite, par exemple "Réponses". Laissez bien "personnelle" pour la visibilité.

Les `/*` et `*/` sont des marqueurs de commentaires, pour que Big Query n'essaye pas d'interpréter le script comme du code SQL. Vous taperez les réponses aux questions entre ces marqueurs, en n'oubliant pas de sauvegarder à chaque fois !

En premier lieu, écrivez les noms des membres de votre binôme, pour que je puisse attribuer les notes.

NB 1 : si vous fermez le script par erreur, vous le retrouverez dans l'explorateur sous "Requêtes enregistrées"

NB 2 : s'il est plus pratique pour vous de travailler indépendamment sur les parties 1 et 2, vous pouvez avoir 2 fichiers de réponses différents

Voyons maintenant le contexte du TP.

Contexte

Le cas d'utilisation est celui d'un service de reconnaissance musicale (type Shazam), basé sur l'utilisation d'**embeddings** (vecteurs) censés représenter les "empreintes digitales" des morceaux. La distance euclidienne entre deux vecteurs est interprétée comme la dissimilarité entre les sections musicales qu'ils résument.

Les données sont constituées de morceaux composés par des artistes ; chaque morceau a préalablement été transformé en un certain nombre de vecteurs, dont le nombre peut varier. La raison de cette variabilité est que chaque morceau a été découpé en fenêtres de quelques secondes, et qu'un vecteur est l'empreinte d'une seule fenêtre. La durée d'un morceau étant variable, il en est de même du nombre de fenêtres et donc de vecteurs.

NB : les données sont (malheureusement) fictives, il ne faut pas chercher de cohérence entre elles.

Le premier jeu de données est un fichier `metadata.csv`, qui énumère les 301 morceaux. Il contient une ligne par morceau avec les champs suivants séparés par des ";" :

- `song_id` : identifiant numérique du morceau
- `artist` : nom de l'artiste
- `title` : titre du morceau
- `duration` : durée du morceau, en secondes

Le deuxième jeu de données est un ensemble de fichiers "JSONL", c'est-à-dire des groupes de documents JSON stockés dans un même fichier à raison d'un document par ligne. Ces fichiers correspondent aux vecteurs issus des morceaux. Il y a au total 301 documents, répartis en 22 fichiers `vectors_01.jsonl` à `vectors_22.jsonl`.

Chaque document JSON (une ligne de fichier JSONL donc) est structuré ainsi :

- Un champ “id” qui contient l’ID du morceau (le même que dans le fichier `metadata.csv`)
- Un champ “vectors” qui est un tableau d’objets :
 - Chacun de ces objets a un seul champ “vector”, qui est un tableau de 16 nombres flottants
 - Tous les vecteurs sont donc de la même taille (16), mais il y a bien un nombre variable de tels vecteurs pour un morceau donné, selon la taille du tableau “vectors”

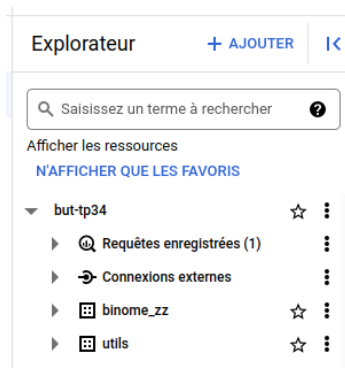
Concrètement, chaque document (i.e. chaque morceau) contient un tableau d’objets, et chaque objet contient un unique attribut qui est lui-même un tableau. Pour l’anecdote, Big Query ne permet pas le chargement de tableaux à deux dimensions, d’où ce passage par un niveau “objet” intermédiaire.

Toutes les données d’entrée sont dans le système de stockage objet de GCP, Google Cloud Storage.

Partie 1 - Manipulations dans Big Query

La première partie consiste à charger puis explorer les données dans Big Query pour les utiliser ensuite.

Dans le panneau de gauche, vous devez voir les éléments de votre “explorateur” personnel. Il doit y avoir l’identifiant de votre binôme ; c’est un dataset Big Query, qui permet d’organiser les objets créés (tables, etc.). Dans la capture d’écran, le dataset s’appelle `binome_zz`.



