Rendu 2 : Architecture des Données, Choix Technologiques & ETL.

1. Introduction

Le présent rapport détaille la conception, l'implémentation et l'optimisation d'une solution de **matching** entre offres d'emploi et profils candidats. Nous présenterons l'architecture data retenue, le pipeline ELT, les stratégies de matching, ainsi que les enjeux liés à la gouvernance, la sécurité et la performance.

---

2. Architecture des Données & Choix Technologiques

Cette deuxième phase s'inscrit dans la continuité du Rendu 1 (collecte et normalisation des données) et porte sur le design global de l'architecture data : choix du SGBD, modélisation logique (3NF et schéma en étoile) et définition du pipeline ELT. L'objectif consiste à garantir :

- **Performance** : assurer des temps de réponse rapides pour les requêtes analytiques (OLAP) et les opérations de matching candidats-offres (OLTP).
- **Scalabilité & maintenabilité** : structurer les données en couches (RAW, SILVER, GOLD) et centraliser les transformations pour faciliter les évolutions et le versioning.

2.1 Choix du SGBD : Snowflake

2.1.1 Justification

- **Cloud-native** : architecture serverless avec séparation compute/stockage et auto-scaling à la demande.
- **Performance OLAP** : support natif du SQL, clustering automatique des données, cache de résultats pour accélérer les analyses volumineuses.
- **Sécurité & gouvernance** : gestion fine des rôles, fonctionnalité Time Travel pour restauration historique, Zero Copy Cloning pour les tests et audits.

2.1.2 Alternatives considérées

SGBD	Avantages	Limites
PostgreSQL	open-source, extensible	scalabilité verticale limitée
MongoDB	flexibilité NoSQL, schéma souple	complexité OLAP, requêtes lourdes

