

BigQuery

Sommaire

- Présentation de Google Cloud Platform (GCP)
- Introduction à BigQuery :
 - Qu'est-ce que BigQuery ?
 - Fonctionnement et architecture (serveurless, MPP, stockage/traitement)
- Interface utilisateur BigQuery (UI, CLI, API)
- Créer son premier projet GCP et dataset BigQuery

Sommaire

- Rappels SQL appliqués à BigQuery
 - SELECT, FROM, WHERE
 - ORDER BY, LIMIT
 - Fonctions intégrées (string, date, math, etc.)
- Jointures et sous-requêtes
- GROUP BY, HAVING, fonctions d'agrégation
- Fonctions analytiques (fenêtres, RANK, LAG, etc.)
- Table expressions

Sommaire

- Chargement de données (CSV, JSON, AVRO, Parquet)
- Création de tables (manuelles ou à partir de requêtes)
- Tables temporaires vs permanentes
- Optimisation des requêtes : partitionnement, clustering
- Introduction aux vues

Sommaire

- Gestion des accès (IAM, rôles BigQuery)
- Scripts SQL (boucles, conditions)
- Programmation via Cloud Functions / Scheduler
- Introduction aux UDF (User Defined Functions)
- Visualisation des résultats avec Data Studio ou Looker Studio

Présentation de Google Cloud Platform (GCP)

Google Cloud Platform (GCP) est une suite de services de cloud computing proposée par Google qui s'appuie sur la même infrastructure interne que celle utilisée par Google pour ses produits destinés aux utilisateurs finaux, comme Google Search, Gmail, Google Drive et YouTube. GCP offre une large gamme de services dans plusieurs domaines, notamment le calcul, le stockage, la mise en réseau, le Big Data, l'apprentissage automatique (Machine Learning), l'Internet des Objets (IoT), la sécurité et le cloud management.

Présentation de Google Cloud Platform (GCP)

1. **Compute Engine** : Un service IaaS (Infrastructure as a Service) qui permet aux utilisateurs de lancer des machines virtuelles (VM) sur la infrastructure de Google.
2. **App Engine** : Une plateforme PaaS (Platform as a Service) pour le développement et l'hébergement d'applications web dans les centres de données gérés par Google.
3. **Google Kubernetes Engine (GKE)** : Un service de gestion de conteneurs pour l'exécution et l'orchestration de systèmes d'applications conteneurisées à l'aide de Kubernetes.

Présentation de Google Cloud Platform (GCP)

- 4. **Cloud Storage** : Un service de stockage d'objets puissant et simple pour stocker et accéder à des données depuis n'importe où sur le web.
- 5. **BigQuery** : Un entrepôt de données d'entreprise pour l'analyse de Big Data qui est entièrement géré et sans serveur, permettant des analyses rapides SQL sur de grands ensembles de données.

Présentation de Google Cloud Platform (GCP)

Compute



Compute Engine



App Engine



Container Engine



Container Registry



Cloud Functions

Identity & Security



Cloud IAM



Cloud Resource Manager



Cloud Security Scanner



Cloud Platform Security

Networking



Cloud Virtual Network



Cloud Load Balancing



Cloud CDN



Cloud Interconnect



Cloud DNS

Big Data



BigQuery



Cloud Dataflow



Cloud Dataproc



Cloud Datalab



Cloud Pub/Sub



Genomics

Storage and Databases



Cloud Storage



Cloud Bigtable



Cloud Datastore



Cloud SQL



Persistent Disk

Machine Learning



Cloud Machine Learning



Vision API



Speech API



Natural Language API



Translation API



Jobs API

Introduction à BigQuery

Qu'est-ce que BigQuery ?

BigQuery est le **service d'analyse de données à grande échelle** de Google Cloud.

Il fait partie de la catégorie des **entrepôts de données (data warehouses)**, et permet d'exécuter des **requêtes SQL** sur d'importants volumes de données **rapidement et sans gérer d'infrastructure**.

Introduction à BigQuery

- Stocker de grandes quantités de données (plusieurs téraoctets ou pétaoctets),
- L'interroger via SQL de manière fluide et rapide,
- Créer des tableaux de bord, des modèles de machine learning ou des pipelines analytiques automatisés.

Introduction à BigQuery

Caractéristique	Description
Entrepôt de données	BigQuery est conçu pour centraliser et analyser de grandes quantités de données.
Langage SQL standard	Pas besoin de savoir coder : les requêtes sont en SQL.
Très rapide	Même sur des milliards de lignes, les résultats peuvent être obtenus en quelques secondes.
Aucune gestion serveur	On ne déploie ni VM, ni cluster : Google gère toute l'infrastructure.
Paieement à l'usage	On paye à la quantité de données analysée et stockée.

Introduction à BigQuery

Fonctionnement et architecture de BigQuery

1. Serverless

- **Pas de machines à configurer**, pas de serveurs à maintenir.
- L'utilisateur se **concentre uniquement sur la donnée et la requête**.
- L'allocation de ressources se fait automatiquement en fonction du besoin.

Avantages : simplicité, économie, scalabilité automatique.

Introduction à BigQuery

2. Architecture MPP (Massively Parallel Processing)

- Une requête est automatiquement **découpée en sous-tâches**.
 - Ces sous-tâches sont **exécutées en parallèle sur des milliers de nœuds** internes à l'infrastructure Google.
 - Les résultats sont agrégés et retournés rapidement à l'utilisateur.
- Cela permet à BigQuery d'analyser **des milliards de lignes en quelques secondes**, sans ralentissement.*

Introduction à BigQuery

3. Séparation stockage / traitement

BigQuery sépare **le stockage** des données et **leur traitement** :

Stockage:

- Les données sont **stockées de façon colonne**, ce qui est très efficace pour les lectures analytiques.
- Les fichiers sont répartis sur **le système de stockage distribué de Google** (Colossus).
- Résilient, chiffré, compressé, avec une disponibilité très élevée.

Introduction à BigQuery

Traitement :

- Les requêtes SQL sont envoyées à une **infrastructure de calcul temporaire et dynamique**.
- Aucun composant n'est dédié à un utilisateur : tout est élastique et partagé.
- On paye **au volume de données analysées**, pas à la durée ou au nombre de CPU.

Interface utilisateur BigQuery (UI, CLI, API)

BigQuery propose **trois façons principales** d'interagir avec ses services :

- une **interface graphique (UI Web)**
- une **interface en ligne de commande (CLI)**
- une **API** (avec bibliothèques clients comme Python, Java...)

Interface Web (UI)

👉 <https://console.cloud.google.com/bigquery>

Avantages :

- Aucun besoin d'installation
- Très pédagogique
- Parfait pour l'exploration manuelle, les requêtes SQL simples, et les démos

Interface Web (UI)

Principales fonctionnalités

- Écriture et exécution de requêtes SQL dans un éditeur graphique
- Navigation dans les projets, datasets, tables
- Création de datasets et tables
- Chargement de fichiers (CSV, JSON, Avro...)
- Export de résultats vers Google Sheets, CSV, Cloud Storage
- Programmation de jobs récurrents (via l'onglet "scheduled queries")
- Visualisation du plan d'exécution des requêtes

Interface CLI (Command Line Interface)

Le CLI utilise les outils `gcloud` et `bq` disponibles dans le [Google Cloud SDK](#).

