Le rôle de l'analytics engineer

À mi-chemin entre le Data Analyst et le Data Engineer, l'Analytics Engineer est un profil de plus en plus recherché dans les entreprises modernes.

Contrairement au Data Analyst, dont la principale mission est l'analyse des données, l'Analytic Engineer met l'accent sur la modélisation des données. Concrètement, ce professionnel est responsable de la transformation, des tests, du déploiement et de la documentation des données qu'il gère.

Au quotidien, les missions d'un Analytics Engineer concernent :

- la conception et l'optimisation des pipelines de données pour l'acquisition, la transformation et le chargement (ETL/ELT) ;
- l'élaboration et la gestion de modèles de données pour simplifier leur analyse ;
- la surveillance et l'amélioration des performances des systèmes de traitement des données ;
- la mise en place de pratiques de gouvernance des données pour garantir leur qualité et leur sécurité.

L'analytics engineer joue un rôle clé dans une démarche de self-service analytics en préparant et structurant les données de manière à les rendre accessibles et compréhensibles pour les utilisateurs métiers.

Il s'assure que les données sont propres, organisées, bien documentées et disponibles via des outils de visualisation ou de reporting faciles à utiliser.

Ainsi, il crée une infrastructure qui permet aux utilisateurs non techniques d'explorer et d'analyser les données en toute autonomie, tout en garantissant la qualité et la gouvernance des données.

L'Analytics Engineer joue un rôle clé au sein d'une équipe data. Afin de s'assurer que les données soient prêtes pour l'analyse et la visualisation, il conçoit, développe et maintient des pipelines de données performants. L'Analytic Engineer travaille en étroite collaboration avec les membres de son équipe pour comprendre les besoins métier et les exigences en matière de données, dans le but de proposer des solutions adaptées.

Rappel SQL

Une formation SQL est disponible dans l'espace formation Quadratic.

Premières requêtes

Afficher ses données

Les mots clés de base pour afficher les données sont `SELECT` et `FROM`. A la suite de `SELECT` on listera les colonnes que l'on souhaite afficher et à la suite de `FROM`

```
SELECT
colonne1,
colonne2
FROM table1
```

Filtrer

Pour filtrer les données requêtées, on utilisera le mot clé `WHERE`

```
SELECT
colonne1,
colonne2
FROM table1
WHERE
colonne1 > 50
AND colonne2 LIKE 'quad%'
```

Ordonner

Le mot clé `ORDER BY` est utilisé pour ordonner les données.

```
SELECT
colonne1,
colonne2
FROM table1
WHERE
colonne1 > 50
AND colonne2 LIKE 'quad%'
ORDER BY colonne1 DESC
```

Agréger

L'aggrégation de données est le fait de regrouper les lignes d'une table selon une combinaison de colonne sous le mot clé `GROUP BY` en appliquant ou non une fonction d'aggrégation.

```
SELECT
SUM(colonne1) as total_colonne1,
colonne2
FROM table1
WHERE
colonne1 > 50
AND colonne2 LIKE 'quad%'
GROUP BY
colonne2
ORDER BY colonne1 DESC
```

Créer un schema CREATE TABLE

Nommer clairement : utiliser des noms descriptifs et cohérents pour les tables et les colonnes. Définir des types de données appropriés : choisir le type de données le plus adapté pour chaque colonne (par exemple, INT pour les numéros, DATE pour les dates). Utiliser des clés primaires : chaque table doit avoir une clé primaire pour identifier de manière unique chaque ligne. Utiliser des clés étrangères pour les relations : définir des relations entre les tables via des clés étrangères pour maintenir l'intégrité référentielle. Prévoir des contraintes pour garantir la qualité des données : comme les contraintes d'unicité, de vérification, etc. SQL - Le data definition language (DDL)

```
CREATE TABLE utilisateur
(
    id INT PRIMARY KEY NOT NULL,
    nom VARCHAR(100),
    prenom VARCHAR(100),
    email VARCHAR(255),
    date_naissance DATE,
    pays VARCHAR(255),
    ville VARCHAR(255),
    code_postal VARCHAR(5),
    nombre_achat INT
)
```

Altérer ses données INSERT, UPDATE, DELETE

Il est possible d'altérer les données d'une table avec les mots clés `INSERT`, `UPDATE`, `DELETE`.

- `INSERT` permet d'ajouter des enregistrements à une table.

```
INSERT INTO table VALUES ('valeur 1', 'valeur 2', ...)
```

- `UPDATE` permet de changer la structure de la tables comme ajouter une colonne.

```
UPDATE table
SET nom_colonne_1 = 'nouvelle valeur'
WHERE condition
```

- `DELETE` permet de supprimer des enregistrements à une table.

```
DELETE FROM `table`
WHERE condition
```

Les Jointures

INNER JOIN et LEFT JOIN

Les deux types de jointure les plus utilisés sont INNER JOIN et LEFT JOIN :

- INNER JOIN permet d'afficher les lignes en commun entre la table de gauche et la table de droite.

```
SELECT
t1.colonne1,
t2.colonne3,
t1.colonne2
FROM table1 t1
INNER JOIN table2 t2
ON t1.primarykeycol = t2.secondarykeycol
```

- LEFT JOIN permet d'afficher toutes les lignes de la table de gauche, qu'elles aient ou non une correspondance avec la table de droite.

```
SELECT
t1.colonne1,
t2.colonne3,
t1.colonne2
FROM table1 t1
INNER JOIN table2 t2
ON t1.primarykeycol = t2.secondarykeycol
```

RIGHT JOIN et FULL JOIN

- RIGHT JOIN permet d'afficher toutes les lignes de la table de droite, qu'elles aient ou non une correspondance avec la table de gauche.