---

3. Modèle Relationnel 3NF (OLTP)

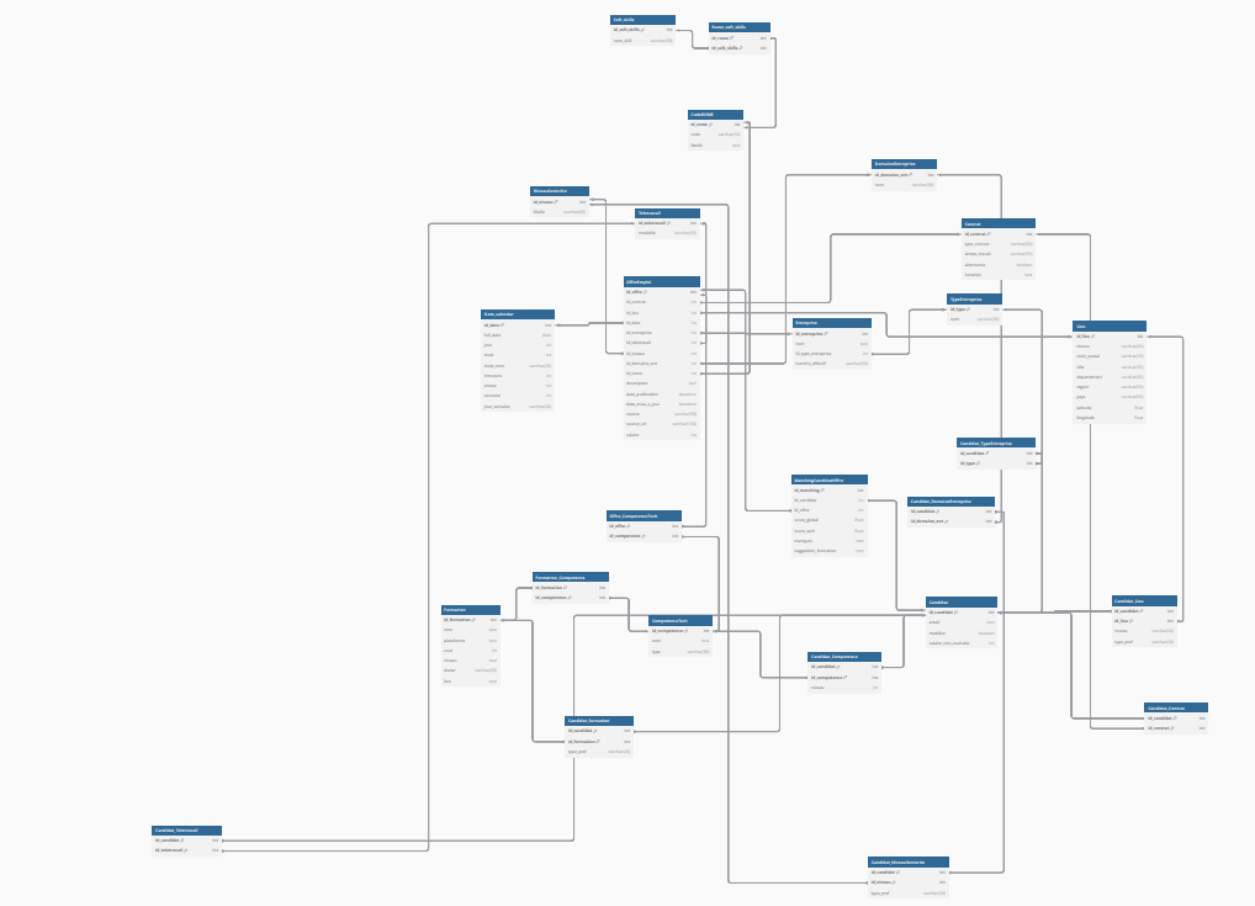


Schéma 1 : modèle 3NF, utilisée pour le matching offre/candidat

Objectif : Garantir l'intégrité référentielle et éliminer la redondance pour un matching fiable et performant entre offres et candidats.

3.1 Principales Entités et Structures

Entité	Description
Offre	Source Adzuna et France Travail d'offreEmploi (id_offre, description, date_publication, salaire, source, etc.)
Candidat	Candidat (id_candidat, email, mobilité, salaire_min_souhaite)
Entreprise	Entreprise (id_entreprise, nom, tranche_effectif, id_type_entreprise)
Lieu	Lieu (id_lieu, niveau, ville, département, région, latitude, longitude)
Date	Date_calendar (id_date, full_date, jour, mois, trimestre, annee, semaine, jour_semaine)

Contrat	Contrat (id_contrat, type_contrat, temps_travail, alternance, horaires)
Compétence	CompetenceTech (id_competence, nom, type)
SoftSkill	Soft_skills (id_soft_skill, nom_skill)
Teletravail	Teletravail (id_teletravail, modalite)
Niveau	NiveauSeniorite (id_niveau, libelle)
CodeROME	CodeROME (id_rome, code, libelle)

3.2 Tables de Liaison

Pour gérer les relations plusieurs-à-plusieurs et les préférences multiples du candidat, des tables de liaison sont mises en place :

- **Offre_CompetenceTech** (id_offre ↔ id_competence) : chaque offre peut contenir plusieurs compétences (language, framework…)
- **Candidat_Competence** (id_candidat ↔ id_competence, niveau) : un candidat peut renseigner plusieurs compétences.
- **Candidat_Lieu** (id_candidat ↔ id_lieu, type_pref) : Un candidat peut avoir des préférences sur plusieurs lieux, il peut être mobile.
- **Candidat_Contrat** (id_candidat ↔ id_contrat) : un candidat peut vouloir choisir de voir les offres proposant CDI et CDD, ou alors stage et alternance.
- **Candidat_Teletravail** (id_candidat ↔ id_teletravail)
- **Candidat_NiveauSeniorite** (id_candidat ↔ id_niveau)

3.3 Table de Matching offre/candidiat

Enfin, Le point central : notre table de matching offre/candidat. A ce stade de notre projet, nous imaginons que cette table sera gérée par du Maching Learning.

