

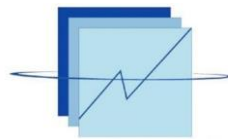
RÉPUBLIQUE DU SÉNÉGAL

Un Peuple - Un But - Une Foi



Ministère de l'Économie, du Plan et de la Coopération

Agence nationale de la Statistique et de la Démographie (ANSD)



Ecole nationale de la Statistique et de L'Analyse économique Pierre NDIAYE (ENSAE)



BIGDATA ET CLOUD COMPUTING:

**DOCUMENTATION SUR LA CREATION D'UN DATA
WAREHOUSE POUR L'ANALYSE DES VENTES**

Rédigé par :

Judicaël Oscar Gandwende KAFANDO

Ramatoulaye NDEYE NDOYE FALL

Djerakei MISTALENGAR

Ben Idriss Diloma SOMA

Sous la supervision de :

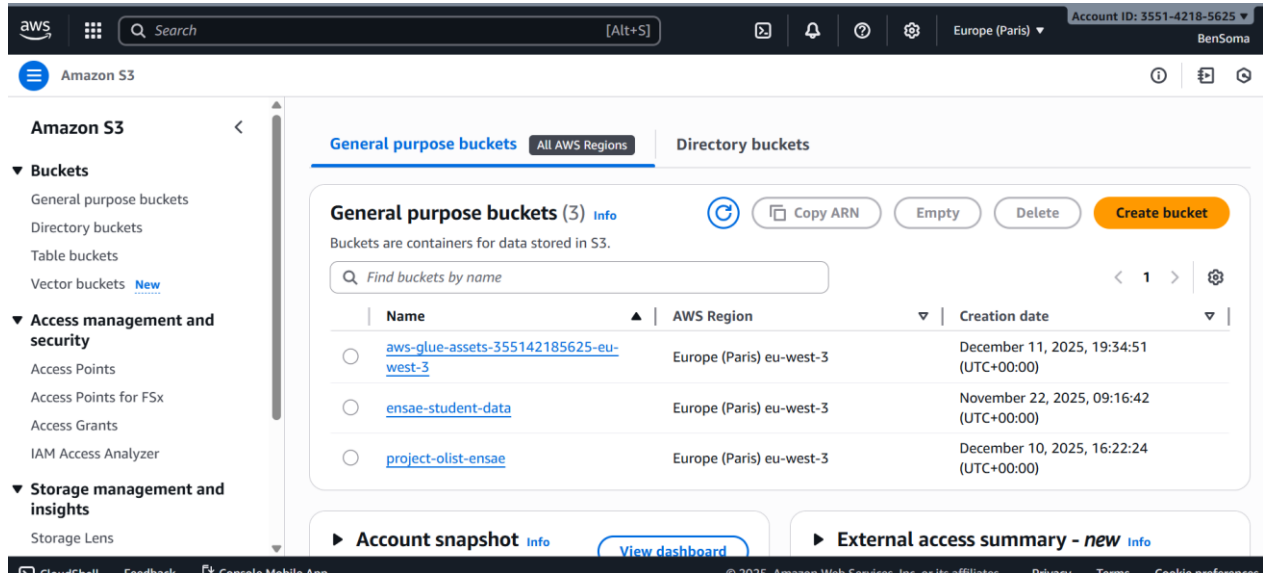
Mme Mously DIAW

Table des matières

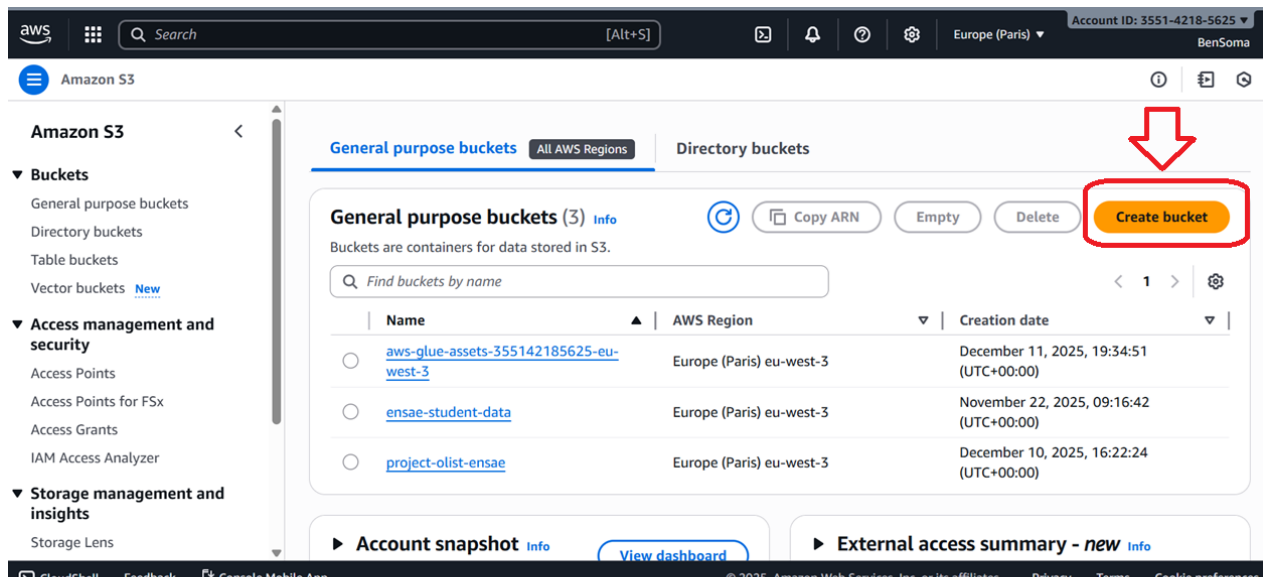
I.	Création d'un data lake avec amazon S3	2
I.1	Création d'un groupe et de plusieurs IAM ID (5 au total)	4
I.2	Création d'un bucket sur AWS Glue	5
I.3	Transformation des tables CSV en parquet	8
II.	PARTIE TEMPS REELLE DU PROJET	12
II.1	1. Le moteur de simulation et la gestion des stocks (DynamoDB).....	12
II.2	Génération de commandes via AWS Lambda	14
II.3	Ingestion, Buffering et Stockage.....	15
II.4	Intégration dans le Data Warehouse et Transformation (Redshift & dbt)	16
II.5	Visualisation (Power BI).....	17
II.6	Architecture Streaming (Temps Réel Pur) et Perspectives	19

I. Création d'un data lake avec amazon S3

On part sur AWS , ensuite on recherche le service S3 qui se présente comme ça :



On crée un compartiment en créant sur le bouton **create bucket**



Après la création du bucket, les tables de la base de données sont chargées dans celui-ci.

Le bucket, déjà créé, a été nommé **projet-olist-ensae**. En cliquant sur projet-olist-ensae, on accède à cette section où la base de données est visible, les différentes tables se trouvant dans le dossier bronze.

The screenshot shows the Amazon S3 console interface for the bucket 'project-olist-ensae'. The 'Objects' tab is active, displaying a list of objects. The 'bronze/' folder is highlighted with a red circle. The interface includes a search bar, a list of objects with columns for Name, Type, Last modified, Size, and Storage class, and various action buttons like 'Copy S3 URI', 'Copy URL', 'Download', 'Open', 'Delete', 'Actions', 'Create folder', and 'Upload'.