Un RIGHT JOIN pourra toujours s'écrire avec un LEFT JOIN (table A LEFT JOIN table B équivaut à table B RIGHT JOIN table A).

Tout dépend de l'ordre dans lequel on souhaite écrire les tables dans la requête !

- FULL JOIN permet d'afficher toutes les lignes des tables de gauche et de droite.

Ce type de jointure est moins utilisé car il rallonge le temps de requête : elle affiche une table avec plus de colonnes (2 colonnes ID) et potentiellement plus de lignes.

Se repérer parmi les cloud providers, optimisation et bonnes pratiques SQL pour travailler en collaboration.

Du “On premise” vers le cloud

Avant

Les entreprises achetaient et géraient leurs propres serveurs physiques. Elles s'occupaient de faire évoluer leur matériel en achetant de nouveaux composants pour ajouter de la mémoire et/ou de la puissance. Enfin, elles faisaient la maintenance de leur matériel mais aussi les sauvegardes et étaient responsables de la sécurité des données.

Maintenant

Les entreprises se tournent de plus en plus vers des solutions cloud. Il existe trois types de services cloud :

- Les services IaaS (Infrastructure as a Service) : Ils permettent d'accèder à distance à des composants d'une structure informatique gérer par le cloud providers. Il n'y a donc pas besoin d'acheter, ni de configurer et il n'y a pas de maintenance à effectuer sur ces composants pour pouvoir bénéficier de ressources de calcul, de stockage etc ...
- Les services PaaS (Platforme as a Service) : Ils s'appuient sur les services IaaS pour fournir des outils et services nécessaires pour créer et déployer des applications. Ce sont par exemple les systèmes d'exploitations et les middlwares.
- Les services SaaS (Software as a Service) : Ce sont des applications prêtent à l'emploi accessible à distance et dont on a pas se soucier du développement et de déploiement comme Google Gmail, Docs ...

Les trois grands cloud providers

Un cloud provider est une entreprises qui fournit des ressources de calcul évolutive et accessible via le web. Les trois principaux cloud providers sont :

- Google
- Amazon
- Microsoft

Cloud Prodvider	Data lake	Data warehouse
Google	Google Cloud Storage	BigQuery
Amazon	Amazon S3	Redshift
Microsoft	OneLake	Azure SQL server

Chaque fournisseur a ses points forts et ses inconvénients. Les critères de choix des entreprises se font sur :

- la performance,
- le coût,
- la flexibilité.

Les grandes entreprises bénéficient en général de contrats personnalisés négociés avec les fournisseurs.

Distinguer Data Lake du Data Warehouse

Historiquement, l'ensemble des données d'une entreprise était contenue dans une base de données relationnelle (année 80). Ces bases de données, reposaient sur un langage simple à utiliser (SQL) et permettaient des opérations CRUD (Create, Read, Update, Delete) qui respectaient les propriétés ACID (Atomicity, Consistency, Isolation, Durability). On utilisait ces bases de données relationnelles pour créée des rapport quotidiens à des fins opérationelle.

Avec le temps, un besoin de réaliser des rapports analytiques a émergé. Ce besoin a nécessité d'avoir des données historiques pour projeter des tendances. Les Data Warehouses sont nés avec ce besoin. Ce sont de vaste bases de données historiques organisées par thème et sujets métiers. Ces bases de données ont garder les capacité OLTP des bases de données relationnaelles mais apport des fonctionnalités supplémentaires comme le processus ETL et des outils de traitement OLAP.

Bien que les data warehouse peuvent stocker des données non structurées, ils s'avèrent peu agile devant les demandes métiers de rajout de nouvelles sources et les changements de structures de sources. C'est à partir de cet instant que sont nés les Data Lakes qui se sont avéraient plus agiles que les Data Warehouse. En effet, en plus de pouvoir stocker de la données brutes structurés ou non, les changements de structure de ces données sont rapidement pris en charges. On passe d'un schéma on write (créer un schéma pour la donnée avant de l'écrire sur la base de données) à un schéma on read (on crée le schéma d'écriture seulement en lisant la donnée).

Pour requêter un data lake on utilise un langage NoSQL.

Data lake

Un data lake est un entrepôt de données où vous pouvez stocker tous vos types de données – qu'elles soient structurées (comme des tableaux de chiffres), semi-structurées (comme des fichiers XML ou JSON), ou non structurées (comme des vidéos, des images, des fichiers audio)

Data warehouse

Un data warehouse est un grand entrepôt de données centralisées. Il est conçu pour stocker des données structurées provenant de diverses sources et les rendre faciles à analyser pour obtenir des informations utiles.

Tableau comparatif

Caractéristiques	Data lake	Data warehouse
Type de données	Structurées, semi-structurées, non structurées	Principalement structurées
Organisation	Données brutes et non organisées	Données organisées et pré-structurées
Stockage	Stockage à faible coût pour grandes quantités	Stockage plus coûteux en raison de l'optimisation
Scalabilité	Très évolutif et flexible	Evolutif, mais à un coût plus élévé
Coût	Moins cher à stocker des données brutes	Plus cher en raison de l'optimisation et de la structure

Pourquoi optimiser ces requêtes

- Réduction des coût : malgré la baisse du coût du stockage, il peut être intéressant d'optimiser ces requêtes pour qu'elles consomment moins de ressources afin de diminuer les coûts d'exploitation.
- Amélioration des performances : les dashboards et les rapports se chargent plus rapidement, ce qui offre une meilleure expérience à l'utilisateur.
- Gain de temps : les équipes peuvent accéder aux données plus rapidement, ce qui accèlère le processus de prise de décision.

