5DATA

Projet final

Conception d'une plateforme d'ingénierie des données

Edgar Weber
Mathis Roger
Stéphan Grould
Supinfo International University
Campus de Rennes
M.Sc 2

Contexte

Engagés par Supinfo, notre mission consiste en la conception et au déploiement d'un POC afin de fournir une plateforme de type "Data Science", et orientée Big Data dans le but d'analyser les données produites au travers des activités de l'école.

Analyse Fonctionnelle

Introduction

Comme évoqué précédemment, Supinfo souhaite se doter d'une plateforme Big Data afin de mener des analyses sur les données produites au travers de ses activités.

La solution doit donc permettre :

- la collecte et l'intégration de données massives en provenance de différents systèmes hétérogènes
- aux data scientists et analystes de requêter et analyser les données
- aux data engineers de concevoir et exécuter des traitements big data, ainsi que de les planifier, de monitorer la solutions
- aux équipes de collaborer de manières agiles sur des projets data

Les données comportant des données à caractère personnel, la solution doit répondre aux obligations relatives à la RGPD.

Analyse des données

Parmis la connaissance que souhaite tirer le métier de ces données, on peut citer :

- Qui sont les étudiants les plus performants, par région, par institution d'origine
- Qui sont les étudiants qui arrêtent leurs études et pourquoi
- Pourquoi y a-t-il plus d'étudiants dans certaines régions que dans d'autres
- Quelles régions ont plus de contrat pro et pourquoi
- Comment revitaliser les campus
- Quelles est l'impact des portes ouvertes, salons sur les recrutements
- Quelle est la durée moyenne des recrutements des diplômés

- Quelles entreprises recrutent le plus d'étudiants
- Quelles écoles concurrentes recrutent nos étudiants
- Quelles sont les prévisions de croissance de l'école

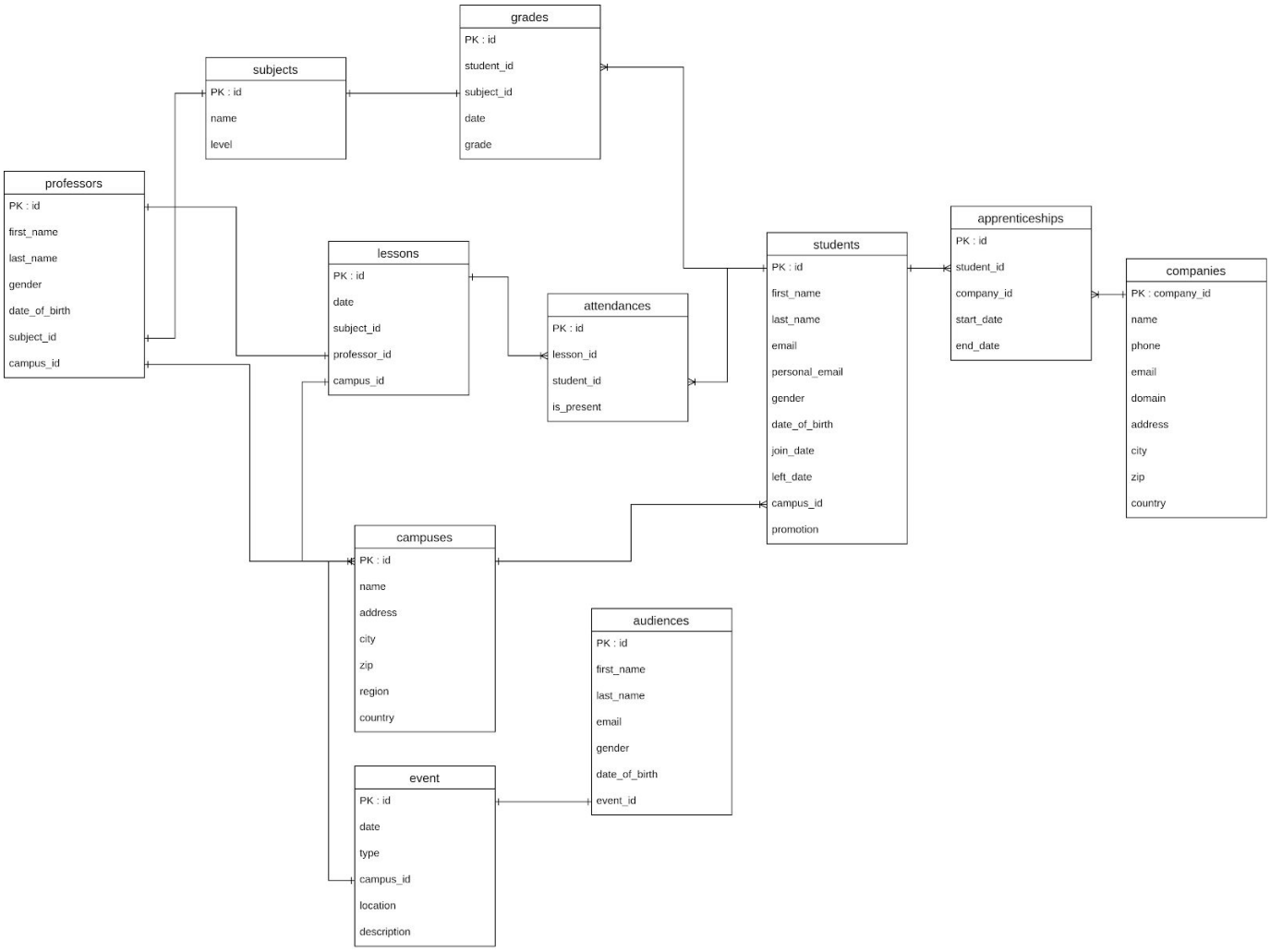
Dataset

L'expression de besoin ne fournissant pas de dataset, nous en avons construit un en extrapolant sur la base des analyses qui seront effectuées, et proposons les entités suivantes :

- les campus : ***campuses***
- les étudiants : ***students***
- les professeurs : ***professors***
- les matières : ***subjects***
- les cours (sujet, date, campus..) : ***lessons***
- les présences des étudiants en cours : ***attendances***
- les notes des étudiants dans chaques matières : ***grades***
- les contrats pro : ***internships***
- les entreprises, dans lesquelles sont effectués les contrats pro : ***companies***
- les évènements (portes ouvertes, salons) : ***events***
- les contacts de potentiels futurs étudiants, collectés lors des évènements : ***audiences***

Nous considérons que ces données sont stockées dans l'ERP de Supinfo, donc dans une base de données relationnelle.