3.3 Enrichissements et Dimensions Complémentaires

Nous avons conçu notre schéma 3NF par rapport aux informations présents dans notre source API. Néanmoins, afin d'enrichir nos tables, nous avons imaginé plusieurs autres tables, qui devront être crées à partir d'informations externes.

- **TypeEntreprise** : catégorisation Start-up, PME, ETI, Grand Groupe pour chaque entreprise.
- **DomaineEntreprise** : domaine d'activité principal (Banque, Santé, Transport, etc.).
- **Formation** : lien vers plateau de formation pour combler les manques de compétences.
- **Rome_SoftSkills** : association entre Code ROME et compétences comportementales.

Le schéma 3NF permet de centraliser et d'organiser toutes les données utiles au matching tout en assurant des mises à jour cohérentes et des performances optimales sur les transactions OLTP.

4. Schéma en Étoile (OLAP)

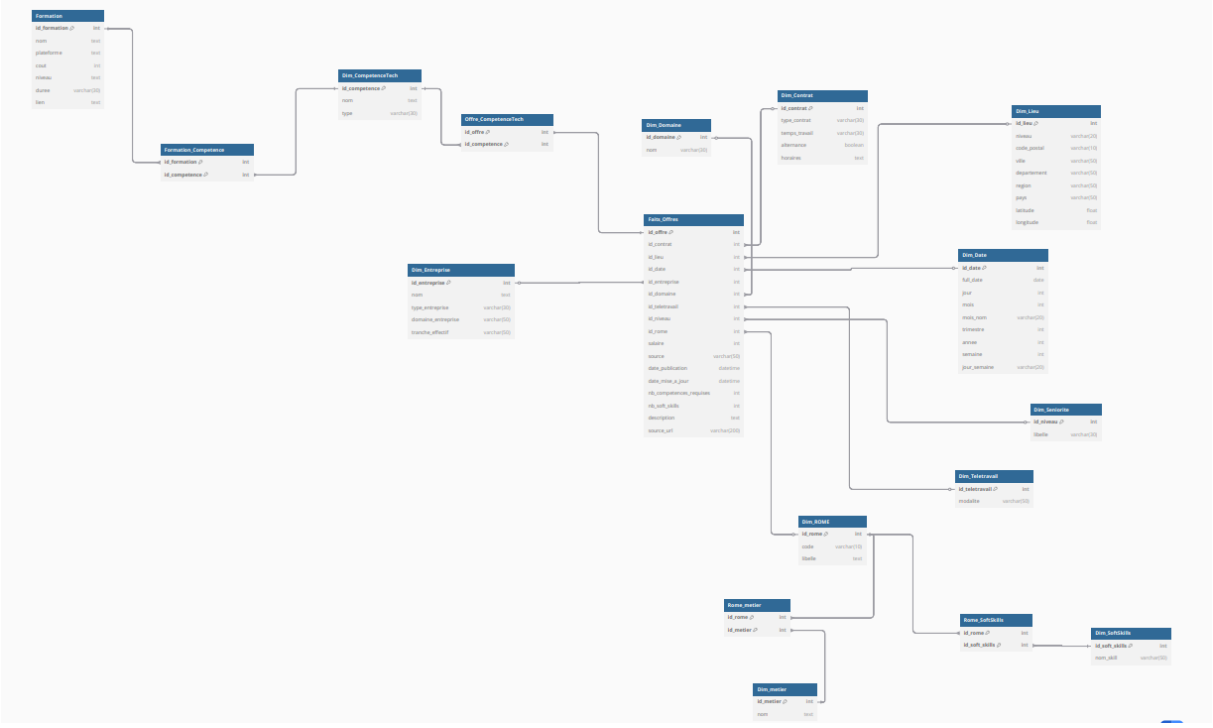


Schéma 2 : Diagramme en Etoile, utilisée pour les requêtes rapides

Objectif : Accélérer les requêtes analytiques (agrégations, slicing & dicing) et faciliter la génération de rapports et de tableaux de bord.