En cliquant sur les 3 points en face du dataset, vous pouvez créer une table. L’opération va créer la structure de la table, et importer des données dedans à l’occasion.

Question 1

A l'aide de l'interface graphique, créer une table metadata dans votre dataset, dont les données se trouvent dans Google Cloud Storage.

- Après avoir choisi le type de source, un champ apparaît avec un bouton "PARCOURIR". En cliquant dessus, il faut sélectionner le "bucket" (espace de stockage) `tv1-but-vecteurs`, et à l'intérieur, le fichier `metadata.csv`
- Le format doit être CSV
- Vous pouvez cocher "Détection automatique" dans la section "Schéma" pour que Big Query trouve tout seul les champs et leurs types
- Attention, le séparateur de champs est un point-virgule dans le fichier, mais Big Query suppose une virgule par défaut. Trouvez où changer ce paramètre
- Ne changez pas les autres paramètres par défaut
- Cliquez enfin sur "CREER LA TABLE" pour lancer le "job" de chargement

Si tout s'est bien passé, vous ne devez pas avoir d'erreurs. Sinon, vérifiez bien les informations.

En cliquant sur le nom de la table dans l'explorateur, vous pouvez voir la structure que Big Query a déduite, et en cliquant sur "PREVIEW", avoir un extrait des données. Vérifiez bien que le contenu semble correct, sinon il faut supprimer la table ("Supprimer" depuis les 3 petits points de l'explorateur) et recommencer le chargement.

Question 2

Créer de même une nouvelle table, `raw_vectors`, qui va cette fois contenir les vecteurs bruts tirés des documents JSON.

- Ces données sont au format JSONL
- Big Query est capable de détecter le schéma comme pour `metadata`
- Elles se trouvent dans le même bucket que `metadata.csv`, mais dans le répertoire `songs`. Vous pouvez sélectionner un des fichiers du bucket, et remplacer ensuite son nom par un nom générique (ex. `vectors_01.jsonl` → `*.jsonl`) pour que Big Query importe tout d'un coup

Comme précédemment, vérifiez le résultat via l'explorateur. La structure doit ressembler à ceci :

raw_vectors	REQUÊTE	PARTAGER	CC
SCHÉMA	DÉTAILS	PREVIEW	TRAÇABILITÉ
<div>Filtre</div> <div>Saisissez le nom ou la valeur de la propriété</div>			
<input type="checkbox"/>	Nom du champ	Type	Mode
<input type="checkbox"/>	vectors	RECORD	REPEATED
<input type="checkbox"/>	vector	FLOAT	REPEATED
<input type="checkbox"/>	id	INTEGER	NULLABLE

Et la prévisualisation n'est pas très lisible, à cause des tableaux imbriqués dans des objets :

raw_vectors

REQUÊTE

SCHÉMA

DÉTAILS

PREVIEW

Ligne	vector...vector	id
1	0.91438435... 0.92972281... 0.38236167... 0.07594775... 0.04126600... 0.41076529... 0.54526431... 0.84256877... 0.05405859... 0.33556630... 0.00000000...	15

Question 3

Actuellement, les données de vecteurs dans la table ont un modèle calqué sur celui des fichiers, qui n'est donc pas très pratique. On cherche à les “dénormaliser”, c'est-à-dire à transformer chaque ligne (qui correspond à un morceau) en autant de lignes qu'il y a de vecteurs pour son morceau, en répétant l'ID du morceau.

Schématiquement, cela revient à faire une opération de ce style pour chaque morceau (ici celui d'ID 42) :

id	vectors
42	[{"vector": [1, 2, 3]}, {"vector": [4, 5, 6]}, {"vector": [7, 8, 9]}]

⇒

id	window_id	vector
42	0	[1, 2, 3]
42	1	[4, 5, 6]
42	2	[7, 8, 9]

Les ID sont répétés autant de fois qu'il y a de vecteurs. Le nouveau champ window_id est le numéro de la fenêtre dans le morceau, i.e. la position du vecteur dans le tableau (le premier ayant la position 0).