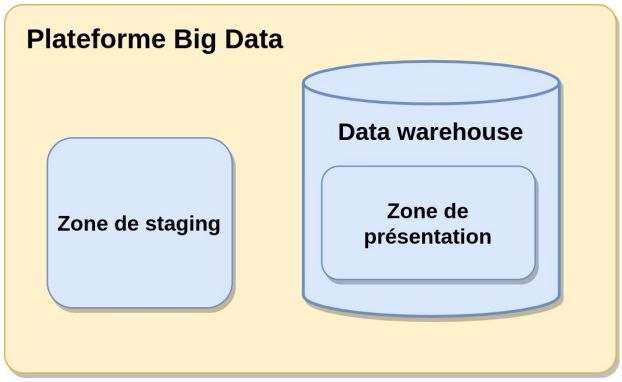
Ci dessous le schéma de l'ERP :


grades
PK : id
student_id
subject_id
date
grade
subjects
PK : id
name
level
professors
PK : id
first_name
last_name
gender
date_of_birth
subject_id
campus_id
lessons
PK : id
date
subject_id
professor_id
campus_id
attendances
PK : id
lesson_id
student_id
is_present
students
PK : id
first_name
last_name
email
personal_email
gender
date_of_birth
join_date
left_date
campus_id
promotion
apprenticeships
PK : id
student_id
company_id
start_date
end_date
companies
PK : company_id
name
phone
email
domain
address
city
zip
country
campuses
PK : id
name
address
city
zip
region
country
audiences
PK : id
first_name
last_name
email
gender
date_of_birth
event_id
event
PK : id
date
type
campus_id
location
description

Stockage des données

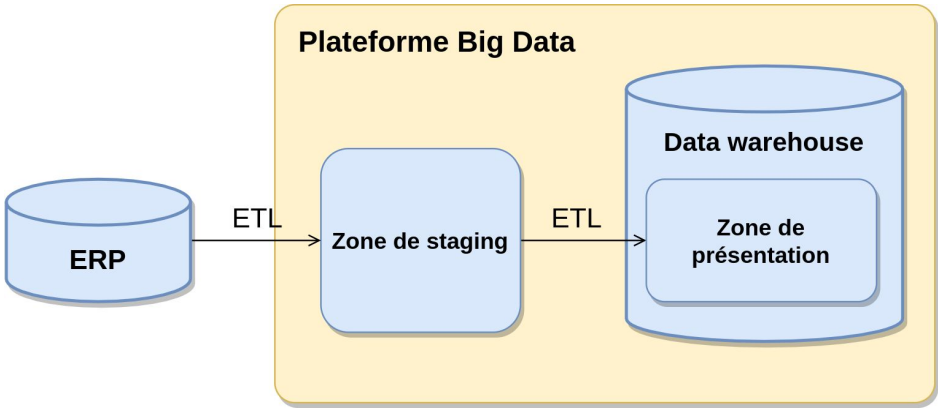
L'objectif de la plateforme Big Data est d'historiser et d'analyser les données de l'ERP sans impacter ce système de production. Pour cela, nous proposons la mise en place d'un Data Warehouse (DWH) avec une architecture 2 tiers :

- Une zone de **staging** où seront déposées les données chaudes telles qu'elles seront extraites depuis l'ERP.
- Une couche de **présentation** avec les données qui auront déjà été traitées une première fois, et où elles seront accessibles par les data scientists et analystes.



Flux ETL

Afin de déplacer les données depuis l'ERP de production vers la plateforme Big Data, et d'extraire des premières connaissances, nous avons identifié les flux ETL suivants qui doivent être mis en place :

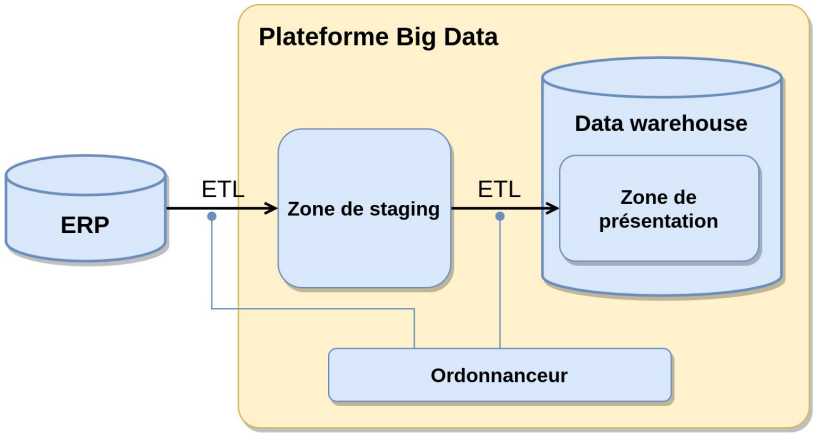


- **extract_all_erp_to_staging :** Ce premier flux ETL doit extraire les données telles qu'elles sont présentes dans chaques tables de l'ERP, puis les déposer sous forme dans la zone de staging. Cela prend la forme de requêtes SELECT pour extraire les données, puis de dépôts de fichiers CSV dans la zone de staging.

- **extract_students_staging_to_public :** L'objectif de ce job est d'extraire du staging les données des étudiants, et de calculer certains indicateurs tels que leurs moyennes, taux de présence en cours, si il ont été recrutés lors d'un événement. *Ce traitement est celui que nous avons choisis d'implémenter afin de démontrer les capacités de notre solution.*

Automatisation

L'expression de besoin évoque que l'exécution des flux ETL doit être industrialisable. Pour cela, un outil d'ordonnancement et de supervision des tâches doit être déployé. En plus d'automatiser l'exécution des flux, celui-ci doit aussi permettre la définition de dépendances entre ceux-ci, permettant la mise en place de workflow complexes de traitements des données.

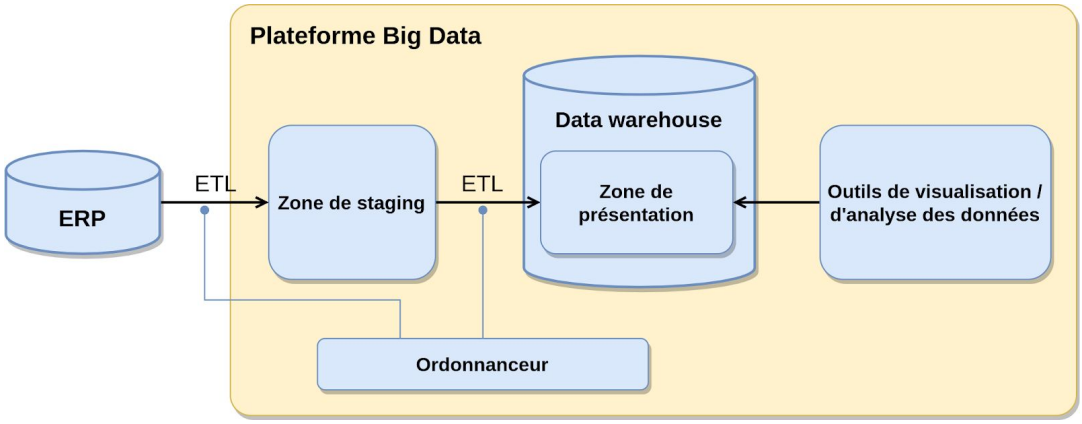


Visualisation

La dernière brique fonctionnelle de la plateforme Big Data est celle de visualisation / d'analyse des données. Celle-ci doit permettre au métier d'accéder aux données ainsi que de les exploiter.