4.1 Table de Faits

Fait offre	Table centrale

4.2 Tables de Dimensions

Dimension	Clés et Attributs Clés
Dim_entreprise	id_entreprise, nom, type_entreprise, domaine_entreprise, tranche_effectif
dim_lieu	id_lieu, ville, département, région, pays, latitude, longitude
dim_date	id_date, full_date, jour, mois, trimestre, semaine, jour_semaine
dim_contrat	id_contrat, type_contrat, temps_travail, alternance, horaires
dim_teletravail	id_teletravail, modalite
dim_seniorite	id_niveau, libelle
dim_rome	id_rome, code, libelle
dim_competence	id_competence, nom, type
dim_softskill	id_soft_skill, nom_skill

4.3 Étapes de Construction

Afin de construire efficacement notre schéma étoile, nous avons dû nous poser les questions des requêtes que nous ferons le plus fréquemment, des insights que nous voudrons avoir. En ayant une vue exhaustive de nos besoins analytiques (OLAP), nous avons pu construire un schéma adapté.

1. **Identification des cas d'usage analytiques** :

Voici quelques-unes des requêtes que nous ferons à partir de notre schéma OLAP :

Requète	Explication
Top des entreprises du marché par secteur d'activité	Nous voulons que le candidat ait une visualisation des acteurs clés du marché, et des domaines (banques, industrie…) qui recrutent le plus
Compétences les plus recherchées pour un métier donné	Cela permettra au candidat de savoir les compétences les plus demandées du moment, afin qu'il puisse comprendre ce que le marché recherche principalement.

Villes offrant le plus d'opportunités d'emploi.	Un candidat qui est mobile pourra avoir une idée des villes les plus dynamiques et ainsi déménager en conséquent.
Offres publiées « aujourd'hui » ou dans une fenêtre temporelle.	En pouvant filtrer sur les offres les plus récentes, le candidat pour être le premier à postuler !

2. **Exemples de requêtes OLAP** :

   - **Top 5 des entreprises** du marché:

```
SELECT d.nom, COUNT(*) AS offres FROM FAIT_OFFRE f JOIN dim_company d ON
f.id_entreprise=d.id_entreprise GROUP BY d.nom ORDER BY offres DESC LIMIT 5;
```

   - **Compétences les plus recherchées du marché** :

```
 SELECT s.nom, COUNT(*) AS freq FROM FAIT_OFFRE f JOIN dim_competence s ON
f.id_rome=s.id_competence GROUP BY s.nom ORDER BY freq DESC;
```

   - **Offres sur Paris**:

```
SELECT l.ville, COUNT(*) FROM FAIT_OFFRE f JOIN dim_location l ON f.id_lieu=l.id_lieu
WHERE l.ville='Paris'
```

Le schéma en étoile pilote nos analyses stratégiques et assure des temps de réponse optimaux pour les tableaux de bord métier.

5. Création des Tables de Dimensions

La définition des tables de dimensions et de leur granularité est cruciale : elle détermine le niveau de détail et la performance des analyses OLAP. Nous avons choisi une granularité **journalière** pour les dates, **niveau entité** pour les entreprises et communes, et **niveau attribut** pour les compétences et autres dimensions.

Illustration avec la création des table DIM_DATE et DIM_LIEU :

5.1 DIM_DATE

- **Granularité** : jour.
- **Objectifs analytiques** :
    - Identifier les jours de la semaine avec le plus d'offres.
    - Mesurer le volume mensuel et hebdomadaire des recrutements.
    - Suivre les offres des 7 derniers jours…
- **Implémentation de la table de dimension associée :** génération avec Python pour créer un Dataframe de dates.