- Interagir avec BigQuery sans interface graphique
- Automatiser les requêtes dans des scripts
- Gérer projets, datasets, tables, autorisations

Interface CLI (Command Line Interface)

Initialisation

Après installation du SDK :

```
gcloud init
```

👉 Cela te permet :

- de te connecter à ton compte Google
- de sélectionner ou créer un projet GCP
- de configurer le CLI pour ce projet

Interface CLI (Command Line Interface)

Vérifier l'authentification :

```
gcloud auth list
```

Lister tes projets :

```
gcloud projects list
```

Interface CLI (Command Line Interface)

Commandes essentielles bq

Créer un dataset :

```
bq mk --dataset mon_projet:mon_dataset
```

Lister les tables d'un dataset :

```
bq ls mon_projet:mon_dataset
```

Charger un fichier CSV dans une table :

```
bq load --autodetect --source_format=CSV \  
mon_dataset.ma_table ./mon_fichier.csv
```

Exécuter une requête SQL :

```
bq query --use_legacy_sql=false '  
SELECT name, COUNT(*)  
FROM `bigquery-public-data.usa_names.usa_1910_2013`  
GROUP BY name  
ORDER BY COUNT(*) DESC  
LIMIT 10;  
'
```

Interface API et bibliothèques client

- Utiliser BigQuery depuis du code (Python, Java, Node.js...)
- Construire des pipelines automatisés
- Intégrer BigQuery dans des applications ou dashboards

Interface API et bibliothèques client

Authentification (2 méthodes)

1. Utilisateur local (développement) :

```
gcloud auth application-default login
```

→ Crée un fichier de credentials dans :

```
~/.config/gcloud/application_default_credentials.json
```

Dans le code Python :

```
from google.cloud import bigquery  
client = bigquery.Client() # utilise les ADC
```

Interface API et bibliothèques client

Authentification (2 méthodes)

2. Service account (production, cloud, serveurs)

1. Créer un **compte de service** depuis GCP
2. Télécharger la **clé JSON**
3. Exporter la variable d'environnement :

```
export GOOGLE_APPLICATION_CREDENTIALS="/chemin/vers/cle.json"
```

Puis dans le code :

```
client = bigquery.Client() # détecte la clé via la variable
```

Résumé

Interface	Idéale pour...	Configuration	Niveau de contrôle
UI Web	Débutants, analystes	Aucune	Basique, manuel
CLI (bq)	Développeurs, DevOps	<code>gcloud init</code>	Élevé
API (Python, etc.)	Intégration logicielle, automatisation	ADC ou service account	Très élevé

Quiz Time



Créer son premier projet GCP et dataset BigQuery

Avant de pouvoir utiliser BigQuery (ou tout autre service Google Cloud), on doit avoir un **projet actif**. Ce projet sert de **contenant administratif et technique** pour toutes tes ressources: datasets, tables, fonctions, budgets, autorisations, etc.

Créer un **dataset** dans BigQuery revient à **structurer ton espace de travail analytique**. C'est la première étape avant de stocker ou interroger des données.

Projet GCP

Un **projet** GCP est :

- Une **unité d'organisation** dans Google Cloud
- Un **espace isolé** avec ses propres :
 - Permissions (IAM)
 - Budget/facturation
 - Services activés
 - Ressources (datasets, VM, buckets...)

Quand on lance une requête BigQuery, elle est **toujours liée à un projet**.

dataset BigQuery

Un **dataset** BigQuery est :

- Un **conteneur logique de tables et vues**
- Nécessaire avant de pouvoir :
 - Charger des données
 - Écrire des requêtes avec des résultats enregistrés
 - Organiser tes ressources analytiques (comme un dossier)

Il appartient à **un seul projet** et est défini par :

- Un **nom (ID)**
- Un **emplacement géographique** (EU ou US)

Créer son premier projet GCP et dataset BigQuery

- Projet GCP : `projet-ventes-2025`
- Dataset BigQuery : `ventes_france`
- Tables dans le dataset :
 - `ventes_2024`
 - `ventes_2025`
 - `clients`

Le projet est le **cadre global**, le dataset est le **répertoire organisé**, et les tables sont les **fichiers contenant les données**.

Créer son premier projet GCP et dataset BigQuery

Étape	Ce qu'on fait	Pourquoi
Créer un projet GCP	On définit un cadre d'exécution	Obligatoire pour toute ressource cloud
Activer BigQuery	On autorise le projet à utiliser BigQuery	Nécessaire pour le service
Créer un dataset	On prépare l'espace logique pour stocker les tables	Obligatoire pour charger ou requêter

Rappels SQL appliqués à BigQuery

Source utilisée : `bigquery-public-data.stackoverflow.posts_questions`

1. Instructions SQL fondamentales

◆ **SELECT**, **FROM**, **WHERE**

```
SELECT id, title, tags
FROM `bigquery-public-data.stackoverflow.posts_questions`
WHERE tags LIKE '%<python>%'
LIMIT 5;
```

- **SELECT** : sélectionne les colonnes à afficher
- **FROM** : indique la table source
- **WHERE** : filtre les lignes selon une condition logique

Rappels SQL appliqués à BigQuery

- ORDER BY, LIMIT

```
SELECT EXTRACT(YEAR FROM creation_date) AS year, COUNT(*) AS nb_questions
FROM `bigquery-public-data.stackoverflow.posts_questions`
GROUP BY year
HAVING nb_questions > 1000
ORDER BY year;
```

- ORDER BY trie les résultats
- DESC ou ASC : ordre décroissant ou croissant
- LIMIT limite le nombre de lignes

Rappels SQL appliqués à BigQuery

- GROUP BY, HAVING

```
SELECT EXTRACT(YEAR FROM creation_date) AS year, COUNT(*) AS nb_questions
FROM `bigquery-public-data.stackoverflow.posts_questions`
GROUP BY year
HAVING nb_questions > 1000
ORDER BY year;
```

- GROUP BY regroupe les lignes selon une colonne
- HAVING filtre les groupes (contrairement à WHERE qui filtre les lignes)

Rappels SQL appliqués à BigQuery

2. Fonctions intégrées BigQuery

- A. Fonctions de chaînes (STRING)

```
SELECT title,  
       UPPER(title) AS maj,  
       LENGTH(title) AS longueur,  
       REGEXP_CONTAINS(tags, r'<python>') AS contient_python  
FROM `bigquery-public-data.stackoverflow.posts_questions`  
LIMIT 5;
```

- UPPER() / LOWER() : majuscule/minuscule
- LENGTH() : longueur
- REGEXP_CONTAINS() : expression régulière
- REPLACE(), TRIM(), CONCAT(), SPLIT(), LEFT(), RIGHT()