Partitionnement

Le partitionnement est une méthode d'indexation propre à BigQuery, les autres base de données et les cloud providers proposent des fonctions similaires. Le partitionnement divise une table en segments plus petits, appelés partitions. Chaque partition correspond généralement à une colonne, souvent des dates, car elles répartissent les données de façon uniforme et sont fréquemment utilisées dans les filtres des requêtes. Le partitionnement permet à BigQuery de ne scanner que les partitions concernées, améliorant ainsi les performances.

Clustering

Le clustering organise physiquement les données à l'intérieur de la table en fonction des colonnes choisies. Par exemple, en clusterisant par pays, les données sont triées par pays, ce qui permet un accès plus rapide aux lignes filtrées par cette colonne. Le clustering est souvent utilisé conjointement avec le partitionnement

Bonnes pratiques

Naming

Dans l'idéal, il faut être descriptif dans les noms des colonnes, des cte et des alias de tables même si le nom est long : ceci_est_un_exemple_de_nom.

Utilisation des CTE

Les CTEs améliorent la lisibilité des requêtes complexes en les décomposant en étapes logiques.

Utilisation d'une indentation cohérente et homogène

On utilisera par exemple l'indentation sur les différentes clauses SELECT, FROM, WHERE, GROUP BY, etc.

Ajouter des espaces entre les opérateurs

Dans les conditions, le fait d'espacer les opérateurs (comme =, <>, >, <, etc.) améliore la lisibilité.

Ne pas faire de group by imprécis

Dans Bigquery, il est possible d'utiliser `GROUP BY ALL` ou `GROUP BY 1,2,3`.

Par exemple,

```
SELECT
categories,
zone,
COUNT(products)

FROM table
GROUP BY 1,2
```

Dans cet exemple, on arrive facilement à identifier les colonnes qui aggrègent nos données. Cependant, pour rendre le code plus lisible, on nommera les colonnes d'aggrégation dans notre clause `GROUP BY`.

Utiliser un maximum les linters

Bouton "Format" dans Bigquery Dans la console google cloud, il faut se rendre sur la page Bigquery. En cliquant sur "Saisir une nouvelle requête", l'éditeur de requête s'ouvre. Après avoir saisi sa requête, il faut cliquer sur le bouton "Plus" puis "Formatter la requête".

Azure SQL Dans SQL Server Management Studio, il est possible d'installer un plugin gratuit pour formatter ces requêtes. Il est disponible à cette adresse.

Redshift Il n'existe pas de client SQL développé par Amazon pour leurs bases de données Amazon Redshift. Le plus simple est d'utiliser un client SQL générique comme SQL Workbench ou DBeaver qui ont une option de formattage des requêtes pré installée.

SQLFlluff en local SQLFluff est un linter open-source, multi dialect et configurable. C'est un package python qui s'install via `pip install sqlfluff` La documentation se trouve à cette adresse.

Data Build Tool : L'outil indispensable de l'AE

1. Présentation de dbt

   dbt est un outil de modélisation de données qui incarne parfaitement l'approche analytics engineering.

   Il est massivement adopté par les entreprises, comme en témoigne sa fréquence de citation dans les retours d'expérience (ex. : podcast DataGen).

2. Centralisation et fiabilité

   dbt permet de centraliser tous les modèles SQL au même endroit, organisés dans des dossiers et sous-dossiers.

   Cela améliore la fiabilité des données : plus de requêtes dispersées, plus de doublons, tout est standardisé et versionné.

3. Travail collaboratif

   Tous les analystes accèdent au même référentiel SQL, ce qui facilite le travail collaboratif et évite les divergences de méthodes ou de résultats.

4. Accessibilité pour les profils SQL

   dbt est conçu pour être utilisé par des profils non-développeurs, comme les data analysts qui maîtrisent uniquement le SQL.

   Cela rend ces profils autonomes pour transformer les données, en complément de leur capacité à les ingérer (via des outils comme Fivetran).

5. Profil de l'analytics engineer

   Ces outils permettent l'émergence de nouveaux profils : les analytics engineers ou data analysts full stack, capables de maîtriser toute la chaîne de la donnée (ingestion + modélisation).

Installation

Pré-requis:

1. Windows Subsystem for Linux –> Choisissez Ubuntu
2. UV
3. Avoir un compte d'essai gratuit Google Cloud ici

Dbt Core avec dagster

dbt Core est la version open source à installer et utiliser en local. Dagster est un data orchestrator au même titre qu'airflow mais il est plus simple à mettre en place avec dbt.

Dans le terminal ubuntu, on crée notre dossier de travail avec `uv init nom_du_projet` et on se positionne dans notre dossier de travail avec `cd nom_du_projet`.

1. `uv add dagster-dbt dbt-bigquery` : On install dagster avec dbt et son adapteur pour BigQuery.
2. `uv run dbt init` : On crée notre dossier de travail dbt avec ces sous-dossiers.

a. On donne un nom à notre projet dbt
b. On selectionne l'adaptateur que l'on va utiliser. En l'occurence ici, on utilisera l'adaptateur BigQuery.
c. On sélectionne une méthode de connexion à bigquery. Je vous conseille la méthode service account qui demandera le chemin vers le fichier clé du service account. Pour obtenir se fichier, il faudra :

- Se rendre à ce lien
- Cliquer sur "Créer un compte de service" et lui donner un nom.
- A la seconde étape de création du compte de service, on donnera les rôles "Editeur de données BigQuery", "Utilisateur de job BigQuery" et "Utilisateur BigQuery".
- La troisième étape est facultative, on pourra donc cliquer sur "OK".