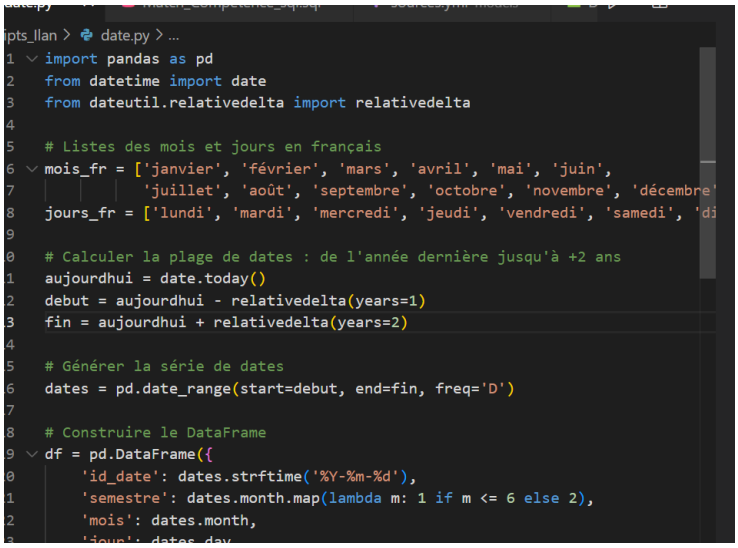


Schéma 3 : Création de la table de dimension DIM_DATE avec un script python

5.2 DIM_ENTREPRISE

- **Source** : API Sirène (fichier .gz). Fichier volumineux de 6GB.
- **Données extraites** : SIREN, nom de l'entreprise, date de création, catégorie (PME, ETI, GE.
- **Étapes de création de la table de dimension** :

1. **Stage Snowflake** : upload du .gz avec Snowflake Connector (Snowpark Python) et commande PUT.

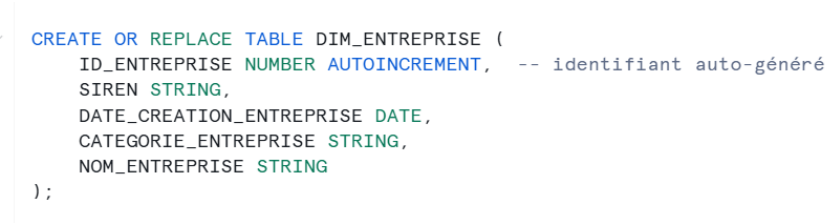
```python

# 2) Exécution du PUT
cs = ctx.cursor()
try:
    local_path = "/home/ilanlp/DBT/Projet_file_rouge/StockUniteLegale_utf8.csv.gz"
    put_sql    = f"PUT file://{local_path} @RAW.sirene_stage AUTO_COMPRESS = FALSE"
    print("Executing:", put_sql)
    cs.execute(put_sql)
    for rec in cs:
        print(rec)  # statut de chaque fichier uploadé
finally:
    cs.close()
    ctx.close()

```

Schéma 4 : Load du fichier .gz dans le stage snowflake grâce au package snowflake connector

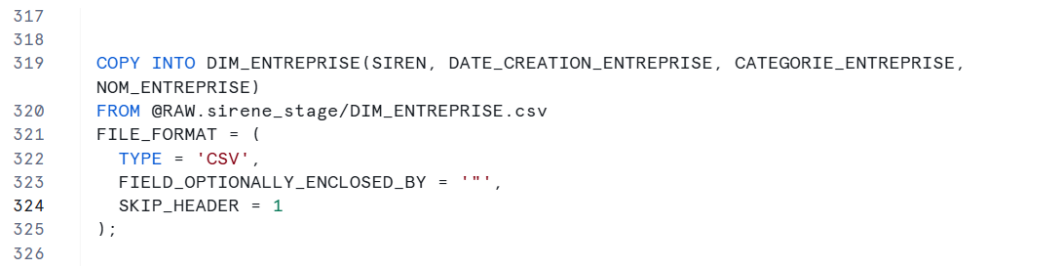
2. **CREATION DU SQUELETTE DE LA TABLE** : choix des colonnes, du type…



```sql
CREATE OR REPLACE TABLE DIM_ENTREPRISE (
    ID_ENTREPRISE NUMBER AUTOINCREMENT,  -- identifiant auto-généré
    SIREN STRING,
    DATE_CREATION_ENTREPRISE DATE,
    CATEGORIE_ENTREPRISE STRING,
    NOM_ENTREPRISE STRING
);
```