Rappels SQL appliqués à BigQuery

- B. Fonctions numériques et arithmétiques

```
SELECT score,  
       Sqrt(score) AS racine,  
       Power(score, 2) AS carre,  
       Mod(score, 3) AS reste  
FROM `bigquery-public-data.stackoverflow.posts_questions`  
WHERE score > 0  
LIMIT 5;
```

- ABS(), ROUND(), CEIL(), FLOOR(), MOD(), POWER(), SQRT(), RAND()

Rappels SQL appliqués à BigQuery

- C. Fonctions de date et heure

```
SELECT creation_date,  
       EXTRACT(YEAR FROM creation_date) AS annee,  
       DATE_TRUNC(creation_date, MONTH) AS mois  
FROM `bigquery-public-data.stackoverflow.posts_questions`  
LIMIT 5;
```

- CURRENT_DATE(), CURRENT_TIMESTAMP()
- EXTRACT(), DATE_TRUNC(), DATE_DIFF(), DATE_ADD(), FORMAT_DATE()

Rappels SQL appliqués à BigQuery

- D. Fonctions logiques et conditionnelles

```
SELECT title, score,  
       IF(score > 10, 'Très populaire', 'Moyen') AS evaluation  
FROM `bigquery-public-data.stackoverflow.posts_questions`  
LIMIT 5;
```

- IF(condition, A, B)
- CASE WHEN ... THEN ... ELSE ... END
- IS NULL, IS NOT NULL, COALESCE(), IFNULL()

Rappels SQL appliqués à BigQuery

- E. Fonctions d'agrégation

```
SELECT EXTRACT(YEAR FROM creation_date) AS annee,  
       COUNT(*) AS nb_questions,  
       AVG(score) AS score_moyen  
FROM `bigquery-public-data.stackoverflow.posts_questions`  
GROUP BY annee  
ORDER BY annee DESC;
```

- COUNT(), SUM(), AVG(), MIN(), MAX(), STRING_AGG(), ARRAY_AGG()

Rappels SQL appliqués à BigQuery

- F. Fonctions analytiques (WINDOW FUNCTIONS)

```
SELECT id, title, score,  
       RANK() OVER (ORDER BY score DESC) AS classement  
FROM `bigquery-public-data.stackoverflow.posts_questions`  
WHERE tags LIKE '%<javascript>%'  
LIMIT 5;
```

- RANK(), DENSE_RANK(), ROW_NUMBER()
- LAG(), LEAD(), FIRST_VALUE(), LAST_VALUE()
- OVER(PARTITION BY ...)

Rappels SQL appliqués à BigQuery

- G. Fonctions sur les tableaux (ARRAY)

```
SELECT id,  
       SPLIT(tags, '|') AS balises  
FROM `bigquery-public-data.stackoverflow.posts_questions`  
LIMIT 5;
```

- SPLIT(), ARRAY_LENGTH(), ARRAY_TO_STRING(), UNNEST(),
 GENERATE_ARRAY()

Rappels SQL appliqués à BigQuery

- H. Fonctions sur les données JSON (JSON)

Si la table contient un champ JSON (comme `body` ou d'autres datasets) :

```
SELECT id,  
       JSON_EXTRACT_SCALAR(body, '$.someField') AS champ_json  
FROM `bigquery-public-data.stackoverflow.posts_questions`  
WHERE JSON_EXTRACT_SCALAR(body, '$.someField') IS NOT NULL  
LIMIT 5;
```

- `JSON_EXTRACT()`, `JSON_EXTRACT_SCALAR()`, `TO_JSON_STRING()`,
`PARSE_JSON()`

Rappels SQL appliqués à BigQuery

- I. Fonctions statistiques

```
SELECT APPROX_QUANTILES(score, 4) AS quartiles  
FROM `bigquery-public-data.stackoverflow.posts_questions`;
```

- APPROX_QUANTILES(), CORR(), STDDEV(), VAR_POP(), COVAR_POP()

Rappels SQL appliqués à BigQuery

- J. Fonctions de cryptographie et sécurité

```
SELECT id,  
       MD5(CAST(id AS STRING)) AS hash_id  
FROM `bigquery-public-data.stackoverflow.posts_questions`  
LIMIT 5;
```

- MD5(), SHA1(), SHA256(), FARM_FINGERPRINT()

Lab 1 : Analyse des commits GitHub avec BigQuery

- Appliquer les instructions SQL : `SELECT`, `WHERE`, `ORDER BY`, `LIMIT`
- Manipuler les dates, les chaînes, et les valeurs agrégées
- Découvrir des fonctions analytiques sur des données de commits GitHub
- Vous êtes analyste data dans une équipe DevOps. On vous confie l'étude des messages de commits GitHub pour mieux comprendre les pratiques des contributeurs (auteurs, fréquence, style, mots-clés, etc.).

Vous allez travailler sur la table suivante :

`bigquery-public-data.github_repos.sample_commits`

Lab 1 : Analyse des commits GitHub avec BigQuery

- **1. Exploration des commits**

- Affichez le sujet du commit (`subject`), le nom de l'auteur (`author.name`), la date du commit (`committer.date`) et le message complet (`message`) pour les 10 derniers commits disponibles.

- **2. Nombre de commits par auteur**

- Identifiez les 10 auteurs ayant réalisé le plus de commits dans l'ensemble de la table.

Lab 1 : Analyse des commits GitHub avec BigQuery

- 3. **Évolution temporelle**

- Affichez, pour les commits disponibles, le nombre total de commits par **mois et année**.
- Triez les résultats par date décroissante.

- 4. **Mots-clés fréquents**

- Listez les commits dont le message contient les mots `fix` ou `bug`, peu importe la casse.

Lab 1 : Analyse des commits GitHub avec BigQuery

- **5. Analyse du style des messages**
 - Calculez la longueur moyenne du champ `message` pour chaque auteur.
 - Classez les auteurs selon la longueur moyenne décroissante.
- **6. Commits avec messages très courts**
 - Affichez les commits dont le champ `message` contient moins de 15 caractères.
 - ■ Triez-les par longueur croissante.

Jointures et Sous-requêtes en SQL avec BigQuery

1. Les jointures (JOIN)

Les jointures permettent de **combiner des données provenant de plusieurs tables** en fonction d'une relation logique (souvent via une clé étrangère ou une colonne partagée).

```
SELECT *  
FROM table1  
JOIN table2  
  ON table1.colonne = table2.colonne
```

Jointures et Sous-requêtes en SQL avec BigQuery

Types de jointures

Type de jointure	Description	Comportement
INNER JOIN	La plus fréquente	Retourne uniquement les lignes qui ont une correspondance dans les deux tables
LEFT JOIN	Jointure externe à gauche	Toutes les lignes de la table de gauche, même sans correspondance
RIGHT JOIN	Externe à droite	L'inverse de LEFT JOIN
FULL JOIN	Jointure complète	Toutes les lignes des deux tables, même sans correspondance
CROSS JOIN	Produit cartésien	Toutes les combinaisons possibles (⚠ très lourd)

Jointures et Sous-requêtes en SQL avec BigQuery

Exemple

- `posts_questions` (id, title, owner_user_id)
- `users` (id, display_name, reputation)

```
SELECT
  q.id AS question_id,
  q.title,
  u.display_name,
  u.reputation
FROM `bigquery-public-data.stackoverflow.posts_questions` AS q
JOIN `bigquery-public-data.stackoverflow.users` AS u
  ON q.owner_user_id = u.id
LIMIT 10;
```

Jointures et Sous-requêtes en SQL avec BigQuery

- `q` et `u` sont des alias
- On relie chaque question à l'utilisateur qui l'a postée
- Si un `owner_user_id` est `NULL` → non pris avec `INNER JOIN`

Jointures et Sous-requêtes en SQL avec BigQuery

2. Les sous-requêtes (Subqueries)

Les sous-requêtes permettent de **composer une requête dans une autre** : pour filtrer, agréger, classer, etc.