Une fois le compte de service créé, on cliquera sur ce compte de service et on génèrera une clé d'autorisation JSON. Un fichier JSON sera téléchargé. On aura plus qu'à copier/coller le chemin d'accès à ce fichier à l'étape de configuration dbt.

d. On donne le nombre de coeurs CPU que DBT pourra utiliser.
e. On donne le nombre du projet bigquery et le nom du dataset bigquery.

f. On sélectionne la même zone géographique US ou EU pour dbt, notre projet bigquery et notre dataset.

Dbt Cloud

Pour utiliser dbtCloud, la version Saas avec une interface web de DBT, on faudra créer un compte d'essai gratuit. La configuration d'un projet dbtcloud passe par les étapes ci-dessous :

1. Se connecter à une platforme

a. Cliquer sur "add new connection"
b. Choisir la platforme Bigquery
c. Charger la clé JSON du compte de service que l'on a créée précédement pour dbt core (étape 2c)
d. Terminer en cliquant sur save.

De l'aide est disponible, à ce lien

2. Configurer l'accès à Github, Gitlab ou notre propre repository git. Sur la page proposant de configurer un repository git, on choisira l'option "Managed" pour que Git soit directement géré dans console dbt cloud.

Pour finir, on choisira un nom pour notre projet dbt et il faudra cliquer sur bouton "Create".

Dbt et la modélisation

Un projet dbt est constitué d'un ensemble de dossiers :

- models : Ce dossier contiendra les requêtes SQL et des fichier YAML qui mettront en place des tests sur nos models et les documenteront.
- snapshots : Ce dossier contiendra les fichiers de configurations des snapshots de tables.
- seeds : Ce dossier contiendra des fichier csv qui pourront être chargés dans la plateforme.
- tests : Ce dossier contiendra des tests personnalisables sur nos models.
- analysis : Ce dossier contiendra des requêtes SQL d'analyse supplémentaires.
- macros : Ce dossier contiendra des fichiers YAML de création de macros jinja. Ce code pourra être utilisé dans les models.

Le fichier dbt_project.yml est le fichier principal de configuration du projet dbt. Il contient des informations qui détermine comment dbt fonctionne dans notre projet. Par exemple des informations sur la materialisation de nos models en vue ou en table, sur les tests à appliquer...

Un model est un fichier sql contenant une requête amélioré avec du code jinja. Ce code jinja est simple à utiliser et est identifiable par les deux accolades qui l'entoure. Pour générer la table issue du model "my_first_model.sql" on

utlisera la commande `dbt run --select:my_first_model` (il ne faut pas oublier de mettre `uv` avant si vous utiliser dbtcore). A la fin de cette opération, on trouvera dans notre dataset BigQuery une nouvelle table portant le nom "my_first_model".

Avant de créer son premier modèle, on va créer nos tables sources et les identifier dans un fichier source.yml.

- Création des tables dans bigquery

Vous trouverez le fichier Chinook_BigQuery.sql qui va créer les tables chinook et insérer données à l'intérieur.Il suffira de copier le script et de l'exécuter dans BigQuery studio. Un nouvel ensemble de données (=dataset) sera créé contenant les tables Album, Artist etc. Ces tables sont décrites dans la formations quadratic SQL.

- Création du fichier source.yml dans le dossier "model" pour identifier nos sources.

Identifier les sources dans un fichier yml permet d'utiliser la fonction jinja `{{source(source\_name,table_name)}}` dans nos models.

Cette référence évite par exemple d'écrire dans notre model `FROM db_name.dataset.table`. De plus, dans ce fichier source il est possible de décrire les tables sources et leurs colonnes pour générer ensuite une documentation. Enfin, il est possible d'appliquer des tests génériques sur nos tables sources.

Dans l'exemple ci-dessous, notre dataset source "chinook" sera référencé par le nom "chin_db". Ensuite, les tables tables contenues dans dataset sont listées et un test générique sera appliqué sur la colonne "AlbumId" de la table "Album". Ce test vérifiera si la colonne clé "AlbumId" n'est pas null et n'a pas de doublons lors de l'utilisation de commandes comme "uv dbt run" ou "uv dbt test".

```
version: 2

sources:
  - name: chini\_db
    schema: chinook
    tables:
      - name: Album
        description: Une table qui contient les informations concernant les albums
        columns:
          - name: AlbumId
            description: Identifiant unique d'un album
            tests:
              - unique
              - not\_null
      - name: Artist
      - name: Customer
```

- Notre premier modèle

Pour faciliter la lisibilité de notre project dbt, il est nécessaire de le structurer selon trois couches :

Staging : Les modèles de staging sont la première couche de transformation dans un projet DBT. Ils consistent en des transformations légères de vos sources de données brutes pour les préparer à des transformations plus complexes. Les modèles de staging traitent généralement une seule source à la fois et normalisent les noms des colonnes, suppriment les lignes en double, convertissent les types de données, etc.

Intermediate : Les modèles intermédiaires sont la deuxième couche de transformation. Ils effectuent des transformations plus complexes et peuvent incorporer plusieurs sources de données. Le but de ces modèles est de préparer les données pour les modèles mart.

Mart : Les modèles mart (ou data marts) sont la couche finale de transformation. Ils créent des vues finales des données qui sont optimisées pour l'analyse et la visualisation. Ces modèles consolident les informations provenant de diverses sources et les présentent dans un format facilement compréhensible et utilisable.

Nous allons donc créer dans dossier "models" un premier sous-dossier "staging" qui contiendra notre premier model nommé '_stg___albums.sql'

Notre code ...

```
SELECT
  AlbumId,
  Title,
  ArtistId,
  INITCAP(Title) AS _cleaned_title,
  CHAR_LENGTH(Title) AS _nb_char_in_title
FROM
        {{source('chin_db','Album')}}
```

sera compilé par dbt ...

```
SELECT
  AlbumId,
  Title,
  ArtistId,
  INITCAP(Title) AS _cleaned_title,
  CHAR_LENGTH(Title) AS _nb_char_in_title
FROM
        `bq_project_name`.`chinook`.`Album`
```

et une vue sera générée dans le dataset "dbt_quad".