Dans le cas des data analysts, il s'agit en général de solutions de visualisation et de dashboarding.

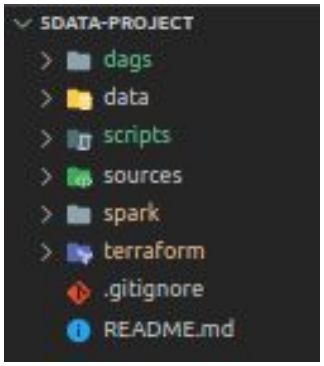
Pour les data scientists, il s'agit d'outils permettant le développement de modèles descriptifs ou prédictifs, telles que des solutions de notebooks.



Implémentation

Afin de fournir un POC d'architecture pour la solution Big Data que nous venons de décrire, nous avons développé un ensemble de scripts et de code permettant de déployer les principaux composants de notre solution, et ainsi démontrer sa faisabilité et la réponse au besoin.

Tout le code est présent dans le dossier joint. Il est aussi accessible via ce repo Gitlab : [5data-project]



Construction du dataset

Comme précisé dans la spec fonctionnelle, le dataset n'étant pas fourni, nous en avons construit un en extrapolant depuis les analyses qui doivent être effectuées.

Pour cela, nous nous sommes aidés du site Mockaroo qui permet de générer des datasets réalistes au format CSV. Via Mockaroo, nous avons générés les fichiers pour :

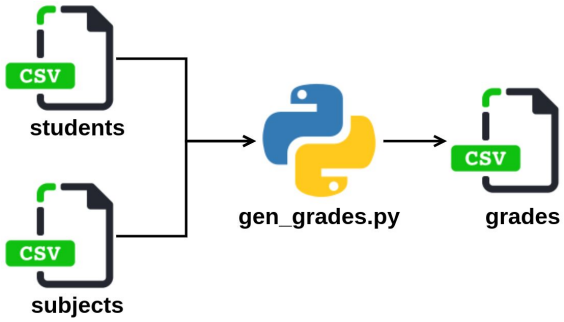
- les étudiants : **students.csv**
- les professeurs : **professors.csv**
- les contacts récoltés lors des évènements et portes ouvertes : **audiences.csv**
- les entreprises : **companies.csv**

extrait de students.csv :

```
id,first_name,last_name,email,personal_email,gender,date_of_birth,join_date,left_date,campus_id,promotion
822491,Alexa,Reid,Alexa.Reid@supinfo.com,Mauris.eu@sempererat.ca,Mme,1980-12-02,2018-01-24,2021-10-26,3,MSC1
777624,Andrew,Hart,Andrew.Hart@supinfo.com,pede@tinciduntnibh.ca,M.,1989-02-27,2020-06-03,2021-05-24,2,BSC
573159,Keane,Carey,Keane.Carey@supinfo.com,lobortis.tellus.justo@metusfacilisis.com,M.,1989-11-10,2013-08-13,2022-01-01,4,MSC2
```

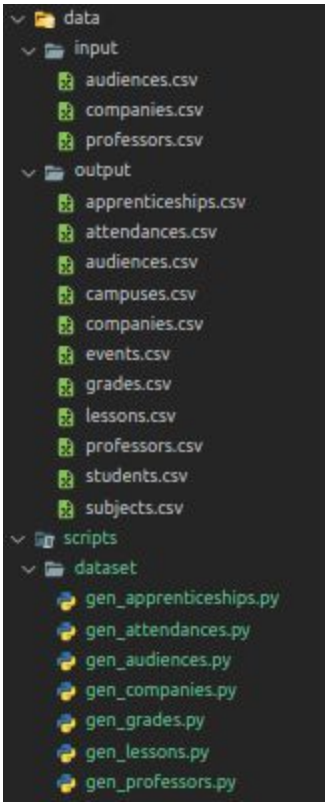
Mockaroo ne permettant pas de définir des règles très complexes pour générer les datasets, nous avons développé un jeu de scripts python permettant de générer le reste du dataset.

Par exemple, le script ***gen_grades.py*** prends en input les fichiers CSV ***students*** et ***subjects***, et génère les notes des étudiants pour les matières de leurs niveau, en suivant une distribution Gaussienne autour de 60



Tous les scripts permettants de générer le dataset se trouvent dans le dossier ***scripts/dataset/***

un dataset déjà généré se trouve dans ***/data/output/***.