Jointures et Sous-requêtes en SQL avec BigQuery

Types de sous-requêtes

Type	Description	Exemple
Dans le SELECT	Une requête pour calculer une valeur	(SELECT COUNT(*) FROM ...)
Dans le FROM	Crée une table temporaire	FROM (SELECT ...) AS sous_table
Dans le WHERE	Permet de filtrer	WHERE x IN (SELECT y FROM ...)

Jointures et Sous-requêtes en SQL avec BigQuery

Exemple

Lister les utilisateurs ayant posté une question avec le tag `<python>`.

```
SELECT id, display_name
FROM `bigquery-public-data.stackoverflow.users`
WHERE id IN (
  SELECT owner_user_id
  FROM `bigquery-public-data.stackoverflow.posts_questions`
  WHERE tags LIKE '%<python>%'
)
```

Jointures et Sous-requêtes en SQL avec BigQuery

Afficher le top 10 des utilisateurs ayant posté le plus de questions.

```
SELECT user_id, nb_questions
FROM (
  SELECT owner_user_id AS user_id, COUNT(*) AS nb_questions
  FROM `bigquery-public-data.stackoverflow.posts_questions`
  GROUP BY owner_user_id
)
ORDER BY nb_questions DESC
LIMIT 10;
```

Jointures et Sous-requêtes en SQL avec BigQuery

Afficher pour chaque utilisateur la moyenne de ses scores.

```
SELECT
  id,
  display_name,
  (SELECT AVG(score)
   FROM `bigquery-public-data.stackoverflow.posts_questions` AS q
   WHERE q.owner_user_id = u.id) AS avg_score
FROM `bigquery-public-data.stackoverflow.users` AS u
LIMIT 10;
```

Jointures et Sous-requêtes en SQL avec BigQuery

Bonnes pratiques

- ⚡ **Optimisation** : BigQuery est optimisé pour les sous-requêtes imbriquées, mais attention à la **duplication de calculs**.
- 💧 **Alias obligatoires** : Toute sous-requête dans `FROM` doit être nommée (avec `AS alias`).
- 📏 **LIMIT** dans une sous-requête dans `SELECT` peut ralentir fortement si mal utilisé.

Lab 2

Datasets :

- `bigquery-public-data.austin_bikeshare.bikeshare_trips`
- `bigquery-public-data.austin_bikeshare.bikeshare_stations`

1. Explorer les trajets

Affichez les 10 premiers trajets, en montrant les colonnes suivantes :

- `trip_id`, `start_time`, `start_station_id`, `end_station_id`,
`duration_minutes`

Lab 2

2. Nom de la station de départ

- Affichez les trajets enrichis avec le **nom de la station de départ**.

3. Classement des stations par nombre de trajets

- Déterminez quelles sont les 10 stations de départ les plus utilisées, avec leur **nombre total de trajets**.

4. Sous-requête dans **WHERE**

- Affichez les **trajets** qui sont partis depuis les **5 stations les plus utilisées** (en nombre de départs).

Lab 2

5. Sous-requête dans FROM

- Créez une table temporaire contenant la **durée moyenne des trajets par station de départ**, puis affichez les stations avec la plus grande durée moyenne.

Lab 2

6. Sous-requête dans `SELECT`

- Pour chaque station, affichez :
 - Son nom
 - Le **nombre total de trajets** qui ont commencé depuis cette station (sous-requête dans `SELECT`)

Lab 2

7. Optionnel : Joindre la station d'arrivée

- Réalisez une requête affichant, pour chaque trajet :
 - Le nom de la station de départ
 - Le nom de la station d'arrivée

Fonctions analytiques (fenêtres) dans BigQuery

Une **fonction analytique**, aussi appelée **fonction de fenêtre** (*window function*), permet de faire **des calculs ligne par ligne tout en conservant toutes les lignes**, contrairement aux fonctions d'agrégation (`SUM`, `AVG`, etc.) qui regroupent les lignes. Elles s'utilisent avec la clause `OVER(...)`.

Fonctions analytiques (fenêtres) dans BigQuery

```
SELECT  
  name,  
  score,  
  RANK() OVER (ORDER BY score DESC) AS classement  
FROM `exemple.table_etudiants`;
```

💡 Cela classe chaque étudiant selon son score, mais **affiche toutes les lignes** avec leur rang.

Fonctions analytiques (fenêtres) dans BigQuery

Structure générale

```
FONCTION() OVER (  
  PARTITION BY ...    -- regroupe les données comme un GROUP BY  
  ORDER BY ...        -- définit l'ordre de calcul  
)
```

Fonctions analytiques (fenêtres) dans BigQuery

Les fonctions les plus utilisées

✓ RANK() / DENSE_RANK() / ROW_NUMBER()

Fonction	Effet
RANK()	Attribue un rang avec des sauts en cas d'égalité
DENSE_RANK()	Rang sans sauts
ROW_NUMBER()	Numéro unique, sans doublons, selon l'ordre

Fonctions analytiques (fenêtres) dans BigQuery

Exemple :

```
SELECT
  name,
  score,
  RANK() OVER (ORDER BY score DESC) AS rang
FROM `exemple.eleves`;
```

Fonctions analytiques (fenêtres) dans BigQuery

✓ LAG() / LEAD()

Permet d'accéder à la valeur **précédente** ou **suivante** dans une fenêtre.

```
SELECT
  name,
  score,
  LAG(score) OVER (ORDER BY date_exam ASC) AS score_prec,
  LEAD(score) OVER (ORDER BY date_exam ASC) AS score_suiv
FROM `exemple.eleves`;
```

Fonctions analytiques (fenêtres) dans BigQuery

✓ `FIRST_VALUE()` / `LAST_VALUE()`

Récupère la première ou la dernière valeur d'une fenêtre.

```
SELECT
  name,
  score,
  FIRST_VALUE(score) OVER (PARTITION BY classe ORDER BY date_exam) AS premier_score
FROM `exemple.eleves`;
```

✓ `SUM()`, `AVG()` sur fenêtre glissante

```
SELECT
  date_exam,
  score,
  SUM(score) OVER (ORDER BY date_exam ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS somme_3_jours
FROM `exemple.eleves`;
```


Fonctions analytiques (fenêtres) dans BigQuery

✓ NTILE(n)

Découpe les lignes en **n** groupes de taille égale.

```
SELECT
  name,
  score,
  NTILE(4) OVER (ORDER BY score DESC) AS quartile
FROM `exemple.eleves`;
```

Fonctions analytiques (fenêtres) dans BigQuery

- Classement dynamique
- Comparaison avec valeurs précédentes
- Calculs glissants
- Segmentation en déciles/quartiles
- Toujours indiquer un `ORDER BY` dans `OVER()` pour les fonctions `RANK`, `LAG`, `FIRST_VALUE`
- Utiliser `PARTITION BY` pour "recommencer" les calculs par groupe
- Préférer `WITH` pour éviter de répéter des calculs coûteux

Table expressions (**WITH** clauses ou CTE)

C'est une **table temporaire définie dans une requête**, qui peut ensuite être utilisée comme source.