La documentation et les tests dans DBT

Après avoir vu comment modéliser les données avec dbt, ce nouveau chapitre aborde les étapes de documentation et de tests dans DBT.

En effet, dbt permet de documenter les modèles de données pour : - améliorer la lisibilité, - faciliter la collaboration, - garder une trace claire de chaque transformation et de sa logique métier.

Ensuite, dbt inclut des fonctionnalités de test intégrées pour garantir la qualité et la fiabilité des données : - tests de non-nullité, - tests d'unicité, - tests de relations entre tables, etc.

Enfin, il est possible de personnaliser le comportement de chaque modèle via des fichiers de configuration comme la gestion de la materialisation (table, vues, incremental), la priorisation des modèles et la gestion de rafraichissement.

La documentation dans DBT

Documenter les données est essentiel pour améliorer la communication entre équipes et garantir l'indépendance de celles-ci. Cela facilite aussi la maintenabilité du code. dbt permet de documenter les modèles via des fichiers *schema.yml*. Ces fichiers contiennent des descriptions des modèles et de leurs colonnes clés, en particulier celles spécifiques aux besoins métiers. dbt génère automatiquement un site web avec la commande dbt docs generate pour faciliter la visualisation et diffusion de cette documentation.

Une documentation bien faite répond aux questions récurrentes des équipes métiers, comme le calcul des KPIs, et évite des échanges répétitifs. Elle accélère également l'onboarding des nouvelles recrues dans une équipe data, car elles peuvent directement accéder aux informations nécessaires sur les modèles et données.

La documentation générée peut être visualisée localement via la commande dbt docs serve, ou directement dans dbt Cloud sans avoir besoin de la générer manuellement. Un des éléments clés de cette documentation est le lineage, une vue en arborescence qui montre les dépendances entre modèles, facilitant la compréhension des relations entre eux.

Il y a deux façons d'écrire de la documentation, la première est de rédiger un fichier.yml pour tous les modèles. Par exemple, dans staging_mod_docs.yml,

```
version: 2



models:
  - name: stg_chin__albums
    description: "This model contains information about albums"
    columns:
```

```
    - name: AlbumId
      description: "Primary key, unique identifier for each album"
      tests:
          - unique
          - not_null
    - name: Title
      description: "information about titles of albums"
    - name: ArtistId
      description: "Foreign key linking the album to the corresponding artist."
      tests:
        - relationships:
            name: artist_id_foreign_key_in_stg_chin_album
            to: ref('stg_chin__artists')
            field: ArtistId
    - name: _cleaned_title
      description: "Name of the album with the first letter in upper case"
    - name: _nb_char_in_title
      description: "Count the number of character in the album title"


- name: stg_chin__artists
  description: "This model contains information about artists"
  columns:
    - name: ArtistID
      description: "Primary key, unique identifier for each artist."
      tests:
          - unique
          - not_null
    - name: Name
      description: "information about the name of the artists"
    - name: _cleaned_name
      description: "Make sure the name of the artist start with an upper case"
    - name: _nb_char_name
      description: "Count the number of character in the name of the artist"
    - name: _snd_name
      description: "Convert the name of the artist in sound characters"
```

on a la description de deux modèles dans le même fichier. Ce fichier .yml se structure de la manière suivante :

```
- name: nom_du_modele
  description: description du modèle
  columns: --> Les colonnes du modèle
      - name: nom_de_la_colonne
        description: description de la colonne
```

Une autre manière d'écrire de la documentation est d'écrire des blocs de code

markdown qui seront ensuite référencés dans le fichier .yml. Ces blocs markdown sont plus personnalisables (caratère en gras, italique, ... puces etc...) que la propriété "description" des fichiers yaml.

Par exemple, dans le dossier des modèles "intermediate" le fichier int_chin___users_invs.md :

```
{% docs int_chin__users_invs %}

This model provides a summary of user activity and invoices, focusing on their order history

    Total Amount Spent: The total amount spent by the user.
    Total Tracks: The total quantity of tracks purchased by the user.
    Total Distinct Tracks: The count of distinct tracks purchased by the user.
    Total Orders: The total number of orders placed by the user.
    Dominant State : The state where most tracks are purchased
    Number of employee to manage : The number of employee by managers.
    Average unit price : The average of unit price of tracks sold.
    Total Distinct Customers : The count of distinct customer by tracks.
    Total Support Solicitations : The number of times support is asked.
    In top ten : True/False field to see if the track is in top ten of sales.

{% enddocs %}
```

Un ou plusieurs blocs markdown peuvent être écrits pour être référencé dans un fichier .yml comme int_chin___employees.yml

```
version: 2

models:
  - name: int_chin__employees
    description: '{{ doc("int_chin__users_invs") }}'
    columns:
      - name: CustomerId
        description: "Primary key, unique identifier for each track."
        tests:
          - unique
          - not_null
      - name: _cust_full_name
        description: "The full name of the customer"
      - name: Country
        description: "The country of the manager"
      - name: total_support_solicitations
        description: "the total of solicitations of the support"
      - name: _manager_full_name
        description: "The full name of the manager"
      - name: nb_employees_to_manage
        description: "The number of employees manage by the manager"
```

Au niveau de la propriété “description” du modèle, on trouve une référence à notre bloc markdown. Cette référence peut être réutilisée dans plusieurs fichiers YAML.