Name	Type	Last modified	Size	Storage class
bronze-parquet/	Folder	-	-	-
bronze/	Folder	-	-	-

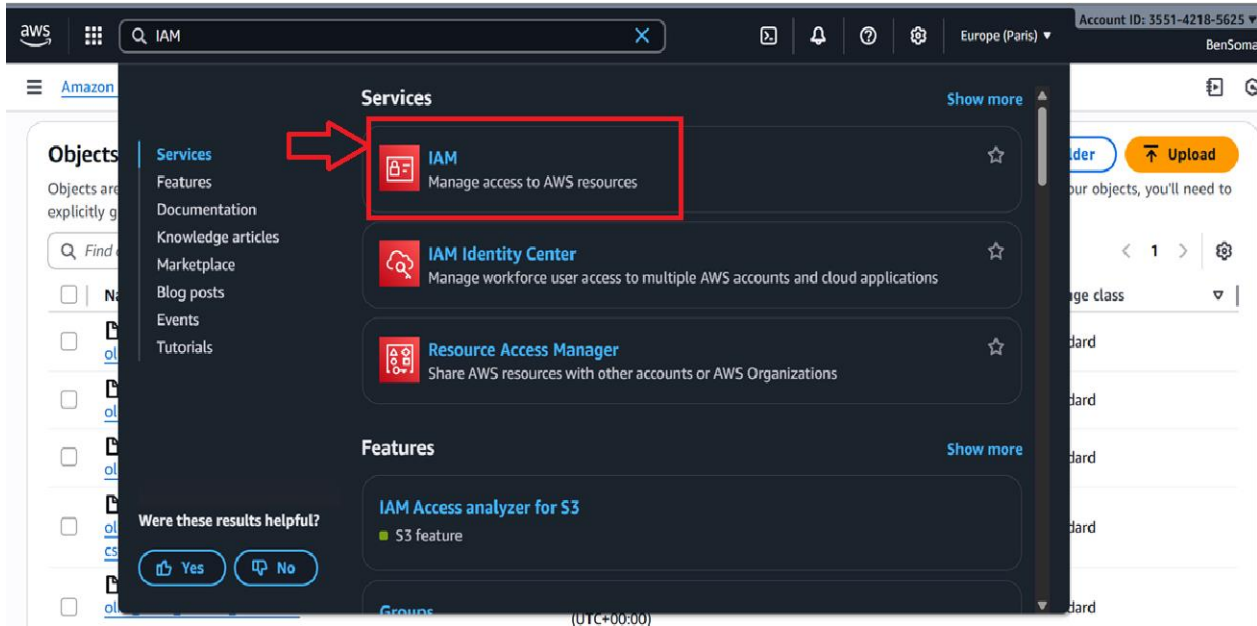
Voici comment les différentes tables se présentent :

The screenshot shows the Amazon S3 console interface for the bucket 'project-olist-ensae', specifically the 'bronze/' folder. The 'Objects' tab is active, displaying a list of objects. The objects are CSV files representing different datasets. The interface includes a search bar, a list of objects with columns for Name, Type, Last modified, Size, and Storage class, and various action buttons like 'Copy S3 URI', 'Copy URL', 'Download', 'Open', 'Delete', 'Actions', 'Create folder', and 'Upload'.

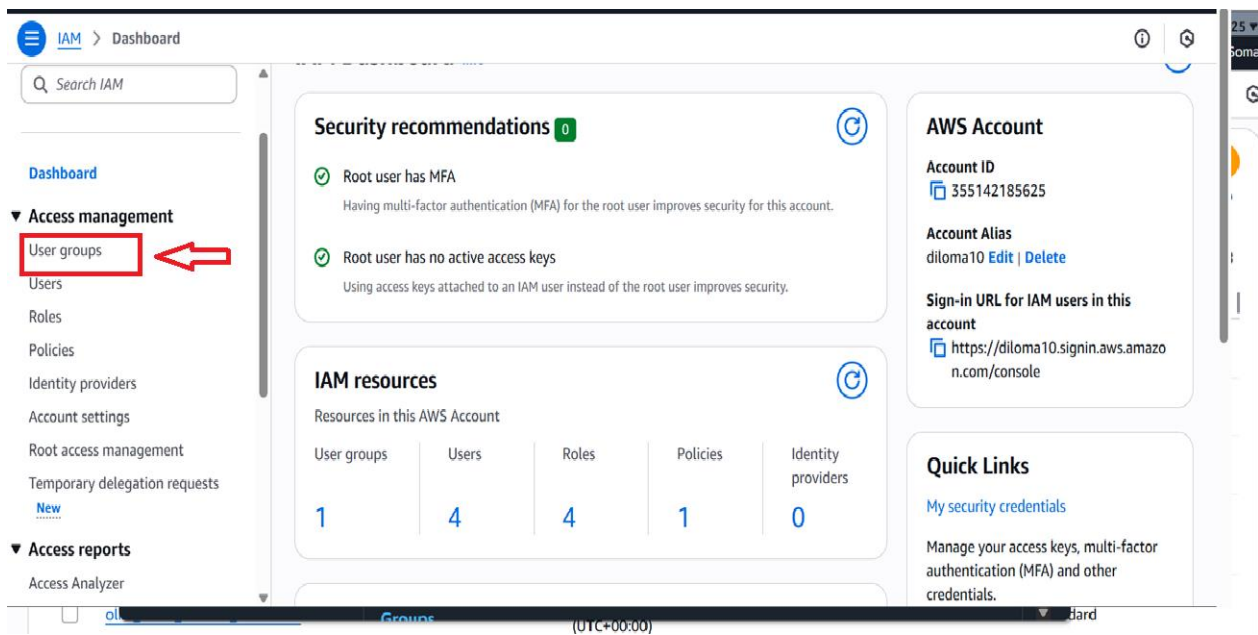
Name	Type	Last modified	Size	Storage class
olist_customers_dataset.csv	csv	December 11, 2025, 18:59:41 (UTC+00:00)	8.6 MB	Standard
olist_geolocation_dataset.csv	csv	December 11, 2025, 18:59:41 (UTC+00:00)	58.4 MB	Standard
olist_order_items_dataset.csv	csv	December 11, 2025, 18:59:41 (UTC+00:00)	14.7 MB	Standard
olist_order_payments_dataset.csv	csv	December 11, 2025, 18:59:41 (UTC+00:00)	5.5 MB	Standard
olist_order_reviews_dataset.csv	csv	December 11, 2025, 18:59:41 (UTC+00:00)	13.8 MB	Standard

I.1 Création d'un groupe et de plusieurs IAM ID (5 au total)

Partir au niveau de la barre de recherche de la console et rechercher **IAM**



On clique sur **IAM**

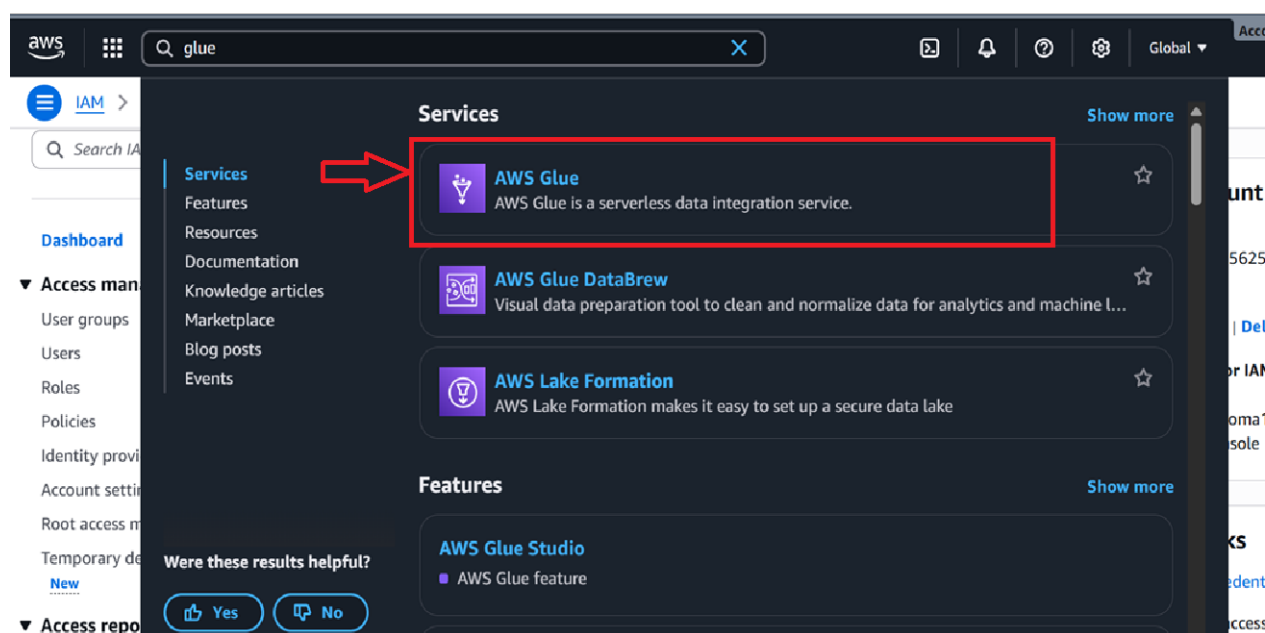


À présent, il convient de cliquer sur **User_groups** afin de créer les différents utilisateurs **IAM** disposant d'un accès complet aux ressources.