A l'aide du *kit de référence*, mettre au point une requête SELECT qui fait cette transposition, puis créer une table `flat_vectors` avec le résultat. La table doit avoir la structure suivante :

- `id` : l'ID du morceau
- `window_id` : numéro de la fenêtre
- `vector` : un vecteur unique = un tableau à 16 flottants

Vous devez obtenir 7 569 lignes, et l'aperçu dans Big Query doit ressembler à ceci (à l'ordre des morceaux est arbitraire) :

flat_vectors			
REQUÊTE			
PART			
SCHÉMA	DÉTAILS	PREVIEW	TRAÇAGE
Ligne	id	window_id	vector
1	19	6	0.36763445...
			0.55259299...
			0.09380185...
			0.52798731...
			0.80197617...
			0.18087792...
			0.81525393...
			0.08239352...
			0.17932082...
			0.86554996...
			0.05315176...
			0.39050600...
			0.21496851...
			0.38612600...
			0.42411792...
			0.01343567...
2	21	6	0.29898353...
			0.19627811...
			0.27611592...
			0.63110329...

Question 4

Maintenant que nous avons des vecteurs à plat, nous allons enrichir la table avec les métadonnées des morceaux.

Créer une table `vectors_with_metadata`, avec comme structure :

- `song_id` (provient des deux tables)
- `window_id` (provient de `flat_vectors`)
- `artist` (provient de `metadata`)
- `title` (provient de `metadata`)
- `duration` (provient de `metadata`)
- `vector` (provient de `flat_vectors`)

Le nombre de lignes doit être identique à celui de la question précédente.

Question 5

Pour les besoins de notre clone de Shazam, il faut être en mesure de calculer la distance euclidienne entre deux vecteurs de taille identique. Nous allons pour cela créer une UDF : User-Defined Function.

Dans le *kit de référence*, vous avez un exemple d'UDF qui met en rapport les éléments de 2 vecteurs passés en paramètre, comme une sorte de jointure. Elle est aussi disponible dans vos environnements sous le nom ``but-tp34.utils.join_vectors``.

Pour cette question, il est demandé de créer une autre UDF, `euclidean2`, qui retourne un flottant et non une table virtuelle, en l'occurrence le carré de la distance euclidienne entre 2 vecteurs qui lui sont passés en paramètre. Vous pourrez utiliser la fonction `join_vectors`, comme si son résultat était une table SQL avec 2 colonnes, une pour chaque vecteur, et une ligne par paire d'éléments.

Pour rappel, si (x_i) et (y_i) sont deux vecteurs, le carré de cette distance est $\sum (x_i - y_i)^2$. On ne cherche pas à appliquer une racine carrée pour avoir la distance absolue, le carré suffit.

Pour tester la fonction :

```
SELECT `but-tp34.binome_XNN.euclidian2`([3., 1., 2.], [4., 5., 6.])
```

Le résultat doit être 33.0.

Question 6

Question 6.a

Ecrire une requête qui calcule, pour chaque morceau, le carré de la distance entre ses vecteurs et tous les vecteurs correspondant au morceau d'ID 31. Le résultat ne doit pas comparer le morceau n°31 à lui-même, et doit avoir les colonnes suivantes :

- `song_id_ref` (ID du morceau comparé au n°31)
- `window_id_ref` (ID de la fenêtre du vecteur du morceau comparé)
- `window_id_31` (ID de la fenêtre du vecteur du n°31)
- `artist_ref`, `title_ref`, `duration_ref` : informations du morceau comparé
- `distance2` (carré de la distance)

Dans ce qui précède, `_ref` désigne donc chaque morceau de la base auquel on compare le n°31 (attention à ne pas mélanger).

Il n'est pas demandé de créer une table, en revanche pensez à copier la requête dans le fichier de script.