Les tests dans DBT

Le fait de tester les données permet de garantir leur qualité et d'éviter des erreurs courantes dans les dashboards d'entreprise (écarts de chiffres entre tableaux, erreurs de segmentation des données, etc.). En plus des tests basiques, des tests plus avancés comme la détection d'anomalies peuvent être réalisés pour intervenir avant que les données problématiques ne soient chargées en base.

Les tests dans dbt sont des affirmations sur les données. Par exemple : “chaque commande doit avoir un montant supérieur à zéro” ou “chaque utilisateur doit avoir un user_id unique et non nul”. Deux types de tests existent :

- Tests singular : Des requêtes SQL personnalisées, stockées dans le dossier /test du projet.
- Tests generic : Des tests standards (unicité, valeurs acceptées, non null, etc.) définis dans les fichiers YAML. Ces tests sont pré-codés dans dbt et ne nécessitent pas d'écrire de SQL.

Les tests s'exécutent avec la commande `dbt test`. Il est possible de lancer tous les tests ou de se limiter à un modèle spécifique avec `dbt test --select nom_modèle`. Si un test échoue, dbt renvoie une liste des erreurs pour les analyser et les corriger.

Les tests génériques Les tests génériques sont à intégrer dans le fichier yaml contenant la documentation. Dans staging_mod_docs.yml,

```
version: 2



models:
  - name: stg_chin__albums
    description: "This model contains information about albums"
    columns:
      - name: AlbumId
        description: "Primary key, unique identifier for each album"
        tests:
            - unique
            - not_null
      - name: Title
        description: "information about titles of albums"
      - name: ArtistId
        description: "Foreign key linking the album to the corresponding artist."
        tests:
```

```
          - relationships:
              name: artist_id_foreign_key_in_stg_chin_album
              to: ref('stg_chin__artists')
              field: ArtistId
      - name: _cleaned_title
        description: "Name of the album with the first letter in upper case"
      - name: _nb_char_in_title
        description: "Count the number of character in the album title"
```

on trouve l'implémentation d'un test d'unicité et d'un test non-null sur la colonne de clé primaire AlbumId. Sur la colonne de clé secondaire ArtistId, on trouve un test de relation qui va tester l'intégrité référentielle de notre relation. Enfin, dans le modèle stg_chin___genre, sur la colonne "Name", on trouve le dernier type de test générique qui vérifie si les valeurs de notre colonne sont bien contenue dans la liste spécifié par le test.

```
  - name: stg_chin__genre
    description: "This model contains details of products available for sale, primarily focu
    columns:
      - name: GenreId
        description: "Primary key, unique identifier for each genre of music."
        tests:
            - unique
            - not_null
      - name: Name
        description: "The label of the genre"
        tests:
          - accepted_values:
              values: ['Rock', 'Science Fiction', 'Drama', 'Alternative & Punk','Pop','Metal
                'Soundtrack','Sci Fi & Fantasy','Blues','R&B/Soul','Rock And Roll','Electron
'TV Shows','Jazz','Heavy Metal','Opera','Bossa Nova','Classical','Alternative','Reggae','Eas
      - name: _cleaned_name
        description: "The label of the genre with the first letter in upper case"
      - name: _nb_char_name
        description: "The number of characters of the genre"
```

Les tests singuliers Les tests singulier sont implémentés dans le dossier "tests" de notre projet dbt.

Par exemple, le test quantity_positive.sql lance une requête sql sur le modèle stg_chin___invoice_lines pour véfifier si les valeurs de la colonne "Quantity" ne sont pas négatives. Le test en lui-même repose sur une requête sql qui cherche des valeurs négatives. S'il en trouve une, le test échoue.

```
select
    InvoiceLineId,
from  {{ ref('stg_chin__invoice_lines.sql') }}
```

```
where Quantity < 0
```

Travailler en collaboration avec Git

En entreprise, l'utilisation de Git comme système de gestion de versions est essentielle pour faciliter la collaboration entre les membres d'une équipe de développement. Git permet à plusieurs développeurs de travailler simultanément sur un projet sans risquer de perdre ou d'écraser le travail des autres.

Une des pratiques clés associées à Git est le "peer review" ou revue de code par les pairs. Cette approche collaborative consiste à soumettre les modifications de code sous forme de "pull requests", permettant ainsi aux autres membres de réviser, commenter et suggérer des améliorations avant l'intégration du code dans la branche principale. Ce processus favorise non seulemment la qualité du code, mais aussi le partage des connaissances et de bonnes pratiques au sein de l'équipe.

De plus, Git offre des fonctionnalités comme les "branch" et le "merge", qui aident à gérer les différents versions du projet et à intégrer les contributions de manières organisée et contrôlée. Ainsi, Git devient un outil indispensable pour maintenir un flux de travail efficace et collaboratif dans un environnement professionnel.

Rappel Git

Git est un outil puissant pour le contrôle de version, et maîtriser ses commandes essentielles est crucial pour une utilisation efficace. Parmi les commandes les plus importantes, *git clone* permet de copier un dépôt distant sur votre machine local, vous permettant de commencer à travailler sur un projet.

git add esst utilisé pour ajouter des modifications ajoutées dans l'index (=staging) préparant ainsi les fichiers pour un commit.

Avec *git commit*, vous pouvez enregistrer les modifications ajoutées dans l'index avec un message décrivant les changements effectués.

git pull et *git push* sont essentiels pour la synchronisation avec un dépôt distant : *git pull* récupère les modifications du dépôt distant et les fusionne avec votre copie locale, tandis que *git push* envoie vos commit locaux vers le dépôt distant.

git branch est utilisé pour lister, créer ou supprimer des branches, ce qui est fondamental pour travailler sur différentes fonctionnalités ou corrections de bugs de manière isolée.