Elle améliore la **lisibilité**, la **réutilisabilité** et les **performances** en évitant les calculs répétés.

Table expressions (**WITH** clauses ou CTE)

```
WITH nom_table_temp AS (  
    SELECT ...  
    FROM ...  
    WHERE ...  
)  
SELECT ...  
FROM nom_table_temp
```

Table expressions (**WITH** clauses ou CTE)

```
WITH trajets_2024 AS (  
  SELECT *  
  FROM `bigquery-public-data.austin_bikeshare.bikeshare_trips`  
  WHERE EXTRACT(YEAR FROM start_time) = 2024  
)  
SELECT start_station_id, COUNT(*) AS nb_trajets  
FROM trajets_2024  
GROUP BY start_station_id;
```

Table expressions (**WITH** clauses ou CTE)

Avantages des **WITH**

- On évite de répéter un **SELECT** complexe plusieurs fois
- On **structure la logique** en étapes lisibles
- BigQuery peut **optimiser l'exécution** de chaque **WITH**

Table expressions (**WITH** clauses ou CTE)

```
WITH trajets_2024 AS (  
  SELECT *  
  FROM `bigquery-public-data.austin_bikeshare.bikeshare_trips`  
  WHERE EXTRACT(YEAR FROM start_time) = 2024  
)  
SELECT  
  bike_id,  
  start_time,  
  ROW_NUMBER() OVER (PARTITION BY bikeid ORDER BY start_time) AS ordre_des_trajets  
FROM trajets_2024;
```

Objectif ici : numéroté les trajets dans l'ordre pour chaque vélo (**bike_id**)

Lab 3

Dataset utilisé :

```
bigquery-public-data.austin_bikeshare.bikeshare_trips
```

1. RANK des trajets les plus longs

Pour chaque vélo (`bike_id`), affichez ses 3 trajets les plus longs, avec leur rang (1 = plus long) en utilisant `RANK()`.

- Affichez : `bike_id`, `trip_id`, `duration_minutes`, `rang_duree`

Lab 3

2. Évolution de la durée de trajet

Affichez, pour chaque vélo, la durée de son trajet et celle du trajet précédent. Utilisez `LAG()`.

Affichez : `bike_id`, `trip_id`, `start_time`, `duration_minutes`,
`duration_precedente`

Lab 3

3. Moyenne glissante sur 3 trajets

Pour chaque vélo, calculez la moyenne mobile de la durée sur **3 trajets successifs**.

Affichez :

- `bike_id`
- `start_time`
- `duration_minutes`
- `moyenne_mobile_3`

Lab 3

4. Utilisation de **WITH** pour isoler les trajets récents

Créez une CTE **trajets_recents** contenant uniquement les trajets depuis janvier 2017.

Puis, à partir de cette CTE :

- Affichez les trajets groupés par **start_station_id**
- Comptez le nombre total de trajets par station
- Triez par ordre décroissant

Lab 3

5. `FIRST_VALUE()` de la première station utilisée par vélo

Affichez, pour chaque trajet, la première station de départ utilisée par le vélo (`bike_id`) dans l'historique, avec `FIRST_VALUE()`.

Affichez : `bike_id`, `trip_id`, `start_time`, `start_station_id`,
`premiere_station_utilisee`

Lab 3

6. (Challenge bonus) Total cumulé de minutes de trajet par vélo

Pour chaque vélo :

- Calculez un total **cumulé** des minutes de trajet, triés par `start_time`.
- Utilisez `SUM(...) OVER (...)`.
- Affichez : `bike_id`, `start_time`, `duration_minutes`,
`total_minutes_cumulees`

Chargement de données (CSV, JSON, AVRO, Parquet)

BigQuery permet d'importer des données depuis des fichiers externes (stockés dans Cloud Storage ou en local via l'interface Web ou CLI) dans des **tables BigQuery**.

Les formats les plus courants pris en charge sont :

- **CSV** : texte délimité par des virgules ou autres séparateurs
- **JSON (NDJSON)** : chaque ligne est un objet JSON
- **AVRO** : format binaire optimisé, avec schéma intégré
- **Parquet** : format binaire colonnes, très efficace pour l'analyse
- **ORC** : moins courant mais aussi supporté

Chargement de données (CSV, JSON, AVRO, Parquet)

- Depuis un fichier local (via l'**interface Web**)
- Depuis **Google Cloud Storage (GCS)** (recommandé pour les gros volumes)
- Via **Streaming** (ligne à ligne, pas adapté pour CSV/JSON complets)
- Via **Federated Query** (lecture directe depuis un fichier sans l'importer)

Création de tables

Dans BigQuery, une **table** est une structure de stockage dans un **dataset**, qui peut être :

- Créée manuellement (vide ou avec un schéma défini),
- Générée à partir du résultat d'une requête,
- Optimisée par des mécanismes comme le partitionnement et le clustering,
- Sécurisée par du chiffrement (automatique ou personnalisé).

Création de tables

1. Création manuelle d'une table

- Interface Web (Google Cloud Console)
- Ligne de commande avec `bq`
- API REST
- SQL DDL (`CREATE TABLE`)

Création de tables

- Exemple avec SQL :

```
CREATE TABLE mon_projet.mon_dataset.ma_table (  
  id INT64,  
  nom STRING,  
  actif BOOL  
);
```

- Exemple avec bq CLI :

```
bq mk --table mon_dataset.ma_table \  
id:INT64,nom:STRING,actif:BOOL
```

Création de tables

2. Création à partir d'une requête SQL

BigQuery permet de créer une table directement à partir d'un `SELECT`.

```
CREATE TABLE mon_dataset.table_resultat AS  
SELECT * FROM mon_dataset.source  
WHERE annee = 2024;
```

Création de tables

3. Partitionnement

Le **partitionnement** permet de diviser une table en **sous-ensembles logiques**, pour :

- Accélérer les requêtes,
- Réduire les coûts de scan,
- Optimiser la gestion des données.

Création de tables

Types de partitionnement :

Type	Syntaxe SQL	Description
DATE ou TIMESTAMP	PARTITION BY DATE(colonne)	Sur une colonne date ou timestamp
INT RANGE	PARTITION BY RANGE_BUCKET(colonne, [0, 10, 20, 30])	Par plages d'entiers
Ingestion-time	PARTITION BY _PARTITIONDATE	Auto-généré à l'import

```
CREATE TABLE dataset.commandes
PARTITION BY DATE(date_commande)
AS SELECT * FROM autre_table;
```

Création de tables

4. Clustering

Le **clustering** trie les données à l'intérieur de chaque partition sur une ou plusieurs colonnes.