Une stratégie possible :

- Préparer une requête qui extrait les vecteurs du morceau n°31
- Encapsuler cette requête dans une clause WITH pour avoir une “variable” de table (*voir le kit de référence*)
- Faire un produit cartésien entre cette “variable” et la table générale
- Calculer la distance au carré avec la fonction de la question précédente
- Sélectionner les colonnes qui vont bien, avec les bons noms

Pour contrôler, voici un les premières lignes du résultat (83 138 lignes au total) après tri par distance2, song_id_ref, window_id_ref puis window_id_31 :

Résultats de la requête									
ENREGISTRER LES RÉSULTATS EXPLORER LES DONNÉES									
INFORMATIONS SUR LE JOB		RÉSULTATS	GRAPHIQUE	BÊTA	JSON	DÉTAILS DE L'EXÉCUTION	GRAPHIQUE D'EXÉCUTION		
Ligne	song_id_ref	window_id_ref	window_id_31	artist	title	duration	distance2		
1	277	35	2	Chroma Chaos	Midnight Rebellion Ballad: Luna...	411	0.380703315136...		
2	9	9	0	Ultraviolet Uprising	Voltage Virago's Symphony: De...	140	0.425768673185...		
3	21	10	1	Ultraviolet Uprising	Time-Warped Streets: Retrogra...	116	0.429955815775...		
4	57	14	7	Cybernetic Cyclone	Midnight Rebellion: Lunar Lulla...	225	0.435694150383...		
5	26	14	0	Nebula Nexus	Uprising in Silence: Undergroun...	230	0.448328559010...		
6	153	7	5	Starlight Surge	Stella fetched a rubber duck, davdreaming about a future in	319	0.452243716568...		

Question 6.b

Une fois ceci mis au point, trouver les 5 morceaux les plus proches du n°31, par ordre croissant de distance. Là encore, pas besoin de créer la table mais mettre la requête dans le script.

Pour ce faire, on peut :

- Encapsuler la partie principale de la requête de la question 6.a dans une autre clause WITH (*voir le kit pour enchaîner les WITH*)
- ... et l'utiliser dans une agrégation en gardant, pour chaque couple de fenêtres window_id_ref et window_id_31, le minimum des carrés de leurs distances 2 à 2

Cela revient à considérer que la distance entre 2 morceaux est le minimum des distances entre toutes les paires vecteurs qui les composent. Ainsi, un morceau est identifié dès lors qu'on a trouvé une fenêtre qui correspond à celle d'un morceau déjà en base.

Le résultat doit ressembler à ceci :

Résultats de la requête

ENREGISTR

INFORMATIONS SUR LE JOB

RÉSULTATS

GRAPHIQUE

BÊTA

JSON

DÉTAILS DE L'EXÉCUTION

GRAPHIQUE

Ligne	song_id_ref	artist_ref	title_ref	duration_ref	distance2	
1	277	Chroma Chaos	Midnight Rebellion Ballad: Luna...	411	0.380703315136...	
2	9	Ultraviolet Uprising	Voltage Virago's Symphony: De...	140	0.425768673185...	
3	21	Ultraviolet Uprising	Time-Warped Streets: Retrogra...	116	0.429955815775...	
4	57	Cybernetic Cyclone	Midnight Rebellion: Lunar Lulla...	225	0.435694150383...	
5	26	Nebula Nexus	Uprising in Silence: Undergroun...	230	0.448328559010...	

Question 7

Nous continuons l'exploration des données avec la création d'un modèle de machine learning, au sein de Big Query. C'est un modèle de clustering, opérant sur les vecteurs.

On aurait aimé réutiliser notre fonction de distance, mais Big Query ML n'offre malheureusement pas la possibilité de personnaliser celle-ci (il supporte les distances euclidienne et cosinus). On va donc utiliser la distance euclidienne fournie. Cependant, le k-means suppose que les vecteurs sont sous forme tabulaire, et pas de tableaux imbriqués comme on en a manipulé jusqu'à présent. Il faut transformer les données avant d'entraîner le modèle (on peut voir ça comme une étape de feature engineering).

Nous devons donc "pivoter" les vecteurs pour en faire des colonnes. Mais il faut d'abord les "déplier" en ligne !

Dans le kit de référence, on fournit la requête qui fait le pivot, vous pouvez l'exécuter pour voir son résultat.

On vérifie bien que le nombre de lignes n'a pas changé : 7 569.