Enfin, *git merge* permet de fusionner les modifications d'une branche dans une autre, facilitant ainsi l'intégration des fonctionnalités développées séparément. Ces commandes forment la base pour une utilisation efficace de Git dans un environnement de travail collaboratif.

Une formation certifiante Git Kraken est disponible pour valider les commandes essentielles ci-dessus.

Apporter ces modifications à la branche principale

Par convention, on travaille pas directement sur la branche “master” ou “main” d’un projet. *git checkout -b nom-branch* permet de créer et naviguer entre les branches d’un projet. Une fois les modifications ajoutées, enregistrées dans l’index et envoyées sur le dépôt distant, il faudra terminer par fusionner la branche de travail et la branche priincipale.

Workflow Git

1. Je mets à jour ma branche master/main

git checkout master

git pull

2. Je crée une branche

git checkout –b ma-branche-avec-une-nouveaute

3. Je fais mes modifications sur les fichiers du projet, j’ajoute mes modifications au dossier staging et je les sauvegarde avec un commit

git add .

git commit –m «mon message de commit »

4. J’envoie mes modifications sur le serveur Github/Gitlab

git push origin ma-branch-avec-une-nouveaute

5. Sur Github ou GitLab je crée une pull request : La PR déclenchera la CI/CD, qui doit être validée sans échec avant de passer à la prochaine étape.

6. Je demande une review à un collaborateur une fois la CI/CD passée. Si des changements sont demandés, les appliquer avant de finaliser.

7. Une fois validée, Je fusionne la PR avec la branche de production.

Git merge with main

La commande *git merge master* intègre les changements récents de master dans votre branche actuelle. Elle génère un commit de fusion pour rassembler les modifications.

Git rebase

La commande *git rebase master* applique vos commits après ceux de master, produisant un historique linéaire et ordonné des commits. Cela simplifie la lecture de l'historique, mais peut créer des conflits qu'il faut résoudre avant de finaliser le rebase.

Gérer les conflits git

Lors de merges ou de rebase, des conflits peuvent apparaître si des modifications ont été apportées aux mêmes fichiers sur différentes branches. Pour les résoudre, vous pouvez choisir quelle version garder ou fusionner les modifications pour conserver les deux versions.

Les conflits peuvent être gérés directement dans un IDE comme VSCode ou dans l'interface GitHub, et ils doivent être résolus avant de finaliser la fusion.

A. Cas pratique : Gérer un conflit lors d'un git merge

Dans un dépôt local : 1. Création d'une branche ajout-bonjour *git checkout -b ajout-bonjour*. 2. Création d'un fichier "conflict.txt" contenant sur la première ligne "Bonjour et bienvenue dans ce cas pratique". 3. *git add .*, *git commit "mon nouveau fichier bonjour"*, *git push*

Sur Github : 4. Création et validation de la pull request et fusion de la branche ajout-bonjour à main

Dans le dépôt local : 5. Création d'une branche add-hello *git checkout -b add-hello*. 6. Création d'un fichier "conflict.txt" contenant sur la première ligne "Hello and welcome to this pratice case". 7. *git add .*, *git commit "my new file hello"*, *git push*

Sur Github : 10. Résolution du conflit avec l'interface Github en cliquant sur "Resolve conflit" et une fois qu'on a terminé "Commit merge"

Dans le dépôt local : 11. *git checkout main* 12. *git pull*

B. Cas pratique : Gérer un conflit lors d'un git rebase

Dans un dépôt local : 1. Création d'une branche ajout-bonjour *git checkout -b ajout-aurevoir*. 2. Création d'un fichier "conflict2.txt" contenant sur la première ligne "Merci d'avoir suivi ce cas pratique, au revoir". 3. *git add .*, *git commit "mon nouveau fichier au revoir"*, *git push* 4. Création et validation de la pull request et fusion de la branche ajout-aurevoir à main 5. Création d'une branche add-goodbye *git checkout -b add-goodbye*. 6. Création d'un fichier "conflict2.txt" contenant sur la première ligne "Thanks for following this practice case, bye". 7. *git add .*, *git commit "my new file goodbye"*, *git push* 8. Récupération des modifications récemment fusionnées sur main sur la branche "add-goodbye" *git fetch origin* 9. Application des modifications de "main" sur la branche "add-goodbye" avec *git rebase origin/main* 10. Résolution du conflit avec un éditeur

de texte. 11. Ajout des modifications à l'index *git add conflict2.txt* 12. *git rebase continue* et *git push origin add-goobye –force*

Sur github : 13. Création et validation de la pull request et fusion de la branche ajout-goodbye à main

Mettre son code en production

Les pull requests : Bonnes pratiques

Une Pull Request (PR) regroupe les modifications apportées et propose leur intégration sur la branche principale. Elles doivent être créées sur le dépôt distant à la suite de la commande *git push branche-de-travail* lancée depuis le dépôt local.

Elle doit être accompagnée d'une description claire pour expliquer les changements. En entreprise, les exigences pour valider une PR peuvent varier : familiarisez-vous avec les conventions dès votre arrivée.

Concernant les branches, utilisez des lettres minuscules, des tirets pour séparer les mots, et, si possible, le numéro de ticket associé (ex. : de123-feature-update). Dans les commit, précisez l'action avec des verbes et ajoutez un préfixe pour clarifier l'intention (ex. : fix pour corriger un bug, feat pour une nouvelle fonctionnalité, docs pour documenter). Enfin, comme pour les branches et commits, le titre de la PR doit être explicite, et la description doit inclure les détails nécessaires pour une review efficace. Il est utile d'inclure le lien vers le ticket Jira ou Notion associé.