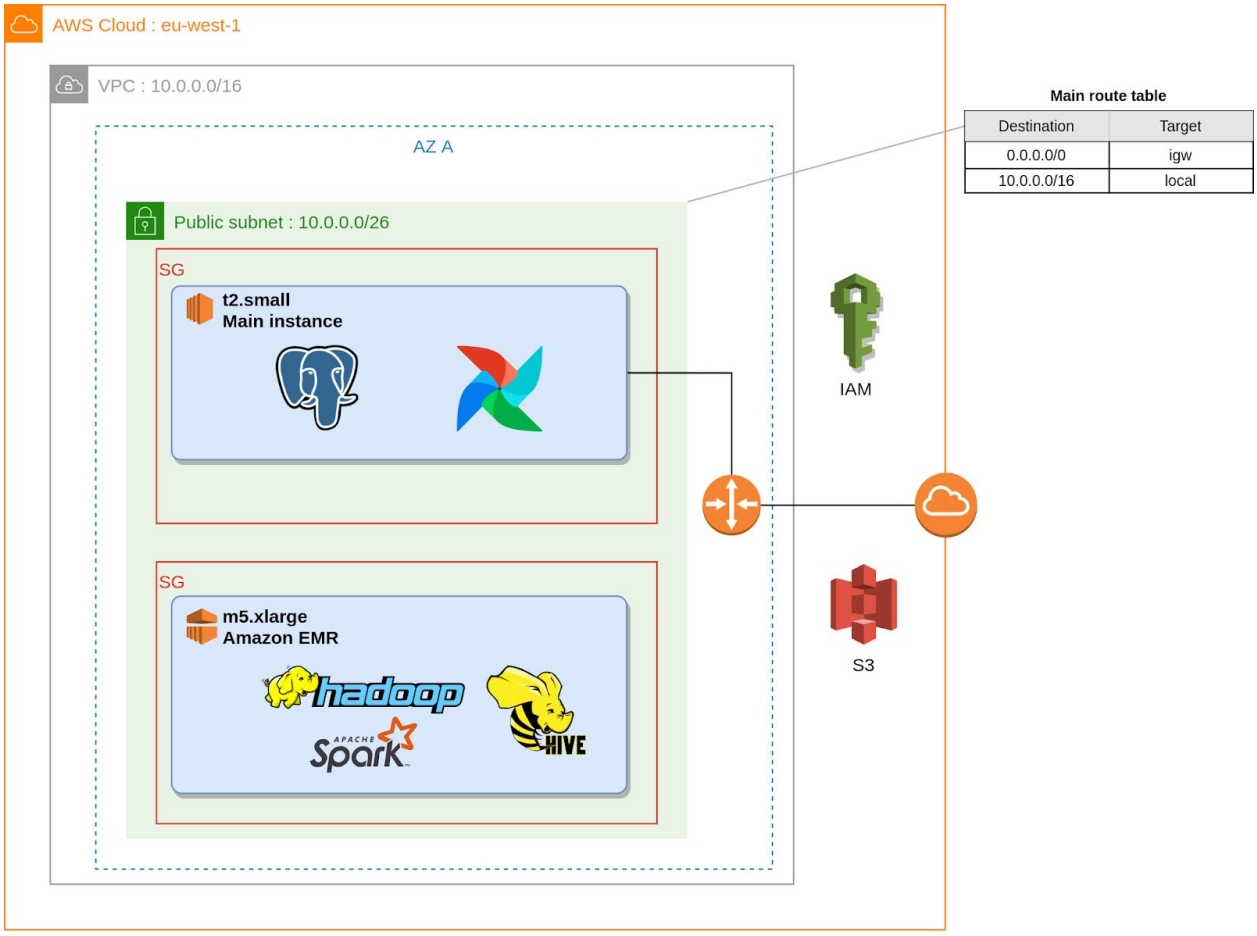
Architecture

Vue globale

Afin de développer notre POC, nous avons fait le choix d'utiliser AWS pour les capacités d'automatisation et les services de cette plateforme.

L'architecture comporte :

- Un Virtual Private Cloud (VPC) situé dans la région eu-west-1 (Irlande)
- Un sous-réseau accessible via internet dans la zone de de disponibilité (AZ) A
- Une instance EC2 t2.small exécutant l'ERP et l'ordonnanceur via Docker
- Un cluster single node Elastic Map Reduce (EMR) exécutant Hadoop, Spark, Hive, Hue
- Un Bucket S3
- Divers composants réseau et sécurité tel que l'Internet Gateway, tables de routages, Security Groups, rôles IAM



Déploiement

Afin d'automatiser au maximum le déploiement de l'architecture et d'employer de bonnes pratiques, nous avons utilisé l'outil d'*Infrastructure as Code* Terraform pour développer notre architecture sur AWS.

Terraform

Cet outil permet de définir et provisionner l'architecture dans des blocs de code appelés "resources", puis de déployer le tout via un terminal.

Cela permet aussi d'avoir une vision claire sur l'ensemble des composants, ainsi que de garantir l'idempotence du déploiement.

```
resource "aws_internet_gateway" "main" {
  vpc_id = aws_vpc.main.id
  tags = {
    "Name" = "5data-igw"
  }
}

resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.main.id
  cidr_block              = "10.0.0.0/26"
  availability_zone       = "eu-west-1a"
  map_public_ip_on_launch = true
  tags = {
    "Name" = "5data-main-subnet"
  }
}
```

L'ensemble du code Terraform est situé dans le dossier ***terraform***.

Pour déployer l'architecture, il faut au préalable :

- installer Terraform sur son poste
- disposer d'un compte AWS configuré avec les accès nécessaires pour faire des appels à l'API
- générer les rôles par défaut pour Elastic Map Reduce

Il faut ensuite créer le fichier ***terraform.tfvars*** dans le dossier ***terraform*** pour renseigner les variables suivantes :

- **key_name** : le nom de la paire de clés qui sera associés aux instances EC2, celle-ci doit être paramétrée sur le service EC2 dans la région *eu-west-1* (Irlande)
- **key_path** : le chemin de la clé SSH pour se connecter aux instances EC2
- **docker_username** : l'identifiant de compte Docker pour permettre à l'instance EC2 principale de télécharger les images Docker
- **docker_password** : le mot de passe du compte Docker
- **bucket_name** : le nom du bucket S3 qui sera créé pour la zone de staging, celui-ci doit respecter les règles de nommages de S3
- **emr_service_role** : l'ARN du rôle de service EMR par défaut créé précédemment
- **emr_instance_profile** : l'ARN du profile d'instance pour les instances EC2 du cluster EMR créé précédemment

Puis, dans le dossier ***terraform***, exécuter :

```
terraform init
terraform apply
```

Terraform va d'abord créer les composants réseau, le VPC, sous réseau, rôles IAM tables de routages. Il va ensuite créer l'instance EC2, puis déployer l'ERP, le peupler, et lancer l'ordonnanceur. Vient ensuite le bucket S3 sur lequel sont aussi copiés les jobs Spark et les scripts d'initialisation du Data Warehouse. Enfin, Terraform crée le cluster EMR avec Hadoop, Hive, Spark et Hue.

Détail de chaque composant

ERP

Pour rappel, nous considérons que le dataset que avons créé, et qui reflète l'activité de Supinfo est stocké dans un ERP.

Dans le cadre de notre POC, nous le simulons par une base de données relationnelle PostgreSQL via un container Docker.

Le script ***scripts/start_erp.sh*** permet de lancer l'ERP, celui ci :

- lance un container ***postgres:alpine***
- crée la base ***ERP***
- applique le schéma situé dans **sources/ERP/DML.sql**
- peuple toutes les tables du schéma avec les données située dans ***data/output/***

```
edgar@[redacted]:~/Supinfo/5DATA/5DATA-project$ docker exec -ti 5DATA-erp psql -U postgres -d ERP -c "\dt"
             List of relations
 Schema |      Name       | Type  |  Owner
--------+-----------------+-------+----------
 public | apprenticeships | table | postgres
 public | attendances     | table | postgres
 public | audiences       | table | postgres
 public | campuses        | table | postgres
 public | companies       | table | postgres
 public | events          | table | postgres
 public | grades          | table | postgres
 public | lessons         | table | postgres
 public | professors      | table | postgres
 public | students        | table | postgres
 public | subjects        | table | postgres
(11 rows)

edgar@[redacted]:~/Supinfo/5DATA/5DATA-project$ docker exec -ti 5DATA-erp psql -U postgres -d ERP -c "select * from subjects limit 5"
  id   |                    name                     | level
-------+---------------------------------------------+-------
 5DATA | Data Engineering                            | MSC2
 4ARCH | AWS Architecture                            | MSC1
 4CCNA | Cisco Networking                            | MSC1
 4DATA | Data Science                                | MSC1
 4DOKR | Developing and Delivering Softwares with Docker | MSC1
```

Ordonnanceur

L'ordonnanceur est le composant logiciel chargé de la planification et de l'exécution des jobs ETL au sein de notre plateforme. Plusieurs outils de ce type existent sur le marché tel que Apache Nifi, Apache Oozie, Dollar Universe ou encore AWS Glue.