Question 7.a

Maintenant que l'on sait pivoter les données, entraîner un modèle de type k-means, en utilisant le squelette proposé par le kit de référence, sur tout le jeu de données pivoté. La transformation doit spécifier que seules les colonnes V0 à V15 sont utilisées.

Les options du modèle doivent être :

- MODEL_TYPE = 'KMEANS'
- NUM_CLUSTERS = 5
- KMEANS_INIT_METHOD = 'KMEANS++'
- DISTANCE_TYPE = 'euclidean' (attention c'est bien 'euclidean')
- STANDARDIZE_FEATURES = FALSE

Vous pouvez aller voir le détail du modèle et accéder à des statistiques sur l'entraînement, les centroïdes des clusters, ...

Question 7.b

Appliquer ensuite la fonction d'inférence sur ce même jeu de données, pour voir comment les clusters ont été attribués.

Vérifier que le résultat de l'inférence ressemble à ceci :

Résultats de la requête									
INFORMATIONS SUR LE JOB		RÉSULTATS	GRAPHIQUE	BETA	JSON	DÉTAILS DE L'EXÉCUTION	GRAPHIQUE D'EXÉCUTION		
Ligne	CENTROID_ID	CENTROID_ID	N. DISTANCE	song_id	artist	title	duration	window_id	V0
1	4	4	1.033398010354...	176	Cosmic Chaos	Polished Chrome Hearts on Mi...	430	3	0.381345160754...
		5	1.122267624808...						
		3	1.146748839084...						
		2	1.205650836624...						
		1	1.280139774467...						
2	4	4	0.973920492740...	13	Analog Assault	Concrete Canyons' Ballads: Lu...	394	0	0.555915834312...
		3	1.010193278140...						
		1	1.030808264655...						

On voit qu'il y a des tableaux imbriqués : la liste des centroïdes est répétée pour chaque ligne du jeu de test (ici identique à l'entraînement), avec en face la distance du vecteur à chaque centroïde.

Partie 2 - Discussion

Cette partie peut être faite sans avoir complété la première, elle ne contient pas de manipulations pratiques.

Question 8

Parmi les données impliquées dans notre scénario Shazam, quelles sont les données structurées ? Semi-structurées ? Non structurées ?

Question 9

Pour aboutir à un vrai produit du type de Shazam, il faut un moyen de soumettre des morceaux inconnus, et de les comparer à la base `vectors_with_metadata` (via un calcul de distance comme à la question 6 par exemple).

Quelle forme (modèle de données) pourraient prendre les données correspondant à de l'audio capté dans l'environnement pour être confronté à la base ?

Question 10

Supposons que notre service fonctionne sur la base d'un abonnement, et que seuls les utilisateurs inscrits puissent le faire fonctionner. L'ID utilisateur serait alors ajouté au modèle de données de la question 9, et transiterait avec l'empreinte audio à identifier. Ensuite, le résultat de l'identification avec l'ID utilisateur serait partagé avec Spotify, afin que celui-ci lui propose d'ajouter le morceau à sa playlist (on suppose que les profils Shazam et Spotify ont été appariés par l'utilisateur, avec son consentement).

Quelles précautions notre Shazam doit-il prendre dans ce scénario, pour être conforme à la réglementation ?

Question 11 (bonus)

Si vous êtes versé(e) en architecture, proposez un chaîne technique pour capter l'audio, calculer son empreinte, la soumettre au service qui déclenchera la comparaison avec les morceaux en base. Une partie impliquera un terminal type smartphone, et une autre des services sur le cloud, en utilisant de préférence les briques du PaaS (le nom exact des composants PaaS n'est pas important, c'est leur typologie et la discussion qui sont utiles ici).

Partie 3 - Cloud Composer

Dans cette partie, vous allez manipuler Airflow sous sa version "managée", Cloud Composer. Nous allons automatiser le chargement de données dans Big Query.

Pour vos manipulations, vous avez un bucket Cloud Storage nommé `tvi-but34-binome_XNN`, avec les droits de lire et d'écrire dedans.