Paramétrer son dépôt distant au code review

Sur Github dans notre dépot distant :

1. Se rendre dans l'onglet "Settings"
2. Se rendre dans les options "Branches"
3. Cliquer sur "Add classic branch protection rule"
4. Désigner la branche sur laquelle on souhaite appliquer les règles, en l'occurence "main"
5. Cocher les règles suivantes :
    - Require a pull request before merging : Empèche la modification de la branche main directement
        - Require approval : Nécessite une approbation d'un tiers
        - Require review from code owners
    - Require conversation resolution before merging
    - Lock branch
6. Cliquer sur "Create"

Mettre en place une CI/CD

Définition

Le CI/CD (Continuous Integration/Continuous Deployment) est un ensemble de workflows automatisés déclenchés lors de la création ou la mise à jour d'une pull request. La CI vérifie la qualité du code (format, compilation), tandis que la CD déploie le code validé sur des environnements intermédiaires comme staging ou dev avant la production.

Au cours de sa mission l'analytics engineer peut rencontrer ces erreurs :

- Code SQL mal formaté : les entreprises utilisent souvent des linters comme SQLFluff. Si le format est incorrect, un échec sera signalé.
- Code dbt non compilable : le code dbt peut parfois ne pas compiler correctement ; la CI/CD détectera cette erreur.
- Échec de tests dbt : si des tests dbt sont intégrés dans le CI/CD, tout test échoué sera signalé.
- Gestion incorrecte des dépendances : un renommage de colonne peut provoquer des erreurs dans les modèles dbt dépendants. Le CI/CD alerte alors sur ces ruptures de dépendance.

La CI/CD prévient les erreurs en production grâce à des vérifications automatisées et permet une collaboration fluide en intégrant ces checks dans les workflows de développement. Son objectif est de garantir que le code livré soit stable et conforme aux standards de qualité de l'entreprise.

Github actions

GitHub Actions est un outil d'intégration continue et de livraison continue (CI/CD) qui permet d'automatiser les workflows directement dans vos dépôts GitHub. Avec GitHub Actions, vous pouvez créer des workflows personnalisés pour construire, tester et déployer votre code chaque fois qu'un événement spécifique se produit, comme un push ou une pull request. Cela aide à s'assurer que votre code est toujours testé et déployé de manière cohérente et fiable.

Pour mettre en place GitHub Actions dans un projet, commencez par créer un répertoire nommé `.github/workflows` à la racine de votre dépôt. Dans ce répertoire, créez un fichier YAML pour définir votre workflow. Par exemple, vous pouvez créer un fichier nommé `my_workflow.yml`. Dans ce fichier, vous définissez les événements qui déclenchent le workflow, comme `push` ou `pull_request`, et spécifiez les jobs à exécuter. Chaque job est composé de plusieurs étapes (steps) qui peuvent inclure des actions préexistantes disponibles sur le GitHub Marketplace ou des scripts personnalisés.

Voici un exemple simple de fichier YAML pour un workflow qui exécute des tests à chaque push :

```
name: CI
```

```
on: [push]

jobs:
  jobname:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Run a one-line script
      run: echo Hello, world!
    - name: Run tests
      run: npm test
```

Dans cet exemple, le workflow est déclenché à chaque push vers le dépôt. Il utilise un container avec la dernière version d'Ubuntu, effectue un checkout du code, exécute une commande simple pour afficher "Hello, world!", et enfin, exécute les tests avec npm. Une fois votre fichier YAML créé et poussé vers votre dépôt, GitHub Actions exécutera automatiquement le workflow selon les événements spécifiés.

Pour contrôler et tester notre code on écrira ce workflow :

```
name: CICD DBT
# https://github.com/marketplace/actions/dbt-action
# https://docs.github.com/en/actions/writing-workflows/quickstart
run-name: ${{ github.actor }} is opening a pull request

# Run jobs when a pull request is created
on: [pull_request]

jobs:
  action:
    # Create an Ubuntu container
    runs-on: ubuntu-latest

    steps:
      - run: echo " The job was automatically triggered by a ${{ github.event_name }} event.
      - run: echo " This job is now running on a ${{ runner.os }} server hosted by GitHub!"
      - run: echo " The name of your branch is ${{ github.ref }} and your repository is ${{
      # Clone the repos
      - name: Checkout repository
        uses: actions/checkout@v4
      - run: echo " The ${{ github.repository }} repository has been cloned to the runner."
      - run: echo " The workflow is now ready to test your code on the runner."
      # Run dbt
      - name: dbt-run
        uses: mwhitaker/dbt-action@master
        with:
```

```
      # Get latest dependancies
      dbt_command: "dbt deps"
      # Run dbt on the latest changes with our profile
      dbt_command: "dbt run - select +state:modified+ - defer - state manifest_file_fold
      dbt_project_folder: "uv_dag_dbt_bq"
    env:
      # BigQuery credentials in secrets of our github project
      DBT_BIGQUERY_TOKEN: ${{ secrets.DBT_BIGQUERY_TOKEN }}
  - name: List files in the repository
    run: |
      ls ${{ github.workspace }}
  - run: echo " This job's status is ${{ job.status }}."
```

Conclusion

En conclusion, cette formation d'Analytics Engineer nous a permis d'acquérir des compétences essentielles pour transformer des données brutes en informations exploitables grâce à des outils puissants tels que dbt, BigQuery et Git.

Nous avons appris à structurer et à modéliser des données de manière efficace avec dbt, à exploiter la puissance de BigQuery pour effectuer des analyses à grande échelle, et à collaborer de manière optimale grâce à Git.

Ces compétences nous positionnent favorablement pour relever les défis du monde de la data et contribuer de manière significative à la prise de décision stratégique au sein des organisations. Nous sommes désormais équipés pour concevoir des pipelines de données robustes, assurer leur maintenance et favoriser une culture de la donnée au sein des équipes.