La création de ces utilisateurs permettra aux membres de l'équipe d'accéder à l'ensemble des travaux et d'y apporter des modifications.

1.2 Création d'un bucket sur AWS Glue

Rechercher **Glue** dans la barre de recherche



Il faut cliquer sur **AWS Glue**.

Après avoir cliqué sur ce service, on se rend dans l'onglet **Databases** situé à droite.

The screenshot displays the AWS Glue console interface. On the left, the 'Data Catalog' section is expanded, with 'Databases' highlighted. A red arrow points to this menu item. The main content area shows the 'Databases (1)' page. At the top right of this page, there is a red-bordered button labeled 'Add database', with a red arrow pointing to it. Below this button is a table listing existing databases. The table has columns for 'Name', 'Description', 'Location URI', and 'Created on (UTC)'. One database is listed: 'olist-database'.

<input type="checkbox"/>	Name	Description	Location URI	Created on (UTC)
<input type="checkbox"/>	olist-database	-	-	December 11, 2025 at 19:07:30

Il convient ensuite de cliquer sur « **Add database** » afin de créer la base de données sur AWS Glue, ce qui permettra de charger les tables depuis S3.

On crée un CRAWLER en cliquant sur **CRAWLER** dans la barre latérale

AWS Glue > Crawlers

Crawlers
A crawler connects to a data store, progresses through a prioritized list of classifiers to determine the schema for your data, and then creates metadata tables in your data catalog.

Crawlers (1) Info
Last updated (UTC) December 13, 2025 at 01:32:42 [Refresh] [Action] [Run] **Create crawler**

View and manage all available crawlers.

Filter crawlers

<input type="checkbox"/>	Name	State	Schedule	Last run	Last run ...	Log	Table cha...
<input type="checkbox"/>	olist	Ready		Succeeded	December 1...	View log	9 created

Après la création de notre crawler, cliquer sur « **run a crawler** » et après ça nous verrons les différentes tables qui ont été importées sur **GLUE** .

La base a été créée et se nomme olist-database on clique sur olist-database

AWS Glue

Databases (1)
Last updated (UTC) December 13, 2025 at 01:27:42 [Refresh] [Edit] [Delete] **Add database**

A database is a set of associated table definitions, organized into a logical group.

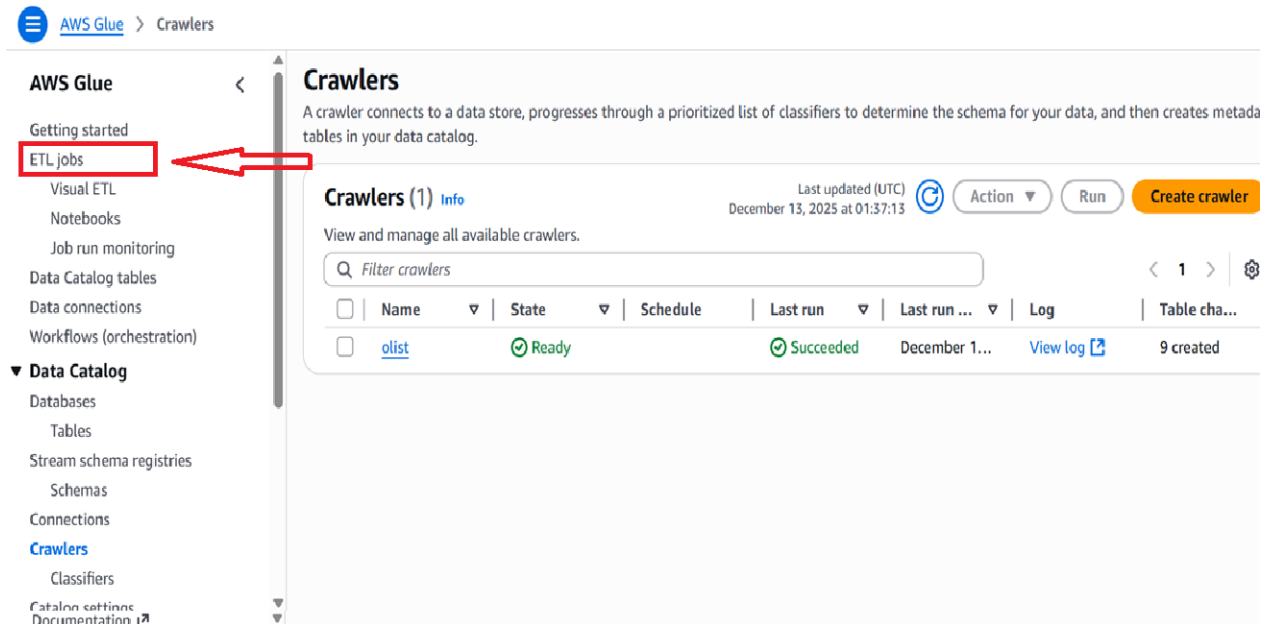
Filter databases

<input type="checkbox"/>	Name	Description	Location URI	Created on (UTC)
<input type="checkbox"/>	olist-database	-	-	December 11, 2025 at 19:07:30

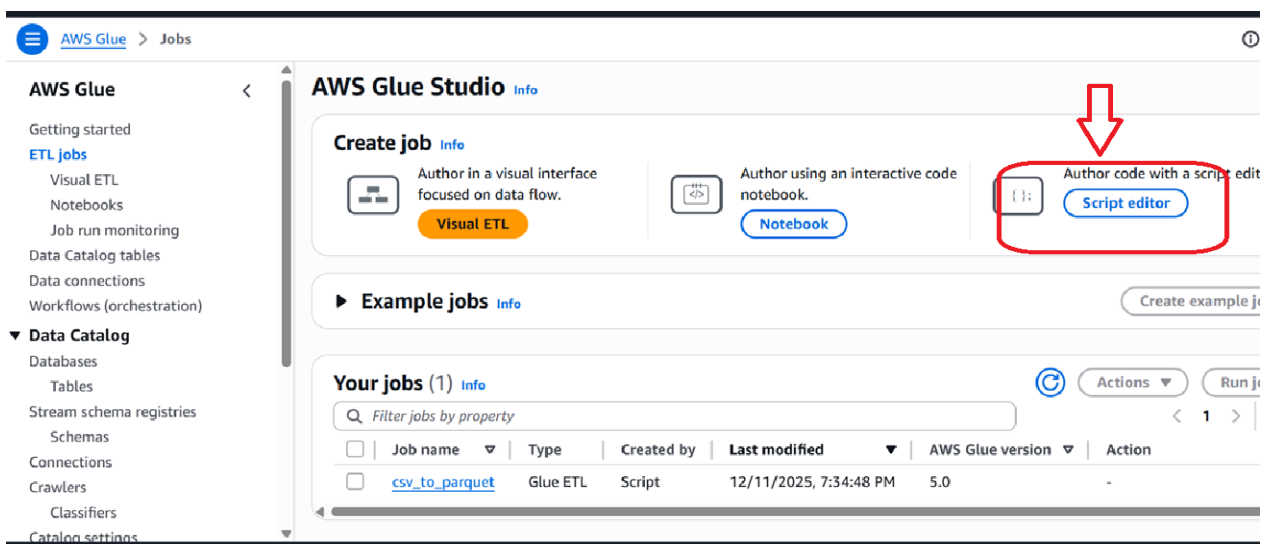
I.3 Transformation des tables CSV en parquet

Pourquoi parquet ?

- Pour pouvoir gérer de grandes bases de données
- Pour faire ceci, il faut aller au niveau de **GLUE** dans la barre latérale a gauche nous cliquons sur « **ETL JOB** »



Dans **ETL Jobs** on clique sur script Editor



On choisit l'option spark pour le code

Un code a été utilisé pour transformer les fichiers CSV en parquet : il se trouve en annexes.

Lorsque l'on se rend dans S3, il est possible de constater qu'un nouveau dossier a été créé et qu'il contient les fichiers parquet. Pourquoi des tables sous format parquet ?