Cela permet :

- Une meilleure **performance** des requêtes,
- Une meilleure **compression**.

```
CREATE TABLE dataset.commandes  
PARTITION BY DATE(date_commande)  
CLUSTER BY client_id, region  
AS SELECT * FROM source;
```

Création de tables

5. Chiffrement (Encryption)

Mode	Détail
Par défaut (GMEK)	Géré automatiquement par Google
Clé personnalisée (CMEK)	Utilisation de ta propre clé KMS

Vues

Une **vue** est une **requête SQL enregistrée**, qui se comporte **comme une table virtuelle**. Elle **n'a pas ses propres données**, mais **affiche dynamiquement les résultats** d'une requête chaque fois qu'on l'utilise.

```
CREATE VIEW mon_dataset.vue_commandes_2024 AS  
SELECT * FROM mon_dataset.commandes  
WHERE EXTRACT(YEAR FROM date_commande) = 2024;
```

“ Cette vue agit comme une table contenant uniquement les commandes de 2024, mais les données restent dans `commandes`. ”

Vues

- Les **types de vues**

Type de vue	Description	Stocke les données ?	Performances
✓ Vue standard	Requête dynamique, résultat calculé à chaque appel	✗ Non	⚠ Peut être lente sur gros volumes
✓ Vue matérialisée	Résultat pré-calculé et mis en cache automatiquement	✓ Oui (partiellement)	🚀 Très rapide

Vues

- **Vue standard (classique)**
 - C'est une définition SQL dynamique.
 - Pas de données stockées : tout est recalculé à chaque requête.
 - Toujours à jour.

Vues

- **Vue matérialisée**

- C'est une **vue optimisée**, qui conserve un **résultat partiel** en cache.
- BigQuery met à jour les données automatiquement **si la source change**.
- Requêtes **beaucoup plus rapides**, mais **plus contraignantes** (requêtes simples, pas de jointures complexes, pas de `UNION`).

Vues

Aspect	Vue standard	Vue matérialisée
Toujours à jour	✓ Oui	✓ (avec latence de mise à jour)
Données stockées	✗ Non	✓ Oui (résultat mis en cache)
Peut contenir des jointures ?	✓ Oui	✗ Non (requêtes simples uniquement)
Performances	⚠ Dépendent des tables sous-jacentes	🚀 Optimisées
Utilisation de quotas	Oui	Oui (moindre pour les vues matérialisées si cache utilisé)
Exportation possible ?	Non directement	Oui

Vues

- **SQL (standard)**

```
CREATE VIEW mon_dataset.vue_clients_actifs AS
SELECT id, nom
FROM mon_dataset.clients
WHERE actif = TRUE;
```

- **SQL (vue matérialisée)**

```
CREATE MATERIALIZED VIEW mon_dataset.mv_total_ventes_par_jour AS
SELECT
    DATE(vente_date) AS jour,
    SUM(montant) AS total
FROM mon_dataset.ventes
GROUP BY jour;
```

Vues

- **Interface Web (UI)**

1. Ouvrir BigQuery Console
 2. Écrire ta requête dans l'éditeur SQL
 3. Cliquer sur `Enregistrer > Enregistrer comme vue`
 4. Donner un nom, un dataset cible
 5. Enregistrer
- “ Pour les vues matérialisées, utiliser l'onglet "Créer une vue matérialisée" dans l'UI. ”

Vues

- CLI avec bq
- Vue standard :

```
bq mk --view \  
'SELECT id, nom FROM mon_dataset.clients WHERE actif=TRUE' \  
mon_dataset.vue_clients
```

- Vue matérialisée :

```
bq mk --materialized_view --query \  
'SELECT DATE(vente_date) AS jour, SUM(montant) AS total FROM mon_dataset.ventes GROUP BY jour' \  
mon_dataset.mv_ventes_journalieres
```

Vues

- On peut **restreindre l'accès à certaines colonnes**.
- On peut accorder aux utilisateurs **l'accès à la vue, mais pas à la table source**.

```
GRANT SELECT ON TABLE mon_dataset.vue_clients TO 'user@domaine.com';
```

“ L'utilisateur ne peut pas voir les données sensibles dans `clients`, mais peut voir ce que retourne la vue. ”

Vues

• Vues vs Tables : résumé

Élément	Vue standard	Vue matérialisée	Table permanente
Stocke les données ?	✗	✓ partiellement	✓
Mise à jour	Automatique	Automatique (avec cache)	Manuelle
Requêtes rapides	✗	✓	✓
Coût d'accès	À chaque exécution	Moindre si cache	Direct
Données à jour	Toujours	Avec un léger délai	Selon rafraîchissement

Vues

- Cas d'usage des vues

Cas	Type de vue
Cacher les colonnes sensibles	Vue standard
Exposer un sous-ensemble de données à une équipe	Vue standard
Optimiser les dashboards (Looker, Data Studio)	Vue matérialisée
Sauvegarder un résultat figé	Table permanente

Lab 4

L'université fictive **CampusNova** centralise les données d'admissions de ses étudiants. Vous êtes chargé de structurer et d'optimiser ces données dans BigQuery. Ce lab se déroule dans un dataset nommé `admissions_lab`, à créer dans la région `europa-west1`.

- Créez un **dataset** nommé `admissions_lab` dans la région `europa-west1`.

Lab 4

Vous devez importer les fichiers suivants dans BigQuery :

1. Un fichier CSV `etudiants.csv` contenant les colonnes :

- id, nom, age, ville

2. Un fichier JSON `inscriptions.json` contenant :

- etudiant_id, formation, date_inscription

3. Un fichier AVRO `examens.avro` avec les champs :

- exam_id, matiere, date_exam, coefficient

Lab 4

4. Un fichier Parquet `resultats.parquet` avec :

- `etudiant_id`, `exam_id`, `note`

Lab 4

Question 1 : Chargez chaque fichier dans une table distincte dans le dataset `admissions_lab`, en utilisant le bon format source.

Question 2 : Créez manuellement une table nommée `candidatures` dans `admissions_lab`, avec les colonnes suivantes :

- `id` (INT64), `nom` (STRING), `formation` (STRING), `score` (FLOAT64), `date_candidature` (DATE)

Question 3 : Créez une nouvelle table `candidats_2023` contenant uniquement les lignes de la table `candidatures` dont l'année de `date_candidature` est 2023.

Lab 4

Question 4 : Effectuez une requête de comptage du nombre total de candidatures **sans définir de table de destination**. Quelle est la nature de la table utilisée par le job BigQuery ?

Question 5 : Reproduisez la même requête, mais cette fois en spécifiant une table de destination appelée `compte_candidats`.

Question 6 : Créez une table `candidatures_optimizees` partitionnée par la colonne `date_candidature` et clusterisée par la colonne `formation`, à partir de la table `candidatures`.