Schéma 5 : Création de la table sur Snowflake (avec une clé primaire Auto-incrémentée)

3. **COPY INTO** : insertion dans DIM_ENTREPRISE, nettoyage des colonnes et filtrage des catégories pertinentes.



```sql

COPY INTO DIM_ENTREPRISE(SIREN, DATE_CREATION_ENTREPRISE, CATEGORIE_ENTREPRISE,
NOM_ENTREPRISE)
FROM @RAW.sirene_stage/DIM_ENTREPRISE.csv
FILE_FORMAT = (
  TYPE = 'CSV',
  FIELD_OPTIONALLY_ENCLOSED_BY = '"',
  SKIP_HEADER = 1
);

```

Schéma 6 : COPY INTO d'un fichier CSV dans Snowflake, avec des paramêtres de séparations, entêtes…

Chaque table de dimension a été conçue pour offrir une granularité adaptée aux besoins métiers et garantir des performances optimales lors des analyses OLAP.

6. Pipeline ELT & Orchestration avec dbt

6.1 Chargement RAW et Automatisation

- **Ingestion quotidienne** : un script Python interroge les APIs France Travail et Adzuna pour récupérer les offres des dernières 24 heures (itération sur les codes ROME).
- **Stockage RAW** : chaque flux horodaté est exporté en CSV puis importé dans le schéma RAW de Snowflake via un programme Python et la commande PUT.

```
        # Vérifier les variables d'environnement
        for var in [
            "SNOWFLAKE_USER",
            "SNOWFLAKE_PASSWORD",
            "SNOWFLAKE_ACCOUNT",
            "SNOWFLAKE_WAREHOUSE",
            "SNOWFLAKE_DATABASE",
            "SNOWFLAKE_SCHEMA",
        ]:
            if os.getenv(var):
                logger.info(f"Variable d'environnement {var} définie ✓")
            else:
                logger.warning(f"Variable d'environnement {var} non définie ✗")

        sys.exit(0)

# Exécution normale du script
logger.info("Démarrage du processus d'ingestion dans Snowflake")
ingestion = JobMarketIngestion()
success = ingestion.run_ingestion()

if not success:
    logger.error("Le processus d'ingestion s'est terminé avec des erreurs")
    sys.exit(1)
```

Schema 7 : Vu du Script python d'ingestion du CSV dans le stage de snowflake

6.2 Couches RAW → SILVER → GOLD

1. **RAW** : données brutes sans transformation.2. **SILVER** : préparation et matching avec les dimensions (nettoyage, enrichissement préliminaire).3. **GOLD** : tables finales 3NF et schéma étoile, chargées en mode incrémental pour les dimensions et batch pour la table de faits.

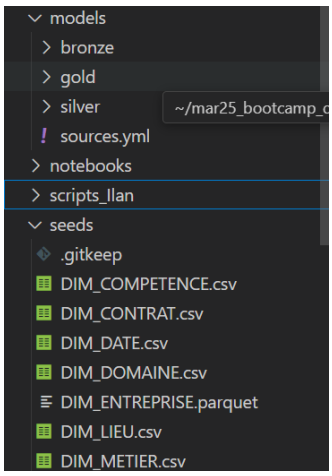


Schéma 8 : Organisation de notre architecture DBT (RAW, SILVER, GOLD)