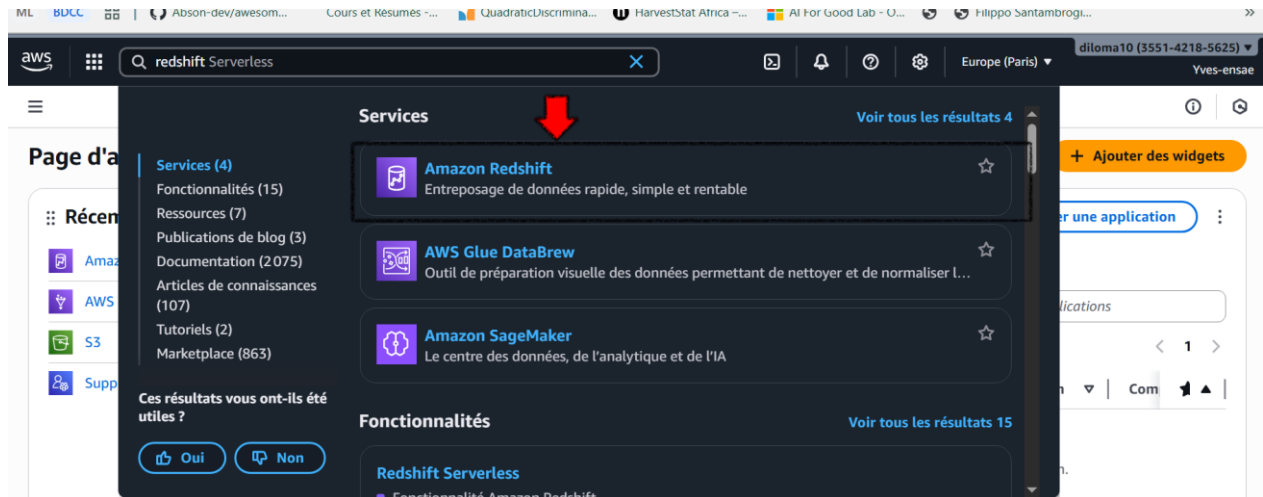
Parce que le format Parquet est plus performant que le CSV : il est orienté colonnes, permet une **compression efficace**, réduit l'espace de stockage sur S3, **accélère les lectures avec Spark**, conserve le **schéma des données** et est mieux adapté aux **analyses Big Data**.

Tableau 1: Résumé des services utilisés et leur rôle jusqu'à présent

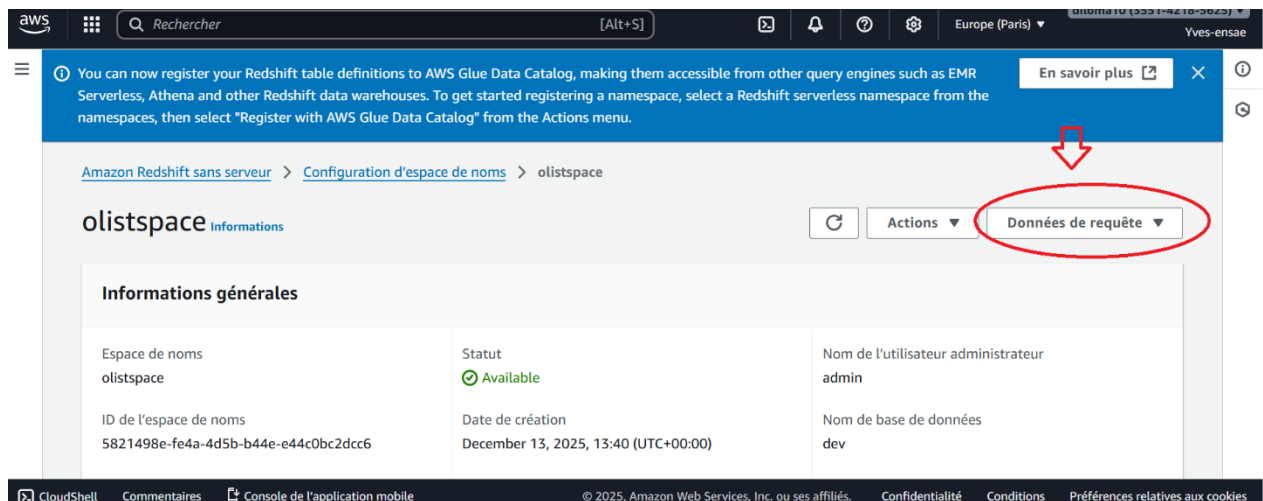
Service AWS	Utilité dans le projet
Amazon S3	Stockage scalable et sécurisé des données brutes et transformées.
IAM	Gestion des accès et des permissions pour les utilisateurs et services.
AWS Glue	Catalogage des données, crawling automatique, et transformation ETL serverless.
Glue Data Catalog	Catalogue centralisé des métadonnées pour interroger les données avec Athena/Redshift.
Glue Crawler	Découverte automatique des schémas et mise à jour du catalogue.
Glue ETL Job	Exécution de scripts PySpark pour transformer les données (CSV → Parquet).
Parquet	Format de stockage optimisé pour l'analytique Big Data.

Source : calculs des auteurs

Après ces différentes étapes, l'accès à Amazon Redshift est effectué.



Après la création du groupe il faut cliquer sur le nom dans l'espace de groupes



Ensuite, il faut se rendre dans **Query Editor** afin d'ajouter des scripts, d'importer des tables et d'effectuer les transformations nécessaires.

Dans cette étape, de nombreuses transformations sont réalisées sur la base de données.

Avant toute transformation, il est nécessaire d'importer les données de S3 vers Redshift. Les codes correspondant à cette tâche figurent en annexe.

Une fois l'importation terminée, les données sont transformées, puis stockées dans un dossier **silver**. Les codes utilisés pour ces transformations sont également disponibles en annexe.

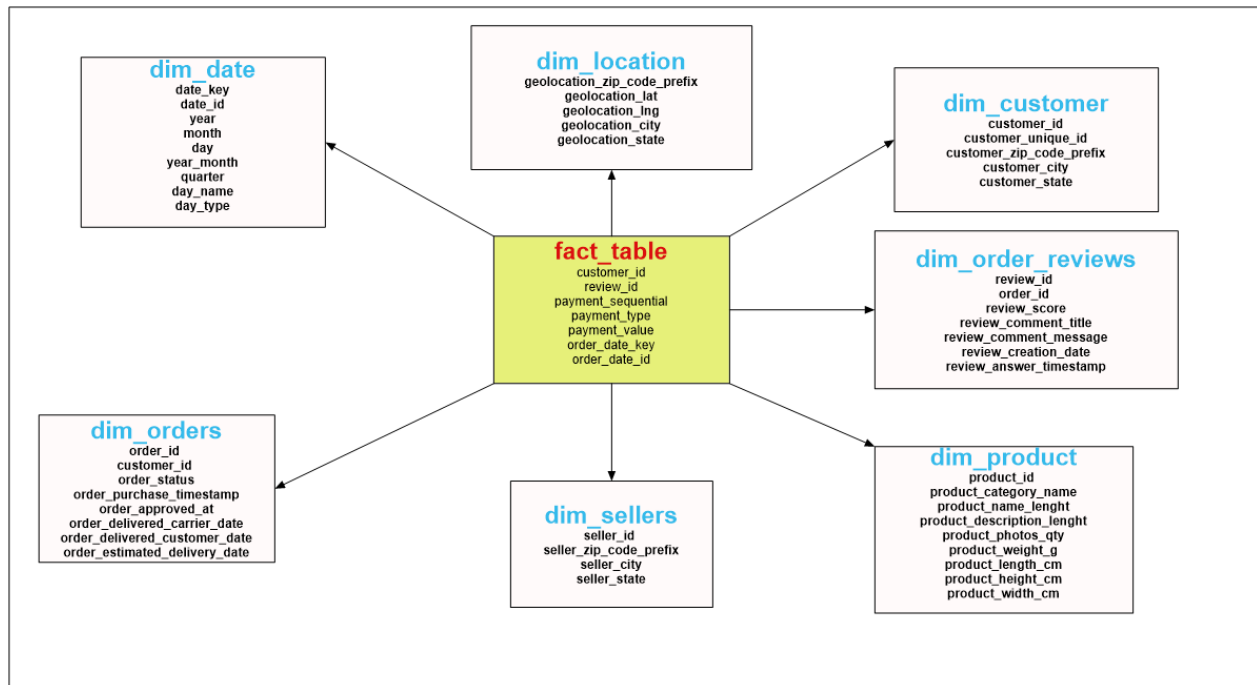
Le schéma du data warehouse est ensuite défini.

Après cela, un dossier **dw** est créé pour stocker les tables de dimensions et de faits, selon une modélisation en **étoile**.