Question 12

Trouvez l'interface de Cloud Storage dans la console Google Cloud, et entrez dans votre bucket. Cliquez sur le bouton "CREER UN DOSSIER", et créez un dossier `new_songs`. Ce répertoire recevra des nouveaux morceaux à indexer.

Pour les besoins du TP nous allons copier dans `new_songs` quelques fichiers du bucket partagé, `tvi-but34-vecteurs`. A l'aide de l'interface Cloud Storage, allez dans le bucket

partagé, téléchargez sur votre poste quelques fichiers JSONL du répertoire songs, et uploadez-les ensuite dans votre bucket dédié (bouton “IMPORTER DES FICHIERS”).

Question 13

En vous appuyant sur le script `dag_airflow.py` dans le repo Github du TP, écrire un programme Python qui crée un DAG Airflow en deux étapes enchaînées :

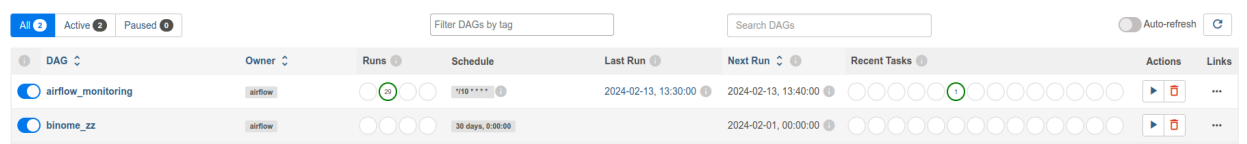
- Afficher un message de bienvenue
- Charger dans Big Query, dans votre table `new_raw_vectors`, les fichiers présents dans le bucket partagé

Nommez le programme Python en fonction de votre binôme, et déployez-le en uploadant le script Python dans le répertoire dags du bucket `tvi-but34-binome_zz`, avec la console Google Cloud.

Ce répertoire est partagé avec tout le monde, merci de ne pas interférer avec les autres :-)

Après quelques minutes, s’il n’y a pas d’erreur, votre DAG devrait apparaître dans la console Airflow :

<https://e748f666d944495aa80492ebf06efae-dot-europe-west9.composer.googleusercontent.com/home>



The screenshot shows the Airflow web interface. At the top, there are tabs for 'All', 'Active', and 'Paused'. Below this is a table of DAGs. The first DAG is 'airflow_monitoring', owned by 'airflow', with a status of 'Running' (indicated by a green circle with a white 'R'). The second DAG is 'binome_zz', also owned by 'airflow', with a status of 'Waiting for the scheduler' (indicated by a grey circle). The table includes columns for DAG, Owner, Runs, Schedule, Last Run, Next Run, Recent Tasks, Actions, and Links.

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks	Actions	Links
airflow_monitoring	airflow	1	* * * * *	2024-02-13, 13:30:00	2024-02-13, 13:40:00	...	▶ 🛑
binome_zz	airflow	0	30 days, 9:00:00		2024-02-01, 00:00:00	...	▶ 🛑

S’il n’apparaît pas au bout de 2-3 minutes, il y a peut-être une erreur dans le script qui empêche le déploiement du DAG. Dans ce cas, vous pouvez aller voir dans la vue DAG d’Airflow, si un bandeau rouge est affiché en tête de page. Il faut alors ajuster le script, l’uploader de nouveau et attendre...

Une fois le DAG au point, il devrait donc s’afficher dans la console Airflow. En cliquant sur son nom, vous pouvez accéder à différentes informations sur le DAG et notamment son graphe.

Le bouton en forme de triangle, à droite, lance manuellement le DAG, et les étapes changent de couleur au fur et à mesure, selon leur succès. Si tout va bien elles passent au vert foncé l’une après l’autre. S’il y a une erreur, il y aura des tentatives de rejeu (cadre jaune vert clair), et il faudra supprimer le DAG, corriger le script et le redéposer.

Quand l'exécution est complète, vérifiez que vous avez bien des données dans la table `new_raw_vectors`. Quant au message de bienvenue, il est visible dans les logs ; on peut les voir dans la vue graphe d'Airflow, en cliquant sur une étape puis le bouton "Log".