Lab 4

Question 7 : Effectuez deux requêtes :

- Une qui filtre sur une date précise dans `candidatures_optimizees`
- Une autre sur la table `candidatures` non partitionnée

Question 8 : Créez une **vue standard** appelée `vue_filtrée` qui retourne uniquement les candidatures dont le score est supérieur à 15, sans afficher la date.

Question 9 : Créez une **vue matérialisée** appelée `mv_score_moyen` qui retourne le score moyen par formation à partir de la table `candidatures`.

Gestion des accès (IAM, rôles BigQuery)

IAM (Identity and Access Management) est le système de gestion des identités de Google Cloud. Il permet d'**attribuer des rôles** à des identités (utilisateurs, groupes, comptes de service) pour accéder à des **ressources précises**, avec un niveau de permission contrôlé.

Dans BigQuery, les ressources principales sont : Projet, Dataset, Table, Vue, Job

Gestion des accès (IAM, rôles BigQuery)

Type de rôle	Description
Rôles basiques	roles/viewer, roles/editor, roles/owner : trop larges, déconseillés pour une bonne sécurité
Rôles prédéfinis BigQuery	Plus fins, adaptés à BigQuery : ex. roles/bigquery.dataViewer, roles/bigquery.jobUser, etc.
Rôles personnalisés	Créés par l'admin, permettent un contrôle très fin, mais plus complexes à maintenir

Gestion des accès (IAM, rôles BigQuery)

- Exemples de rôles BigQuery

Rôle	But
roles/bigquery.viewer	Peut lire les métadonnées des datasets
roles/bigquery.dataViewer	Peut lire les données des tables
roles/bigquery.dataEditor	Peut lire et modifier les données
roles/bigquery.dataOwner	Peut modifier les données + donner des accès
roles/bigquery.user	Peut créer des jobs BigQuery
roles/bigquery.jobUser	Peut lancer des requêtes mais pas forcément lire les données

Gestion des accès (IAM, rôles BigQuery)

- Toujours appliquer le **principe du moindre privilège** : donner juste assez de droits pour que l'utilisateur puisse faire son travail.
- Privilégier les **rôles prédéfinis** adaptés à BigQuery, évitant les rôles génériques comme `Editor`.
- Ne jamais utiliser de **comptes personnels** dans un projet professionnel : préférer les **groupes** ou **comptes de service**.
- Documenter les règles d'accès et les intégrer dans une **politique de gouvernance des données**.

Introduction aux scripts SQL dans BigQuery

BigQuery supporte le **SQL scripté** depuis 2020, dans le style **PL/pgSQL** (PostgreSQL-like), avec des blocs `DECLARE`, `BEGIN`, `END`, `IF`, `LOOP`, etc.

“ Un script SQL est un bloc de commandes exécutées **de manière procédurale** : on enchaîne les instructions, on teste des conditions, on boucle, on utilise des variables. ”

Structure d'un script SQL

```
DECLARE var_name STRING;  
DECLARE total INT64;  
  
BEGIN  
    SET var_name = "BigQuery";  
    SET total = 10;  
  
    IF total > 5 THEN  
        SELECT "More than five" AS message;  
    ELSE  
        SELECT "Five or less" AS message;  
    END IF;  
END;
```

Déclaration de variables

```
DECLARE count_rows INT64;  
DECLARE user_name STRING;  
DECLARE is_active BOOL;
```

On initialise avec SET :

```
SET count_rows = 0;  
SET user_name = "admin";  
SET is_active = TRUE;
```

Conditions

- **IF...THEN...ELSE**

```
IF count_rows > 100 THEN
    SELECT "Too many rows";
ELSE
    SELECT "OK";
END IF;
```

- **CASE**

```
SELECT
CASE
    WHEN score >= 90 THEN "Excellent"
    WHEN score >= 70 THEN "Good"
    ELSE "Needs improvement"
END AS evaluation
FROM my_table;
```


Boucles

- **WHILE loop**

```
DECLARE i INT64 DEFAULT 0;

WHILE i < 5 DO
    SELECT CONCAT("Iteration: ", CAST(i AS STRING));
    SET i = i + 1;
END WHILE;
```

- **FOR loop**

```
FOR i IN UNNEST(GENERATE_ARRAY(1, 5)) DO
    SELECT CONCAT("Value: ", CAST(i AS STRING));
END FOR;
```

Procédures stockées

- Réutiliser du code complexe
- Automatiser des traitements : nettoyage, validation, génération de rapports
- Paramétrer l'exécution

```
CREATE OR REPLACE PROCEDURE dataset.hello_procedure(name STRING)
BEGIN
    SELECT CONCAT("Hello, ", name, "!") AS greeting;
END;
```

Appel de la procédure :



```
CALL dataset.hello_procedure("Alice");
```

Procédure avec logique + paramètres

```
CREATE OR REPLACE PROCEDURE dataset.clean_logs(days_threshold INT64)
BEGIN
  DECLARE deleted_count INT64;
  -- Suppression des lignes trop anciennes
  DELETE FROM dataset.logs
  WHERE event_date < DATE_SUB(CURRENT_DATE(), INTERVAL days_threshold DAY);
  -- Vérification
  SET deleted_count = (
    SELECT COUNT(*) FROM dataset.logs
    WHERE event_date < DATE_SUB(CURRENT_DATE(), INTERVAL days_threshold DAY)
  );

  IF deleted_count = 0 THEN
    SELECT "Cleanup successful!" AS status;
  ELSE
    SELECT "Some entries remain" AS status;
  END IF;
END;
```

Lab 5


- **Question 1:** Déclarez une variable `total_anomalies` et stockez-y le nombre de lignes dont le champ `amount` est NULL ou égal à 0.
- **Question 2:** Ajoutez une **condition** `IF` pour afficher un message selon le nombre d'anomalies :
 - Si `total_anomalies > 2` → afficher “ Trop d'anomalies détectées !”
 - Sinon → “ Données conformes.”

Lab 5

- **Question 3:** Supprimez les lignes contenant `amount IS NULL` ou `= 0` **uniquement si** des anomalies sont présentes.
 - Réutilisez la variable `total_anomalies`
 - Utilisez une instruction `DELETE FROM`
 - Affichez un message après suppression

Lab 5

- **Question 4:** Pour chaque magasin distinct (`store`), affichez la phrase :

“  "Traitement du magasin : [nom du magasin]" ”
- **Question 5:** Combinez toutes les étapes précédentes en un **script unique** qui :
 - Compte les anomalies
 - Les supprime si besoin
 - Affiche un message selon le résultat
 - Affiche les magasins traités

Programmation via Cloud Functions / Scheduler

“ **Cloud Scheduler** est le “**cron**” du cloud.