6.3 Transformations hybrides avec dbt & Snowpark

Pour maximiser la qualité du matching et la maintenance du code, nous avons opté pour une approche **hybride SQL + Python Snowpark** intégrée à dbt. Voici pour illustrer quelques exemples de Matching :

- **Matching entreprise (**DIM_ENTREPRISE**)** :

-1ere etape : Nettoyage : suppression des espaces en début/fin et comparaison case-insensitive + matching sur les différentes colonnes concernées.

```python
from snowflake.snowpark.functions import (
    split, lower, when, lit, col, row_number, element_at,trim,translate
)
from snowflake.snowpark.window import Window

def model(dbt, session):
    raw = session.table("RAW.RAW_OFFRE")
    dim = session.table("SILVER.DIM_ENTREPRISE")

    # 0) Nettoyer les espaces en début/fin
    raw_clean = trim(raw["company_name"])   # ou ltrim(raw["company_name"])
    dim_clean = trim(dim["nom_entreprise"])


    # 1) Extraction du premier mot (élément 1 de l'array)
    raw_first = element_at(split(raw_clean, lit(" ")), 0)
    dim_first = element_at(split(dim_clean, lit(" ")), 0)

    # 2) Conditions de matching
    exact_match = lower(raw["company_name"]) == lower(dim["nom_entreprise"])
    first_word_match = lower(dim_first) == lower(raw_first)
```

Schéma 9 : Nettoyage des données + matching exact/partiel

-2eme étape : Priorité : classification des catégories d'entreprise (GE> ETI> PME> *autres*) et ranking via une fenêtre (row_number() over(Window.partition_by...)). Nous faisons des règles de priorité car il est possible que le matching ai trouvé plusieurs Entreprises. Dans c cas, nous trierons selon les tailles des entreprises.

```python
# 3) Priorité catégorie entreprise
priority = when(df["Categorie_entreprise"] == "GE",  lit(1)) \
           .when(df["Categorie_entreprise"] == "ETI", lit(2)) \
           .when(df["Categorie_entreprise"] == "PME", lit(3)) \
           .otherwise(lit(4)) \
           .alias("priority")

df = df.with_column("priority", priority)

# 4) Fenêtre de ranking : exact > premier mot > partiel > catégorie
w = Window.partition_by("id_local").order_by(
    col("is_exact").desc(),
    col("is_first_word").desc(),
    col("priority").asc()
)

best = (
    df.with_column("rn", row_number().over(w))
      .filter(col("rn") == 1)
      .drop("rn", "priority", "is_exact", "is_first_word")
)

return best
```

Schéma 10 : Priorisation des matching avec la fonction « Window.partition_by »

- **Matching géographique** (DIM_LIEU):
    - **Regex & mots-clés** : extraction du département par regexp_substr et comparaisons de sous-chaînes (contains) + **Distance Haversine** : calcul en radians pour rapprocher latitude/longitude et déterminer la correspondance géospatiale (distance_km < 10).

```
def model(dbt, session):
    name_match   = lower(raw["location_name"]).contains(lower(dim_lieu["ville"]))
    dept_match   = lower(raw["location_name"]).contains(lower(dim_lieu["departement"]))
    region_match = lower(raw["location_name"]).contains(lower(dim_lieu["region"]))
    pays_match = lower(raw["location_name"]).contains(lower(dim_lieu["pays"]))

    # 2. Matching géographique (Haversine)
    lat1 = radians(raw["latitude"]);  lon1 = radians(raw["longitude"])
    lat2 = radians(dim_lieu["latitude"]); lon2 = radians(dim_lieu["longitude"])
    dlat = lat1 - lat2; dlon = lon1 - lon2
    a    = sin(dlat/2)**2 + cos(lat2)*cos(lat1)*sin(dlon/2)**2
    c    = 2 * asin(sqrt(a))
    distance_km = lit(6371) * c
    coord_match = distance_km < lit(10)


    # 4. Fallback code département (2 chiffres)
    #dept_code = expr("regexp_substr(location_name, '\\d{2}')")
    dept_code = expr(
    """
    regexp_substr(
      cast(location_name AS string),
      '\\s*([0-9]{2})\\s*',    -- tolère espaces avant/après
      1,                       -- position de départ
      1,                       -- 1ère occurrence
      'e',                     -- mode extraction
      1                        -- renvoie seulement le groupe 1 (les deux chiffres)
    )
    """
)
```

Schéma 11 : Calcul des distances avec méthode Haversine .