Dans notre cas, nous avons choisi Apache Airflow car c'est un outil populaire que nous maîtrisons.

Airflow permet de définir des workflow d'exécution complexes sous forme de graphes acycliques orientés (DAGs) à l'aide de Python, et de planifier leurs exécution. L'avantage de cet outil est qu'il offre de nombreux connecteurs (appelés Operators) vers d'autres systèmes tels que Postgres, S3 ou EMR dans notre cas, ou permet de les développer soit même le cas échéant.

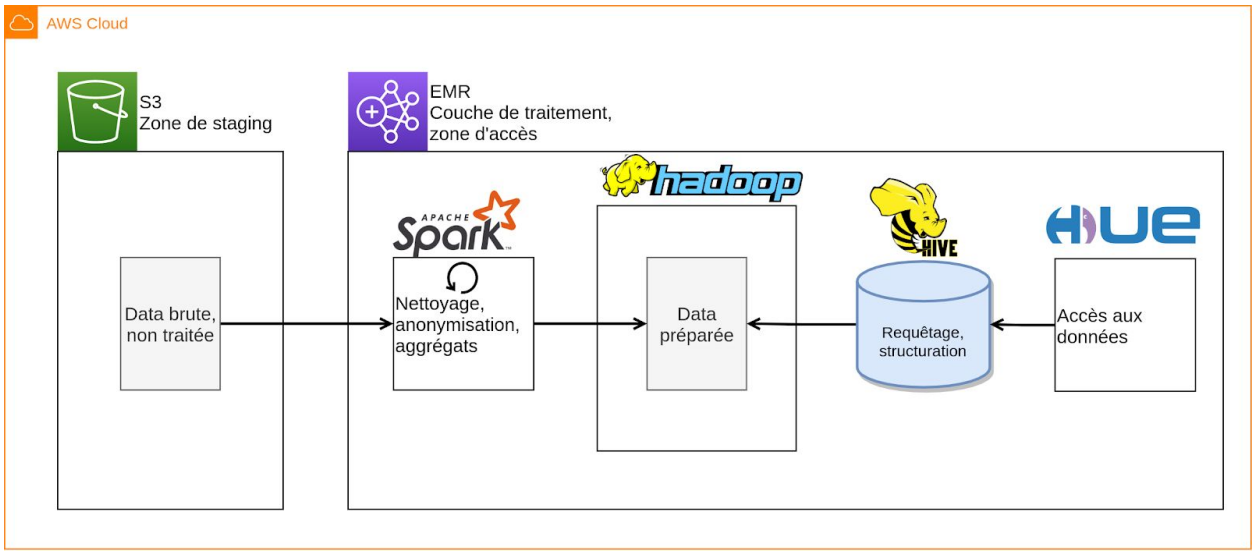
Airflow est exécuté sur l'instance EC2 sur le port 8080.

Data Warehouse, traitement des données

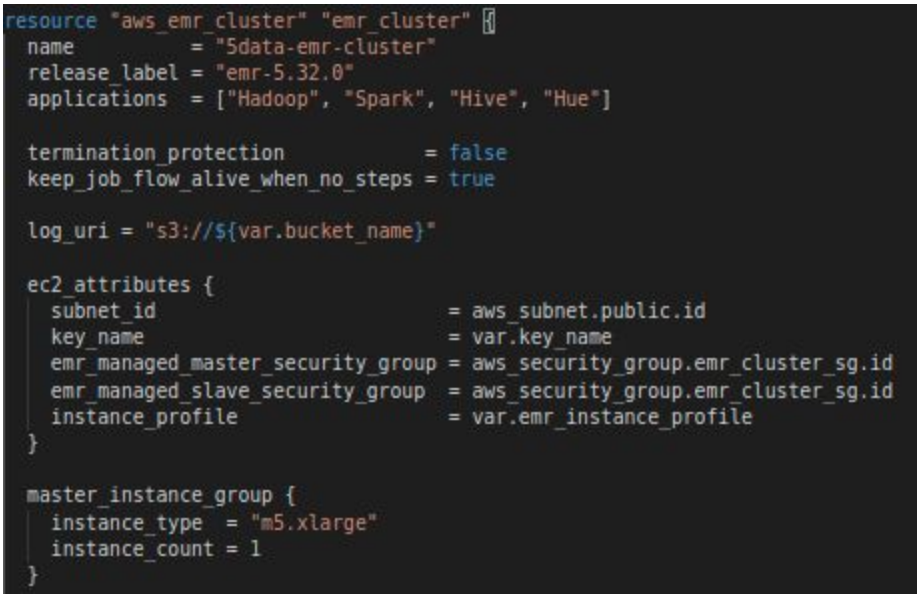
Afin de tirer au mieux profits des services proposés par AWS, nous avons implémenté le modèle de DWH 2 tiers (zone de staging, zone d'accès) en combinant deux services : Amazon Simple Storage Service (S3) et Amazon Elastic Map Reduce (EMR).

S3 est un service de stockage objet qui permet de stocker des quantités massives de données à bas coûts dans des espaces appelés “buckets”. La zone de staging des données est représentée par le répertoires ***“/stage”*** du bucket S3.

EMR est un service qui permet de déployer des clusters de la stack Hadoop, Hive, Spark, Pig, Flink Hbase, etc... L’avantage d’EMR est aussi qu’il étend les fonctionnalités de cette stack et l’intègre avec le reste d’AWS, comme en permettant de transférer des données de S3 à HDFS directement ou de soumettre des traitements (appelés Steps) à Spark sans avoir à se connecter à l’infrastructure sous-jacente.
Dans notre cas, nous déployons un cluster EMR qui fait office de couche de traitement grâce à Apache Spark, et couche d’accès aux données via HDFS, Hive et Hue.