Il permet de **programmer l'exécution régulière** d'une tâche (quotidienne, horaire, etc.) ”

- Il peut **déclencher une URL HTTP**, un **Pub/Sub**, ou une tâche **App Engine**
- Il est souvent utilisé pour **planifier l'exécution** de :
 - Exports de données
 - Nettoyages ou backups
 - Fonctions Cloud
 - Requêtes BigQuery

Démo : automatiser une requête BigQuery tous les jours

- Étape 1 : Créer une requête SQL à lancer

```
-- daily_sales.sql  
CREATE OR REPLACE TABLE dataset.daily_sales AS  
SELECT CURRENT_DATE() AS date, SUM(amount) AS total  
FROM dataset.transactions  
WHERE DATE(event_time) = CURRENT_DATE();
```


Démo : automatiser une requête BigQuery tous les jours

- Étape 2 : Créer une Cloud Function (ex. en Python)

```
from google.cloud import bigquery
def run_query(request):
    client = bigquery.Client()
    query = """
    CREATE OR REPLACE TABLE dataset.daily_sales AS
    SELECT CURRENT_DATE() AS date, SUM(amount) AS total
    FROM dataset.transactions
    WHERE DATE(event_time) = CURRENT_DATE()
    """

    client.query(query).result()
    return "Query executed", 200
```

Démo : automatiser une requête BigQuery tous les jours

- requirements.txt

```
google-cloud-bigquery
```

- **Étape 3 : Déployer la fonction**

```
gcloud functions deploy run_query \  
  --runtime python310 \  
  --trigger-http \  
  --allow-unauthenticated \  
  --region=europe-west1
```

👉 Vous obtenez une **URL** de fonction (ex.
`https://...cloudfunctions.net/run_query`)

Démo : automatiser une requête BigQuery tous les jours

- **Étape 4 : Planifier avec Cloud Scheduler**

```
gcloud scheduler jobs create http daily-job \  
  --schedule "0 7 * * *" \  
  --http-method GET \  
  --uri https://...cloudfunctions.net/run_query \  
  --time-zone "Europe/Paris"
```

Cas d'usage professionnels

Cas d'usage	Déclencheur	Fonction Cloud
Générer des rapports de ventes tous les jours	Scheduler (7h)	Lancer une requête BigQuery
Supprimer les logs vieux de 30 jours	Scheduler (nuit)	Supprimer via requête BQ
Exporter une table vers Cloud Storage chaque semaine	Scheduler	Lancer <code>EXPORT DATA</code>
Charger un CSV dans BigQuery dès qu'il arrive	GCS (nouveau fichier)	Lancer <code>LOAD DATA</code> ou <code>LOAD JOB</code>
Alerter si une table dépasse un seuil	Scheduler	Vérifier une valeur + envoyer un mail

Lab 6

Nous allons utiliser la table publique suivante :

```
bigquery-public-data.faa.us_delay_causes
```

Elle contient les données de retards de vols aux États-Unis (par compagnie aérienne, date, cause...).

- Générer **chaque jour** un rapport des retards de vol par compagnie aérienne, uniquement pour les retards **supérieurs à 15 minutes**, et enregistrer le résultat dans une table `project_id.dataset.daily_flight_report`.

Lab 6

- Créez une requête SQL qui :
 - Sélectionne la date, la compagnie (`carrier`), le nombre de vols retardés (`num_delays`), et le type de retard
 - Ne conserve que les lignes avec `num_delays > 15`
 - Limite aux **30 derniers jours**
- ➡ *Cette requête sera utilisée dans la fonction Cloud.*

Lab 6

- Créer une **Cloud Function** (ex. en Python)
 - La fonction doit se connecter à BigQuery
 - Elle exécute la requête SQL et **crée ou remplace** une table `your_dataset.daily_flight_report`
 - Elle affiche un message de succès ou d'erreur

Lab 6

- Utilisez `gcloud functions deploy` avec :
 - `--runtime python310`
 - `--trigger-http`
 - `--region`
 - `--allow-unauthenticated` (ou authentication IAM si vous le souhaitez)



Lab 6

- Configurez une tâche `gcloud scheduler jobs create http` :
 - Planification : `"0 6 * * *"` (tous les jours à 6h)
 - Méthode : `GET`
- URL : celle de votre Cloud Function

Introduction aux UDF (User Defined Functions)

Une **User-Defined Function (UDF)** est une fonction personnalisée que vous définissez vous-même dans BigQuery pour encapsuler une logique que vous réutilisez souvent.

Elle peut être écrite en :

-  **SQL** (recommandé)
-  **JavaScript** (plus puissant, mais moins lisible et moins performant)

Introduction aux UDF (User Defined Functions)

- Réutiliser du code (ex : formatage de numéros, anonymisation)
- Simplifier des requêtes complexes
- Créer des transformations métier spécifiques
- Centraliser des logiques de validation

Introduction aux UDF (User Defined Functions)

```
CREATE OR REPLACE FUNCTION dataset.remove_accents(input STRING)
RETURNS STRING
AS (
    TRANSLATE(input, 'àâäéèêëîïôöùûüç', 'aaeeeeeiioouuuc')
);
```

 Appel dans une requête :

```
SELECT remove_accents('Crème brûlée');
-- → "Creme brulee"
```

Introduction aux UDF (User Defined Functions)

- **Forme SQL (recommandée pour simplicité)**

```
CREATE OR REPLACE FUNCTION dataset.nom_fonction(param1 TYPE1, param2 TYPE2)  
RETURNS RETURNTYPE  
AS (expression_sql);
```

 Exemple :

```
CREATE FUNCTION dataset.upper_trimmed(s STRING)  
RETURNS STRING  
AS (UPPER(TRIM(s)));
```

Introduction aux UDF (User Defined Functions)

- **Forme JavaScript (si besoin de plus de logique)**

```
CREATE OR REPLACE FUNCTION dataset.reverse_js(input STRING)
RETURNS STRING
LANGUAGE js
AS """
    return input.split('').reverse().join('');
    """;
```

 Appel :

```
SELECT reverse_js('hello');
-- → "olleh"
```



5. Utilisation dans une requête

```
SELECT  
  name,  
  remove_accents(name) AS normalized_name  
FROM  
  dataset.clients;
```

UDFs peuvent être utilisées dans les `SELECT`, `WHERE`, `GROUP BY`, etc.

Introduction aux UDF (User Defined Functions)

Cas métier	UDF Exemple
Nettoyer des noms	<code>remove_accents, upper_trimmed</code>
Masquer des emails	<code>mask_email(user_email)</code>
Calculs métiers	<code>calculate_discount(price, rate)</code>
Analyse linguistique	<code>count_words(text)</code>
Validation de format	<code>is_valid_siret(siret_number)</code>

Introduction aux UDF (User Defined Functions)

- **Préférer SQL** si possible (plus rapide, plus lisible)
- Donnez toujours un **dataset** spécifique pour stocker les fonctions (ex : `shared.udfs`)
- Utilisez un **nom clair et fonctionnel**
- Ajoutez des **tests** ou requêtes d'exemple avec les UDFs pour aider les autres utilisateurs

Lab 7

- **Créer une fonction** `mask_phone(phone_number STRING)`
- Elle doit retourner une version anonymisée du numéro :
- Exemple : `06 12 34 56 78` → `XX XX XX 56 78`

Puis utilisez-la dans une requête sur une table `dataset.customers`.