Pourquoi le modèle étoile ?

Nous avons utilisé le **modèle en étoile** car il permet une **modélisation simple et lisible**, optimise les **performances des requêtes analytiques**, facilite les **jointures entre tables de faits et dimensions**, et est particulièrement adapté aux **outils de BI** comme Power BI pour l'analyse et la visualisation des données.

Figure 1: Modèle en étoile



Source : auteurs

II. PARTIE TEMPS REELLE DU PROJET

Dans le cadre de la construction de notre Data Warehouse basé sur le dataset Olist, nous avons mis en place une architecture capable de gérer les données en temps réel. Il est important de préciser que, dans notre contexte, nous distinguons deux approches du "temps réel". La première est une approche **"Batch rapide"**, où les données arrivent par intervalles de quelques minutes, et la seconde est une version purement streaming, de l'ordre de la seconde ou de la milliseconde. Cette section de la documentation se concentre spécifiquement sur l'implémentation de la partie Batch, simulant un flux de commandes continu qui alimente notre entrepôt de données.

II.1 1. Le moteur de simulation et la gestion des stocks (DynamoDB)

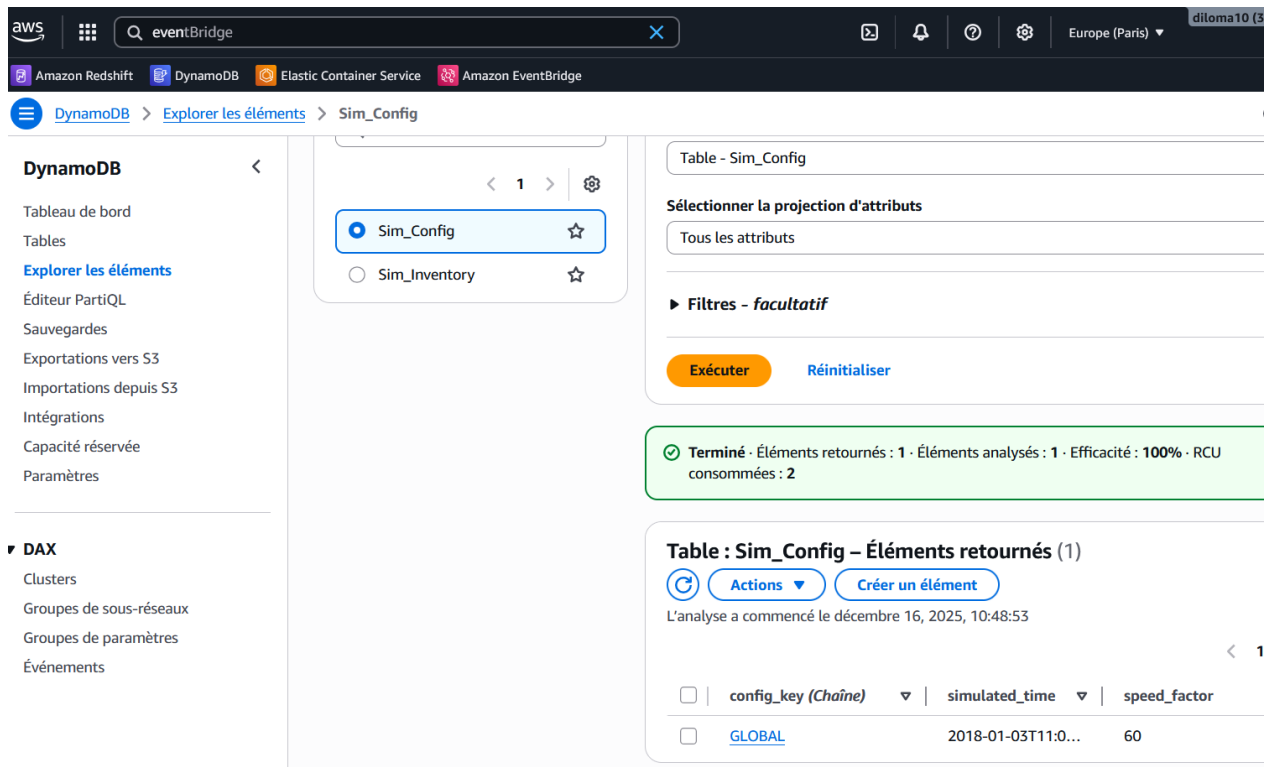
Pour simuler un environnement de production réaliste, nous ne pouvions pas simplement injecter des données statiques. Nous avons donc créé un moteur de simulation basé sur Amazon

DynamoDB. La première composante est la table `Sim_Config`. Cette table agit comme le chef d'orchestre temporel de notre simulation. Elle contient une clé globale qui stocke l'heure virtuelle (`simulated_time`) et un facteur d'accélération (`speed_factor`). Cela nous permet de faire avancer le temps beaucoup plus vite que la réalité (par exemple, une minute réelle équivaut à une heure simulée), testant ainsi la robustesse du pipeline sur de longues périodes virtuelles en peu de temps.

The screenshot shows the AWS Management Console interface for the 'Sim_Config' table in DynamoDB. The left sidebar contains navigation options like 'Tableau de bord', 'Tables', 'Explorer les éléments', and 'DAX'. The main content area displays the table 'Sim_Config' with a selection of 'Sim_Config' and 'Sim_Inventory'. Below this, there's a section for 'Sélectionner la projection d'attributs' with 'Tous les attributs' selected. A status bar indicates 'Terminé · Éléments retournés : 1 · Éléments analysés : 1 · Efficacité : 100% · RCU consommées : 2'. At the bottom, a table titled 'Table : Sim_Config – Éléments retournés (1)' shows a single row with columns 'config_key (Chaîne)', 'simulated_time', and 'speed_factor'. The row value is 'GLOBAL', '2018-01-03T11:0...', and '60' respectively.

config_key (Chaîne)	simulated_time	speed_factor
GLOBAL	2018-01-03T11:0...	60

En parallèle, nous avons mis en place la table `Sim_Inventory` pour gérer les stocks de produits. Plutôt que de reprendre l'intégralité du dataset Olist, nous avons sélectionné un échantillon pertinent pour démontrer la faisabilité du concept. Cette table contient les identifiants produits (`product_id`), ainsi que leur niveau de stock actuel (`stock_level`), la catégorie et le prix. Nous avons utilisé un script Python, exécuté via l'AWS CLI, pour peupler initialement cette table. Ce script prépare le terrain pour que notre générateur de commandes puisse interagir avec des données concrètes.