Il s’agit d’un cluster EMR single node en version 5.32 avec une instance EC2 m5.xlarge.



```
resource "aws_emr_cluster" "emr_cluster" {
  name          = "5data-emr-cluster"
  release_label = "emr-5.32.0"
  applications  = ["Hadoop", "Spark", "Hive", "Hue"]

  termination_protection            = false
  keep_job_flow_alive_when_no_steps = true

  log_uri = "s3://${var.bucket_name}"

  ec2_attributes {
    subnet_id                         = aws_subnet.public.id
    key_name                          = var.key_name
    emr_managed_master_security_group = aws_security_group.emr_cluster_sg.id
    emr_managed_slave_security_group  = aws_security_group.emr_cluster_sg.id
    instance_profile                  = var.emr_instance_profile
  }

  master_instance_group {
    instance_type  = "m5.xlarge"
    instance_count = 1
  }
```

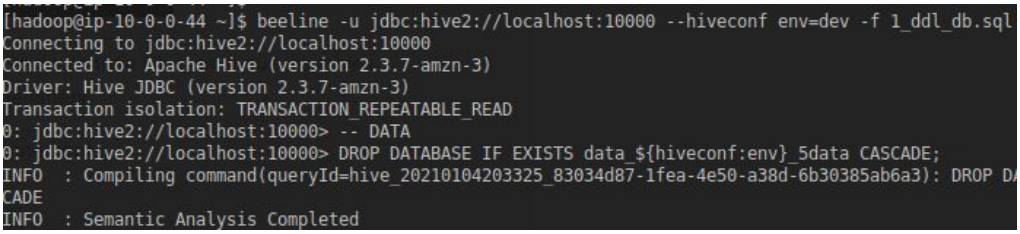
Une fois le cluster déployé par la stack Terraform, le provisionnement du Data Warehouse se fait via les scripts contenus dans le dossier ***scripts/EMR.*** Une fois connecté en via SSH sur le cluster EMR, ceux-ci peuvent être importés depuis S3.

Le premier script ***0_mkdirs_hdfs.sh*** crée :

- l'environnement de développement sur HDFS : ***hdfs://dev/***
- la zone de présentation des données dans l'environnement dev: ***hdfs:///dev/data***
- l'univers dédié à l'import des données de l'ERP dans la zone de présentation : ***hdfs:///dev/data/ERP***
- puis un répertoire HDFS par table cible, par exemple : ***hdfs:///dev/data/ERP/STUDENTS***

Les deux second scripts DML, ***1_ddl_db.sql***, et ***2_ddl_tables.sql*** créent respectivement le schéma Hive, ainsi que les tables Hive qui pointent vers les répertoires HDFS créés précédemment.

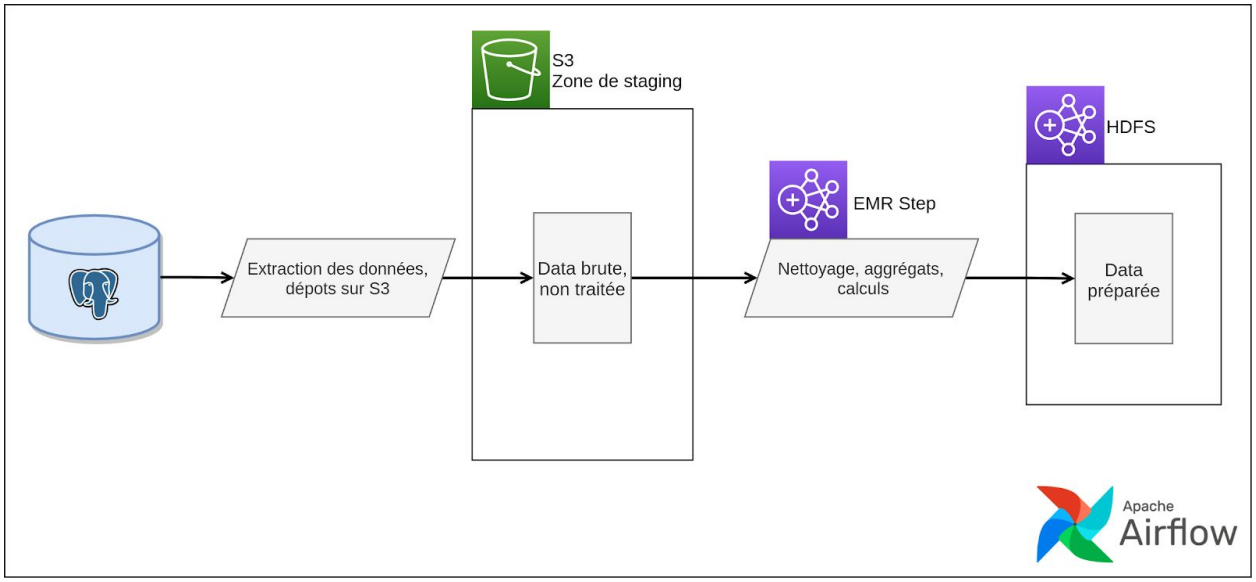
Ils sont exécutables depuis ***beeline***, en spécifiant l'environnement via ***hiveconf*** avec le paramètre ***env***



Flux ETL

Vue globale

Pour rappel, l'objectif des flux ETL est d'intégrer les données produites par l'activité de Supinfo et stockées dans son ERP dans le DWH après avoir été préparées. Pour cela, les données brutes sont d'abord extraites puis déposées dans une zone de staging, puis traitées et intégrées dans la zone de d'acces.



Sur Airflow, un workflow d'exécution est appelé *DAG*, pour *Directed Acyclic Graph* (graphe acyclique orienté). Les sommets du graphe représentent des *Tasks* (tâches) reliées par des arêtes qui définissent leurs relations et dépendances. Chaque *Task* est une implémentation d'un *Operator* qui effectue des actions telle que extraire des données d'une base ou effectuer une requête vers un web service.

Chaque DAG est défini dans un fichier python qui est mis à disposition d'Airflow. Celui que nous avons développé est disponible dans le répertoire ***dags*** sous le nom ***unload_erp.py*.**

```
def query_to_local(sqlFilePath, dest):
    sqlFile = open(sqlFilePath, 'r')
    dml = sqlFile.read()
    pg_hook = PostgresHook(postgres_conn_id='5DATA-ERP')
    conn = pg_hook.get_conn()
    curs = conn.cursor()
    curs.execute(dml)
    results = curs.fetchall()
    with open(dest, 'w') as f:
        writer = csv.writer(f)
        for res in results:
            print(str(res))
            writer.writerow(res)

unload_apprenticeships = PythonOperator(
    dag=dag,
    task_id="unload_apprenticeships",
    python_callable=query_to_local,
    op_kwargs={
        'sqlFilePath': '/usr/local/airflow/dags/scripts/unload_apprenticeships.sql',
        'dest': '/tmp/apprenticeships.csv'
    }
)
```

Déchargement vers la zone de staging

Nous avons développé deux opérateurs Python afin d'extraire les données vers S3 :

- ***query_to_local*** : extrait les données à l'aide du script SQL passé en paramètre, et écrit les données dans le chemin spécifié
- ***csv_to_s3*** : envois les fichier se trouvant dans le répertoire spécifié vers le bucket S3 passé en paramètre

À l'aide de ces deux opérateurs, nous avons construit la première partie de notre flux ETL, à savoir le déchargement des data brutes vers la zone de staging. Pour cela, l'opérateur *query_to_local* est lancé pour chaque table avec une requête spécifique située dans le dossier ***dags/scripts***.

```
unload_audiences = PythonOperator(
    dag=dag,
    task_id="unload_audiences",
    python_callable=query_to_local,
    op_kwargs={
        'sqlFilePath': '/usr/local/airflow/dags/scripts/unload_audiences.sql',
        'dest': '/tmp/audiences.csv'
    }
)

unload_campuses = PythonOperator(
    dag=dag,
    task_id="unload_campuses",
    python_callable=query_to_local,
    op_kwargs={
        'sqlFilePath': '/usr/local/airflow/dags/scripts/unload_campuses.sql',
        'dest': '/tmp/campuses.csv'
    }
)
```

Traitement des données, intégration

La seconde partie du flux ETL consiste au traitement des données brutes, et à l'intégration des données dans le datawarehouse.