    - **Fallback** : ordre de priorité – coordonnée > nom de ville > code département > région. Nous priorisons les matching sur les villes, et les matching de latitude/longitude. Si un matching se fait avec un département ou une région, alors nous choirons la ville qui aura le plus de population du département ou de la region (grâce à la colonne population)

```
df = (
    raw.join(dim_lieu, join_cond, how="left")
       .select(
           raw["id"].alias("id"),
           raw["id_local"].alias("id_local"),
           raw["location_name"],
           raw["latitude"], raw["longitude"],
           dim_lieu["id_lieu"],
           dim_lieu["ville"].alias("dim_ville"),
           when(coord_match,     1).otherwise(0).alias("is_coord"),
           when(pays_match,    1).otherwise(0).alias("is_france"),
           when(dept_code_match, 1).otherwise(0).alias("is_dept_code"),
           when(name_match,      1).otherwise(0).alias("is_name"),
           when(dept_match,      1).otherwise(0).alias("is_dept"),
           when(region_match,    1).otherwise(0).alias("is_region"),
           distance_km.alias("distance_km"),
           dim_lieu["population"].alias("population")
       )
)
```

Schéma 12 : Priorisation des matching des coordonnées, puis de ville, de départements, régions…

L'architecture ELT orchestrée par dbt assure la traçabilité, le versioning et la documentation automatique (dbt docs), tout en offrant la flexibilité nécessaire pour traiter des transformations complexes.

6.4 Construction de la table de fait « Candidat »

Afin d'avoir une quantité intéressante de candidat dans notre table de fait, nous avons utilisé la librairie « faker ».

Pour cela, nous avons crée un programme python, qui, à partir des tables de dimensions en input, crée des candidats fictifs. Le programme attribue des données sur les compétences du candidat, sa mobilité, son niveau d'études etc. Cette table de fait, comme la table de fait offre, sera une table incrémentale enrichie quotidiennement.

```python
import pandas as pd
from faker import Faker
from faker.providers import BaseProvider
import random
import csv
import re
from typing import Any

# Chargement des données de référence depuis les fichiers CSV
df_communes = pd.read_csv("data/france_travail_communes.csv")
df_departements = pd.read_csv("data/france_travail_departements.csv")
df_regions = pd.read_csv("data/france_travail_regions.csv")
df_formation = pd.read_csv("data/france_travail_niveaux_formations.csv")
df_contracts = pd.read_csv("data/france_travail_types_contrats.csv")
df_permis = pd.read_csv("data/france_travail_permis.csv")
df_metiers = pd.read_csv("data/france_travail_metiers.csv")
df_domaines = pd.read_csv("data/france_travail_domaines.csv")
df_secteurs_activites = pd.read_csv("data/france_travail_secteurs_activites.csv")
```

Schéma 13 : Script python pour la construction de la base de donnée candidat.

7. Sécurité, Monitoring & Gouvernance

• Sécurité : vues restreintes, rôles MODELING/ANALYST/VIEWER, masquage dynamique.
• Documentation : diagrammes dbt, catalogues de schéma, data lineage exposés via dbt docs.

8. Conclusion

L'architecture retenue (Snowflake + dbt + 3NF → schéma en étoile) offre un équilibre entre :
• Intégrité & robustesse (modèle 3NF pour matching, évitant la duplication).
• Performance & agilité (schéma étoile pour analyses rapides).
• Scalabilité & maintenance (architecture en couches, versioning dbt, gouvernance centralisée).