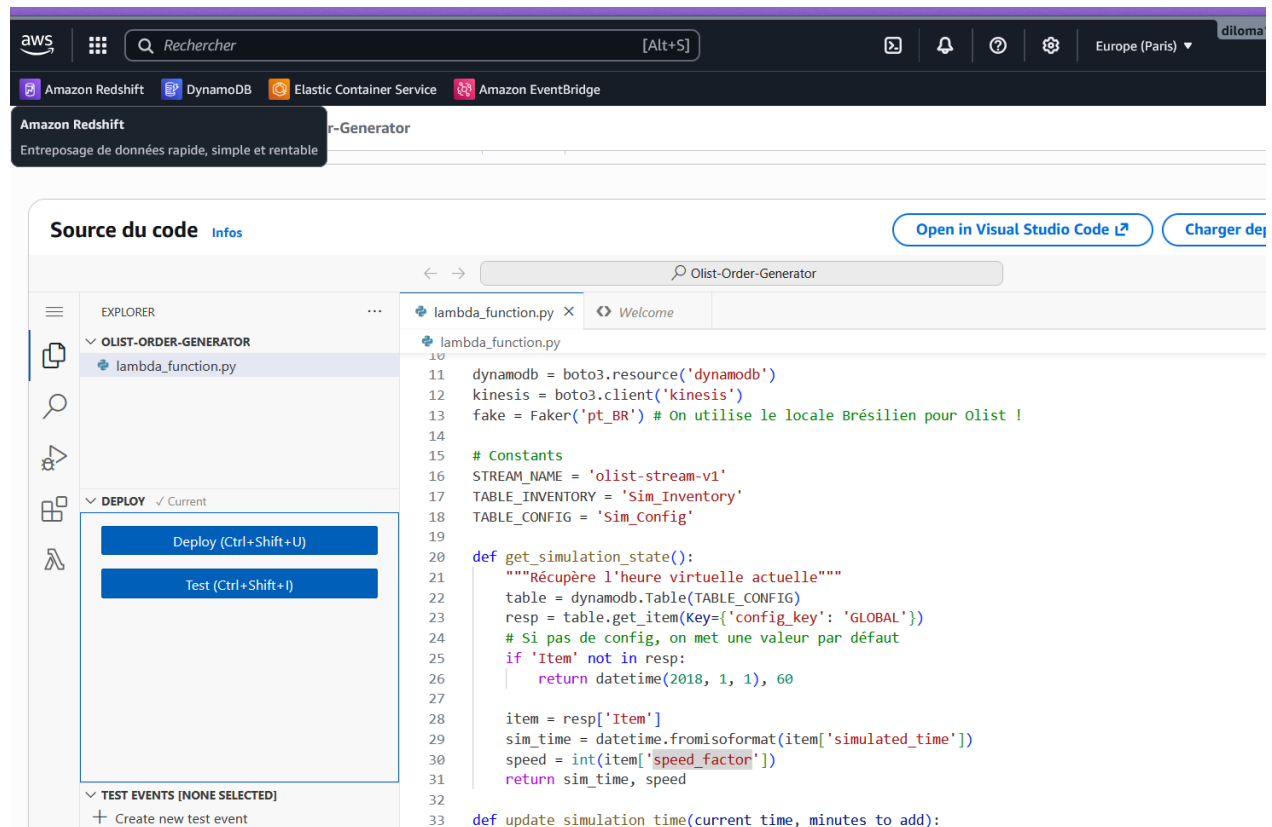


II.2 Génération de commandes via AWS Lambda

Le cœur de notre ingestion de données réside dans une fonction AWS Lambda nommée "Order Generator". Cette fonction est déclenchée automatiquement toutes les minutes grâce à un planificateur Amazon EventBridge. La logique du code est conçue pour être performante et réaliste. Au démarrage, la fonction consulte la table `Sim_Config` pour savoir "quelle heure il est" dans la simulation. Pour éviter de surcharger la base de données et réduire les coûts, nous avons implémenté un système de mise en cache mémoire qui charge une liste limitée d'IDs de produits une seule fois par cycle, évitant ainsi de scanner toute la table d'inventaire à chaque commande.

La fonction génère ensuite un nombre aléatoire de commandes (entre 3 et 20) en utilisant la librairie `Faker` localisée pour le Brésil, afin de rester fidèle au contexte Olist. Point crucial de notre architecture : la gestion de la cohérence des données. Avant de valider une commande, le script effectue une transaction atomique sur `DynamoDB` pour décrémenter le stock. Si le stock est insuffisant, la commande est annulée, ce qui empêche de vendre des produits inexistants. Une fois la commande validée, elle est convertie en JSON et poussée vers notre flux de données. l'événement peut être activé ou désactivé, cela n'impacte pas le temps simulé et nous cela nous économise de l'argent sans qu'il y ait des trous. Firehose stocke le format parquet en format

année/mois/heure/jour/munites en se referant au temps de la vrai vie , pour changer cela il vas falloir appeler une nouvelle fonction lambda et c'est couteux et comme nous ne nous servons pas directement de cette information (par contre si on voulait stocker dans ce style les data statique et les similé ensemble alors la oui) donc nous avons laissé intacte.

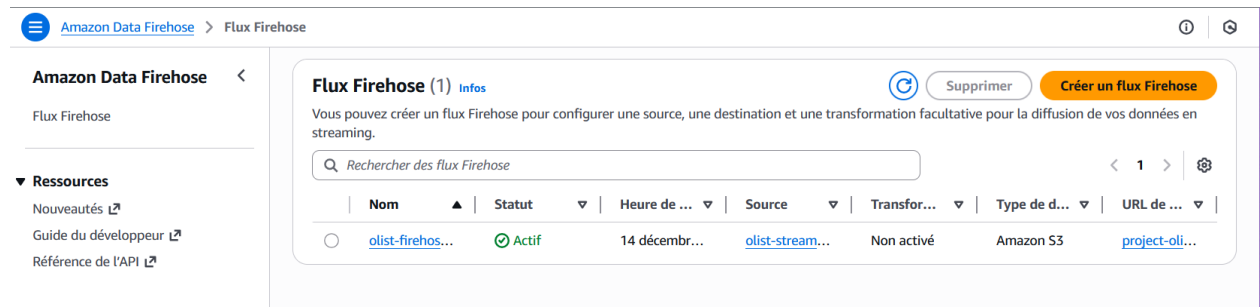


II.3 Ingestion, Buffering et Stockage

Une fois les commandes générées, elles entrent dans le pipeline d'ingestion via Amazon Kinesis Data Stream. Ce flux continu est ensuite consommé par Kinesis Firehose. Nous avons configuré Firehose avec des paramètres de tampon spécifiques : il accumule les données soit pendant 3 minutes, soit jusqu'à atteindre 5 Mo. Ce mécanisme de "micro-batch" est essentiel pour éviter de créer une multitude de trop petits fichiers, ce qui nuirait aux performances ultérieures.

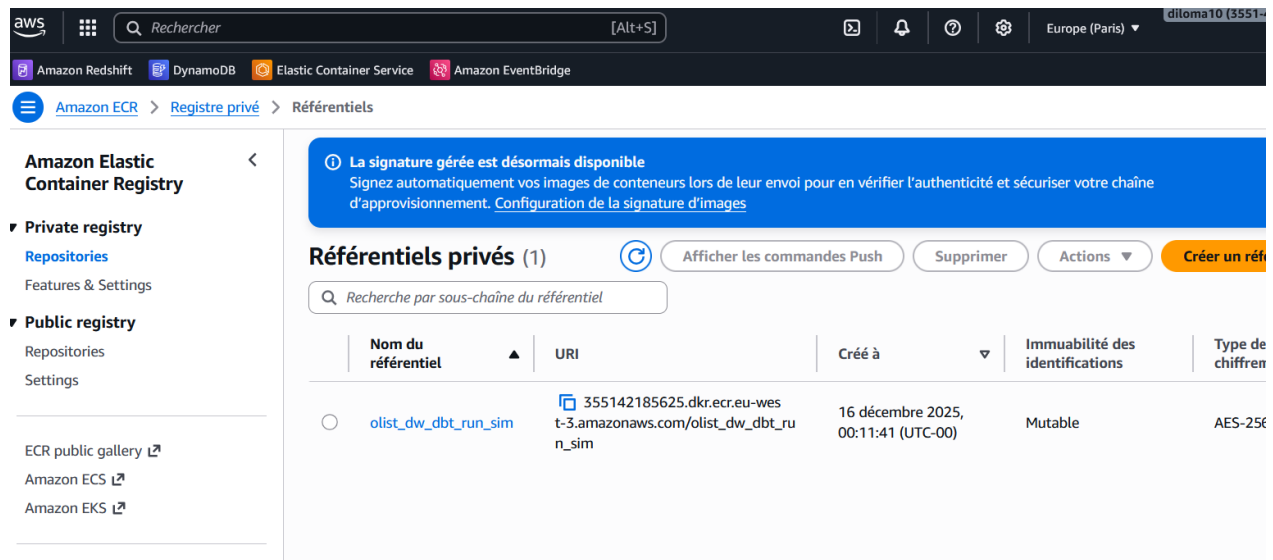
Durant ce transit dans Firehose, une transformation majeure a lieu. Grâce à son intégration avec AWS Glue Data Catalog, Firehose convertit automatiquement les données JSON brutes au format Parquet avant de les écrire moyennant la création d'un data catalogue conforme au format json

attendu. Le format Parquet, étant en colonnes et compressé, est beaucoup plus performant pour l'analytique. Les fichiers résultants sont déposés dans notre bucket S3, dans un dossier désigné comme notre couche "Bronze" (sim-bronze).