Pour illustrer le fonctionnement de la couche de traitement des données, nous avons créé un job Spark. Celui-ci récupère les données des étudiants ainsi que les notes sur le bucket S3 de staging, calcul leurs moyennes, et intègre les données dans le répertoire hdfs ***/dev/data/ERP/STUDENTS*** afin d'alimenter la table STUDENTS.

Le job Spark est disponible sous ***spark/students.py***.

```python
def students(partitions, students, grades, output_uri):

    with SparkSession.builder.appName("Students").getOrCreate() as spark:
        logger.info("Launching spark job")
        students_df = spark.read.csv(students).toDF('id', 'first_name', 'last_name', 'mail', 'personal_mail', 'gender',
        grades_df = spark.read.csv(grades).toDF('id', 'student_id', 'subject', 'date', 'grade')
        float_grades_df = grades_df.select('student_id', grades_df.grade.cast('float'))
        avg_df = float_grades_df.groupBy('student_id').avg('grade')
        students_df_with_avg_grades = students_df.join(avg_df, students_df.id == avg_df.student_id, 'inner')
        if output_uri is not None:
            students_df_with_avg_grades.write.mode('overwrite').csv(output_uri)
```

Une fois le traitement Spark développé, il a fallu l'intégrer au DAG Airflow afin de pouvoir être exécuté avec le reste du workflow.

Comme évoqué précédemment, l'un des avantages d'EMR par rapport à d'autres distributions Hadoop est qu'il permet d'intégrer cette stack avec d'autres systèmes plus facilement. On peut par exemple soumettre des traitements au cluster, appelés *Steps*, au travers de l'API d'EMR.

Un opérateur Airflow permet justement de soumettre des Steps EMR au cluster, il s'agit de l'opérateur ***EmrAddStepsOperator***. Cette opération étant asynchrone, il faut ensuite utiliser l'opérateur ***EmrStepSensor*** afin de suivre l'état du traitement.

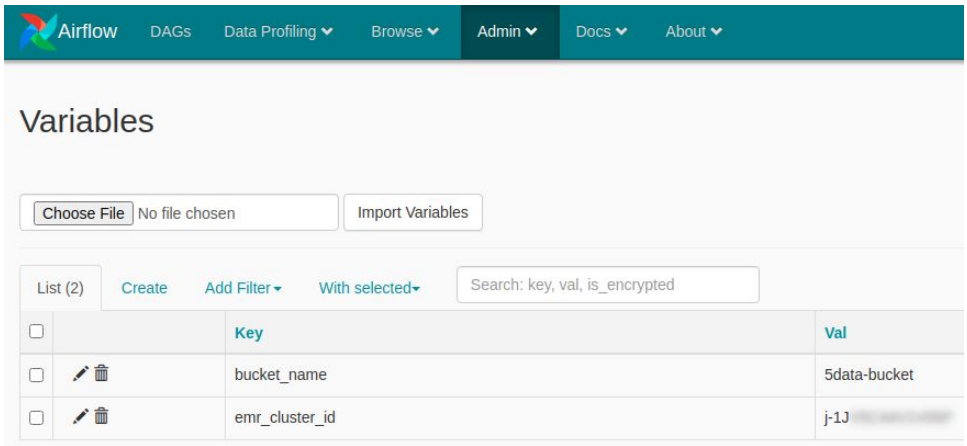
```python
with open('/usr/local/airflow/dags/EMR/steps.json') as steps_file:
    emr_steps = json.load(steps_file)

students_average = EmrAddStepsOperator(
    dag=dag,
    task_id='students_average',
    job_flow_id= Variable.get('emr_cluster_id'),
    aws_conn_id='aws_default',
    steps=emr_steps,
    params = {
        'bucket_name' : Variable.get('bucket_name')
    }
)

step_checker = EmrStepSensor(
    dag=dag,
    task_id='watch_step',
    job_flow_id=Variable.get('emr_cluster_id'),
    step_id="{{ task_instance.xcom_pull('students_average', key='return_value')[0] }}",
    aws_conn_id='aws_default'
)
```

La définition de la Step EMR se fait dans un dictionnaire Python, nous avons donc créé le fichier JSON ***dags/EMR/steps.json*** qui est chargé dans le DAG afin de définir nos Steps.

Celles-ci copient les données depuis S3 vers HDFS pour chaque table excepté pour les étudiants où est exécuté le traitement Spark.

```
{
  "Name": "Copy attendances to HDFS",
  "ActionOnFailure": "CANCEL_AND_WAIT",
  "HadoopJarStep": {
    "Jar": "command-runner.jar",
    "Args": [
      "s3-dist-cp",
      "--src=s3://{{ params.bucket_name }}/stage/attendances.csv",
      "--dest=/dev/data/ERP/ATTENDANCES/attendances.csv"
    ]
  }
},
{
  "Name": "Calculate students average and integrate to HDFS",
  "ActionOnFailure": "CANCEL_AND_WAIT",
  "HadoopJarStep": {
    "Jar": "command-runner.jar",
    "Args": [
      "spark-submit",
      "s3://{{ params.bucket_name }}/spark/students.py",
      "--students",
      "s3://{{ params.bucket_name }}/stage/students.csv",
      "--grades",
      "s3://{{ params.bucket_name }}/stage/grades.csv",
      "--output_uri",
      "hdfs:///dev/data/ERP/STUDENTS"
    ]
  }
}
```