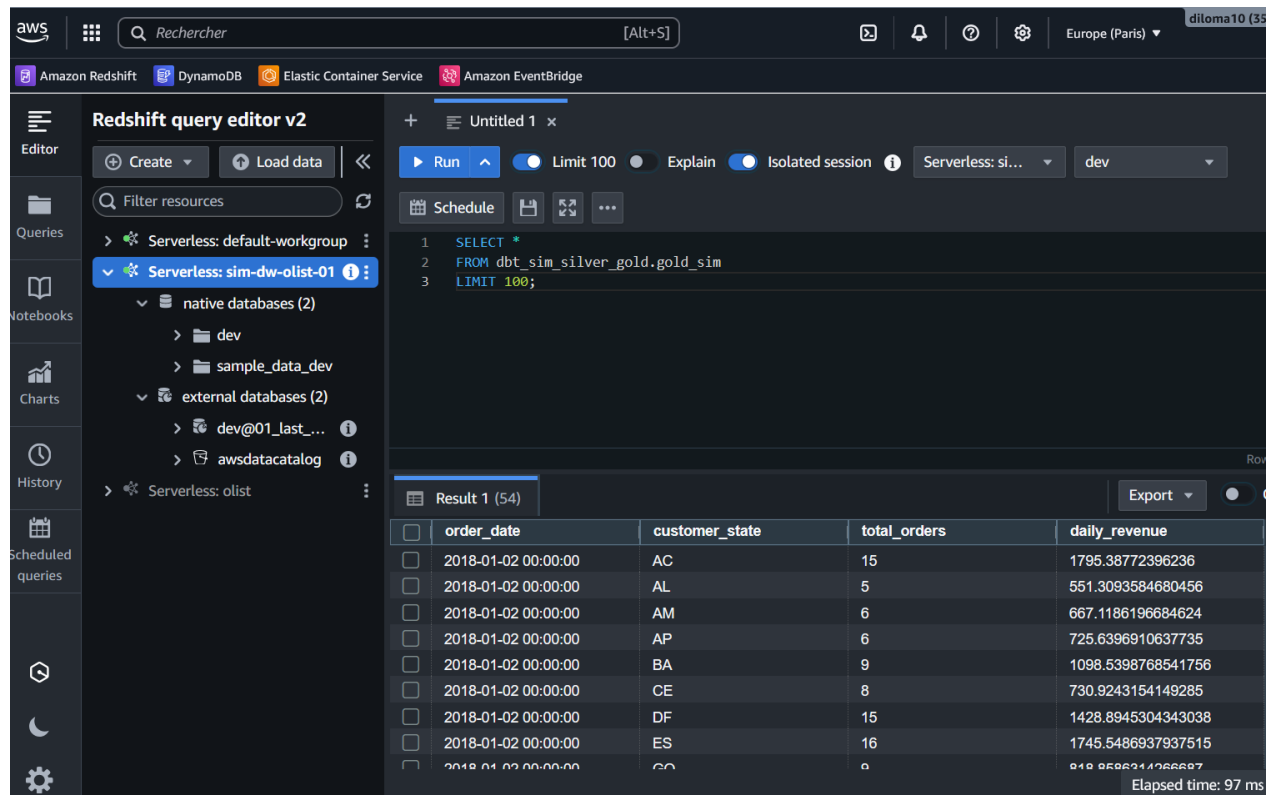
II.4 Intégration dans le Data Warehouse et Transformation (Redshift & dbt)

Pour rendre ces données accessibles à l'analyse, nous utilisons Amazon Redshift Serverless couplé à Redshift Spectrum. Grâce au schéma externe défini dans AWS Glue, Redshift peut lire directement les fichiers Parquet stockés dans S3 sans avoir besoin de les charger physiquement dans son propre stockage interne. Cela nous offre une flexibilité énorme et sépare les coûts de stockage des coûts de calcul.



La transformation de la donnée brute (Bronze) vers des modèles nettoyés (Silver) et agrégés (Gold) est assurée par dbt (data build tool). Pour automatiser cette étape, nous avons conteneurisé notre projet dbt et stocké l'image dans Amazon ECR. L'orchestration est gérée par AWS Step Functions. Le processus est entièrement événementiel : dès qu'EventBridge détecte qu'un nouveau fichier a

été déposé dans le dossier S3 Bronze, il déclenche la Step Function. Celle-ci lance alors le conteneur ECR qui exécute les transformations dbt. Pour sécuriser l'accès à la base de données, le conteneur récupère les identifiants de connexion de manière sécurisée via AWS Secrets Manager.



The screenshot displays the Amazon Redshift Query Editor v2 interface. The left sidebar shows the 'Editor' tab with a 'Filter resources' search bar and a tree view of databases. The main area shows a SQL query in a text editor, and the bottom section displays the results of the query in a table format.

SQL Query:

```
1 SELECT *
2 FROM dbt_sim_silver_gold.gold_sim
3 LIMIT 100;
```

Query Results (Result 1 (54)):

order_date	customer_state	total_orders	daily_revenue
2018-01-02 00:00:00	AC	15	1795.38772396236
2018-01-02 00:00:00	AL	5	551.3093584680456
2018-01-02 00:00:00	AM	6	667.1186196684624
2018-01-02 00:00:00	AP	6	725.6396910637735
2018-01-02 00:00:00	BA	9	1098.5398768541756
2018-01-02 00:00:00	CE	8	730.9243154149285
2018-01-02 00:00:00	DF	15	1428.8945304343038
2018-01-02 00:00:00	ES	16	1745.5486937937515
2018-01-02 00:00:00	GA	6	819.9586241266697

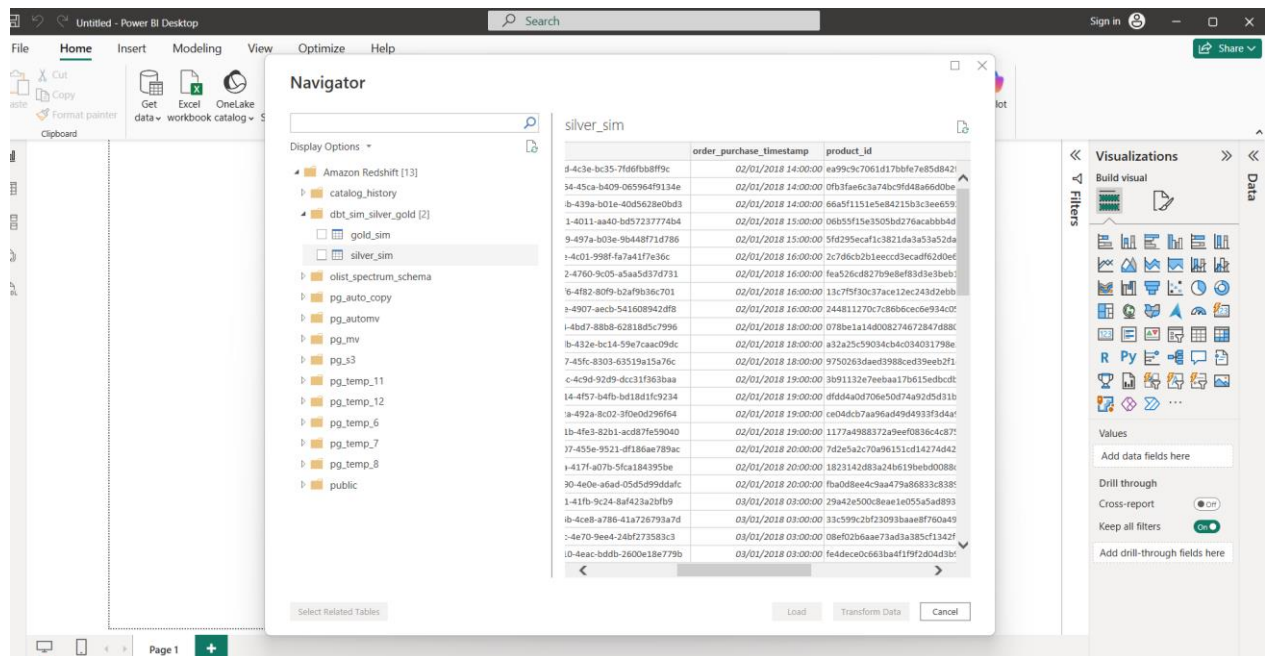
Elapsed time: 97 ms

II.5 Visualisation (Power BI)