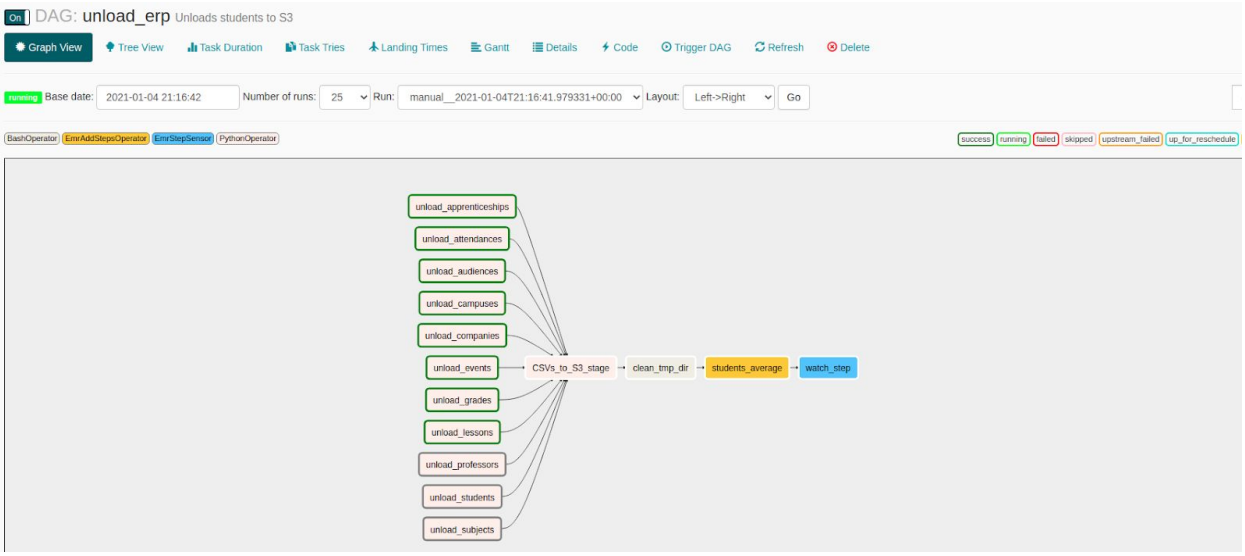
Il faut ensuite renseigner l'ID du cluster EMR à Airflow car nous l'avons variabilisé dans ***emr_cluster_id*** dans le DAG. La variable ***bucket_name*** est créée automatiquement dans Airflow dans notre stack Terraform.



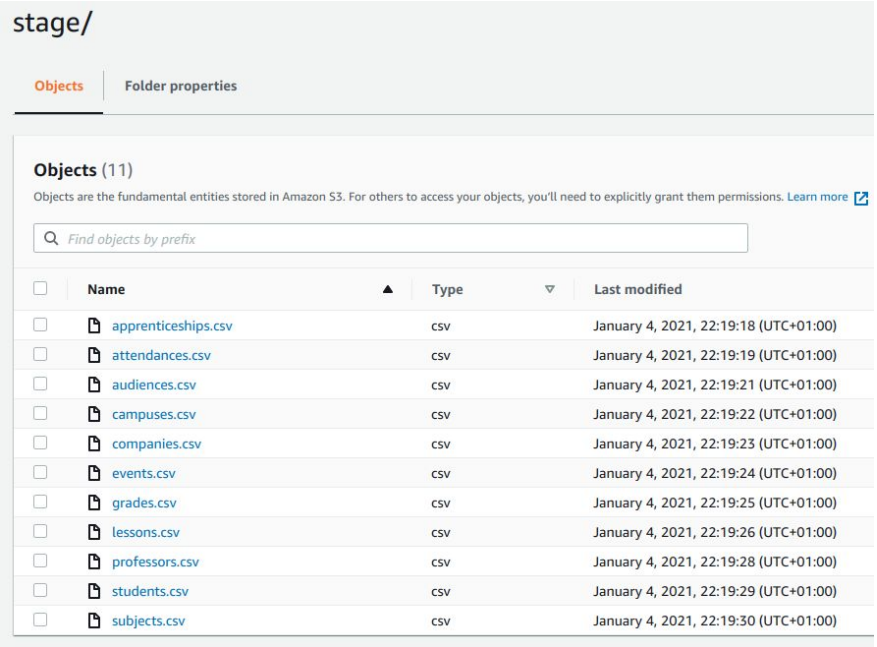
Exécution

Depuis l'interface d'Airflow, dans le menu *DAGs*, exécuter le DAG ***unload_erp***.

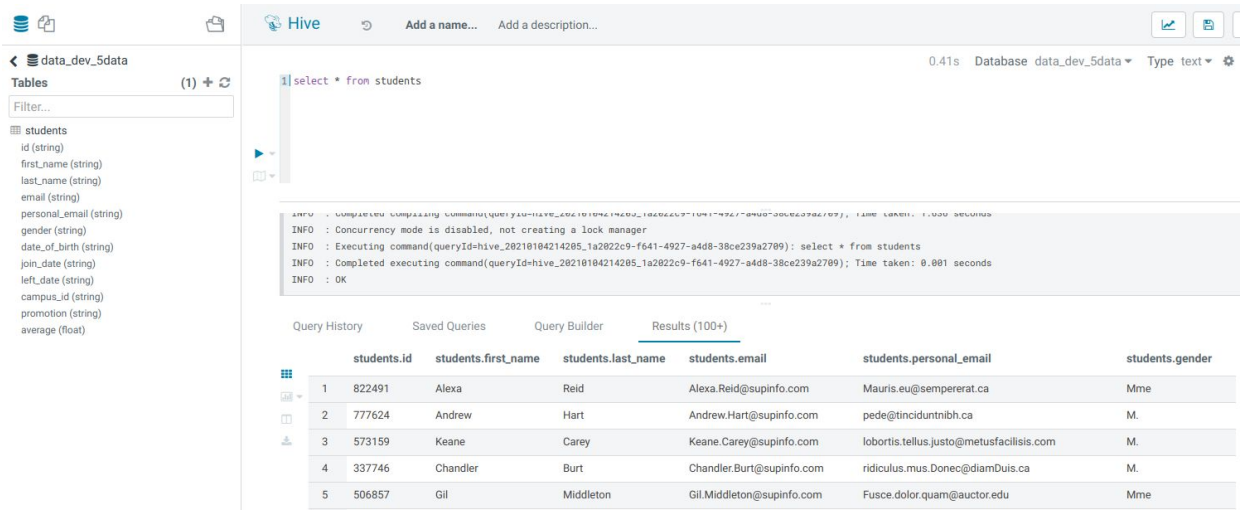
Une fois en cours d'exécution, l'état en temps réel de chaque étape, ainsi que ses logs peuvent être suivis.



Une fois le DAG exécuté, nous pouvons en effet retrouver les données brutes dans le bucket S3.



Ainsi que les données traitées dans HDFS, requêtables depuis Hive. Pour cela, récupérer l'url de connexion à Hue depuis la console de gestion d'EMR, sous l'onglet ***Application user interface***, puis se rendre dans l'éditeur Hive.



Conclusion

Nous pensons que notre plateforme combinant des outils open source tel que Airflow ou Spark couplés aux services d'Amazon Web Services permet de construire un POC de solution qui réponds parfaitement aux besoins de Supinfo, et fournit un outil permettant d'analyser des quantités massives de donnée de façon résiliente, et en faisant bonne usage des ressources.

A l'avenir, notre plateforme pourrait exploiter les capacités de cluster temporaire d'EMR afin d'optimiser encore plus les coûts, et proposer un module de traitement des données en temps réel, aspect que nous aurions aimé avoir le temps d'aborder.