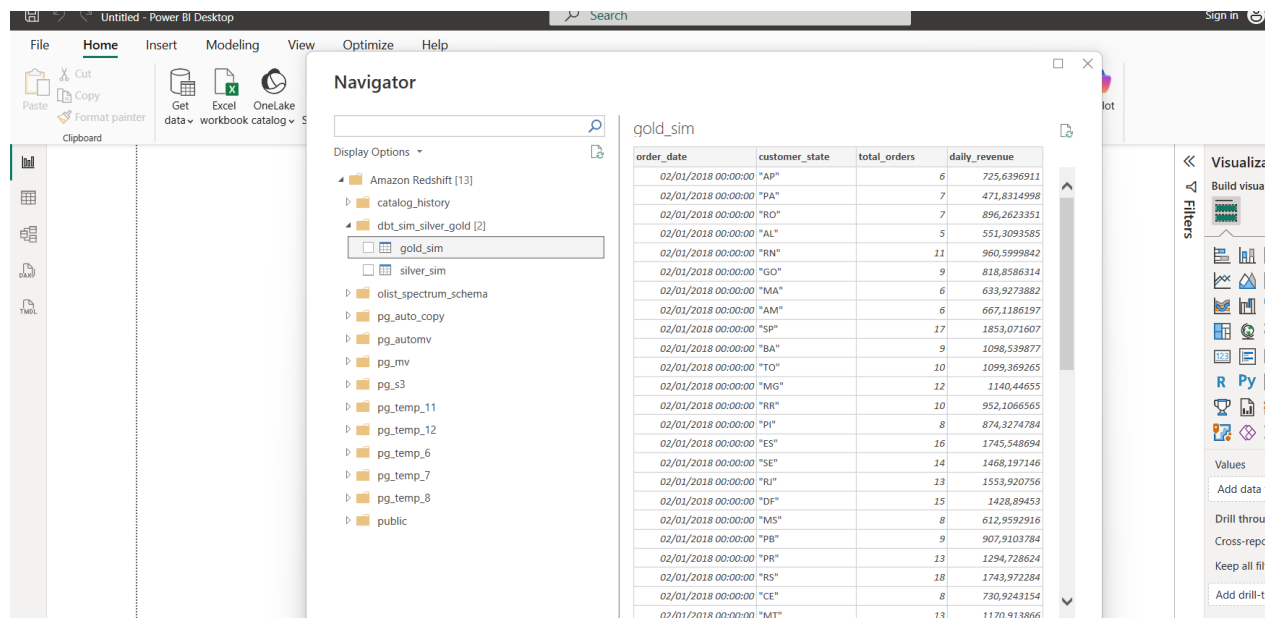
La finalité de ce pipeline est la visualisation des données. Nous avons connecté Power BI directement à notre cluster Redshift. Pour permettre cette connexion, nous avons dû configurer un utilisateur dédié sur Redshift et ajuster les paramètres du VPC (Virtual Private Cloud) et des groupes de sécurité pour autoriser le trafic entrant depuis Power BI. Cela permet aux parties prenantes de suivre l'évolution des commandes et des stocks avec une latence de seulement quelques minutes par rapport à la génération des données.

Les transformation pour avoir le silver et le gold data sont dans le dossier models/ sur <https://github.com/Ben10-som/Data-warehouse-for-sales-analysis->

Dans la prochaine version nous passerons par Cloudformation ou terraform pour refaire le projet et y inclure la partie temps réelle.



order_purchase_timestamp	product_id
02/01/2018 14:00:00	ea99c9c7061d17b0fe7e85d842
02/01/2018 14:00:00	0fb3faec3a74bc9f948a66d0be
02/01/2018 14:00:00	66a5f1151e5e84215b3c3ee559
02/01/2018 15:00:00	06b5f15e3505bd276acabbdbd
02/01/2018 15:00:00	5fd295ecaf1c3821da3a5a32da
02/01/2018 16:00:00	2c7d6cb2b1eecc3ecadff62d0ef
02/01/2018 16:00:00	13c7f5f30c37ace12ec243d2ebb
02/01/2018 16:00:00	1ea526cd827b9e8ef83d3e3beb
02/01/2018 16:00:00	244811270c7c86b6ccce9934c02
02/01/2018 18:00:00	078be1a14d008274672847488k
02/01/2018 18:00:00	432e-bc14-59c7caac09dc
02/01/2018 18:00:00	9750263daed3988ced39eeb27f1
02/01/2018 19:00:00	3b91132e7eeba17b615edbcdff
02/01/2018 19:00:00	dfdd4a0d70e50d74a92d3531b
02/01/2018 19:00:00	ce04dc17a96ad49d493f3d4ar
02/01/2018 19:00:00	1177a4988372a9ee0836c4875
02/01/2018 20:00:00	7d2e5a2c70a96151cd14274d42
02/01/2018 20:00:00	1823142d83a24b619beb0088k
02/01/2018 20:00:00	fba0d8e4c9aa479a86833c838f
03/01/2018 03:00:00	29a42e500c8ea1e055a5ad893
03/01/2018 03:00:00	33c599c2bf23093baae87f60a4f
03/01/2018 03:00:00	08ef02b6ae73ad3a385cf1342f
03/01/2018 03:00:00	fe4dece0c683ba4f1f92804d3bf



order_date	customer_state	total_orders	daily_revenue
02/01/2018 00:00:00	"AP"	6	725,6396911
02/01/2018 00:00:00	"PA"	7	471,8314998
02/01/2018 00:00:00	"RO"	7	896,2623351
02/01/2018 00:00:00	"AL"	5	551,3093585
02/01/2018 00:00:00	"RN"	11	960,5999842
02/01/2018 00:00:00	"GO"	9	818,8586314
02/01/2018 00:00:00	"MA"	6	633,9273882
02/01/2018 00:00:00	"AM"	6	667,1186197
02/01/2018 00:00:00	"SP"	17	1853,071607
02/01/2018 00:00:00	"BA"	9	1098,539877
02/01/2018 00:00:00	"TO"	10	1099,369265
02/01/2018 00:00:00	"MG"	12	1140,44655
02/01/2018 00:00:00	"RR"	10	952,1066565
02/01/2018 00:00:00	"PI"	8	874,3274784
02/01/2018 00:00:00	"ES"	16	1745,548694
02/01/2018 00:00:00	"SE"	14	1468,197146
02/01/2018 00:00:00	"RJ"	13	1553,920756
02/01/2018 00:00:00	"DF"	15	1428,89453
02/01/2018 00:00:00	"MS"	8	612,5592916
02/01/2018 00:00:00	"PB"	9	907,9103784
02/01/2018 00:00:00	"PR"	13	1294,728624
02/01/2018 00:00:00	"RS"	18	1743,972284
02/01/2018 00:00:00	"CE"	8	730,9243154
02/01/2018 00:00:00	"MT"	13	1170,913866

II.6 Architecture Streaming (Temps Réel Pur) et Perspectives

Comme évoqué dans l'introduction, notre analyse distinguait deux niveaux de temporalité : le micro-batch (actuellement en place) et le streaming pur (latence de l'ordre de la seconde). Pour cette itération du projet, nous avons fait le choix conscient de ne pas implémenter la branche de streaming pur. Cette décision a été dictée par des contraintes pragmatiques de temps de développement et de budget (optimisation des coûts des services Cloud), l'architecture actuelle suffisant à démontrer la robustesse de l'ingestion de données.

Toutefois, l'architecture cible pour ce besoin spécifique est déjà théorisée. L'objectif serait de réduire drastiquement la latence en éliminant l'étape de stockage intermédiaire sur Amazon S3 ("Sim-Bronze"). Dans cette configuration, nous utiliserions l'intégration native Streaming Ingestion d'Amazon Redshift. Concrètement, le flux Kinesis Firehose ne déposerait plus de fichiers Parquet, mais injecterait les données directement dans les tables Redshift en temps réel. Cela permettrait de contourner les délais liés à la bufferisation des fichiers et aux triggers événementiels, offrant ainsi une vision quasi instantanée des transactions pour des cas d'usage critiques.

Annexe: Code pour transforme des bases csv en format parquet

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job

# Initialisation du job Glue

args = getResolvedOptions(sys.argv, ["JOB_NAME"])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args["JOB_NAME"], args)

#Emplacements S3
input_path = "s3://your-bucket-name/bronze/"      # tes CSV Olist
output_path = "s3://your-bucket-name/bronze-parquet/" # dossier Parquet à créer

# Lecture des CSV

df_csv = (
    spark.read
        .option("header", "true")
        .option("inferSchema", "true") # (optionnel)
        .option("sep", ",")
        .csv(input_path)
)

print("Nombre de lignes chargées :", df_csv.count())
print("Schéma détecté :")
df_csv.printSchema()

# Écriture en Parquet
(
    df_csv.write
        .mode("overwrite")      # écrase si le dossier existe
        .option("compression", "snappy")
        .parquet(output_path)
)

print("Transformation CSV → Parquet terminée.")

job.commit()
```