PROJET BIG DATA

LIVRABLE 2

FODIL Nel | MARCELLI Enzo | GOUADFEL Rayan

19 avril 2024

Table des matières

Table des figures

I. Intro

Dans un contexte où le potentiel des données médicales devient de plus en plus crucial, le groupe CHU (Cloud Healthcare Unit) se lance dans une transformation numérique ambitieuse. Après avoir établi un référentiel de données dans le livrable précédent, notre objectif dans ce deuxième livrable est de passer à la phase suivante : la concrétisation du modèle physique de l'entrepôt de données. Cette étape est cruciale car elle permettra d'évaluer la performance de l'accès aux données et d'optimiser les temps de réponse des requêtes, répondant ainsi aux besoins d'analyse et d'exploration des utilisateurs.

II. Contexte du projet

Le secteur de la santé, confronté à une demande croissante d'informations exploitables, doit s'adapter à l'ère numérique pour rester efficace. Le groupe CHU a donc entrepris la construction d'un entrepôt de données pour mieux gérer et analyser les données provenant de diverses sources, telles que les dossiers des patients et les retours sur la satisfaction des soins. Notre mission est d'assurer l'intégration de ces données disparates dans une source unique persistante, répondant ainsi aux exigences d'accès et d'analyse des utilisateurs.

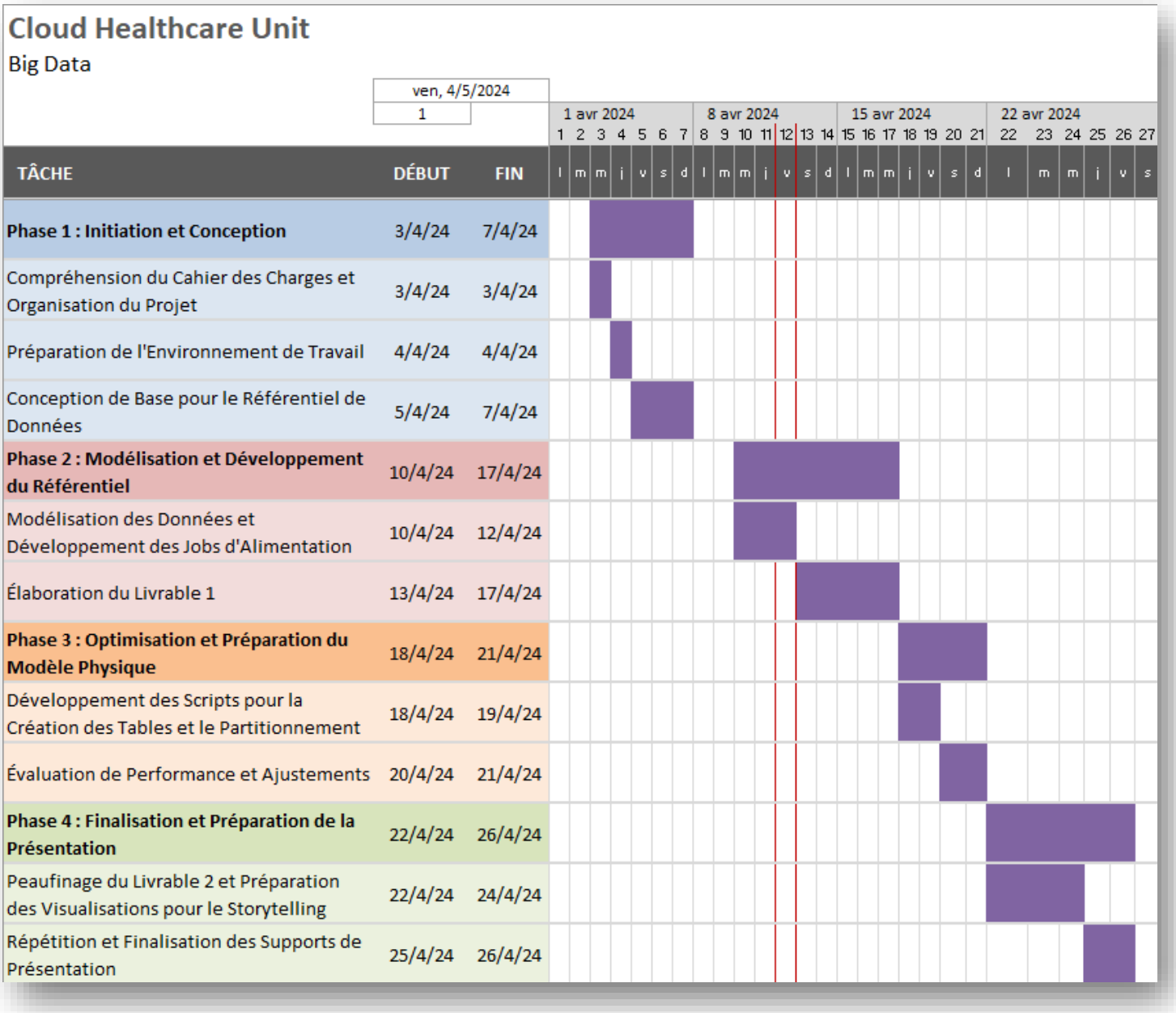
III. Objectif du projet

Dans ce livrable, nous allons :

- Développer un modèle physique des données, traduisant le modèle conceptuel en structures concrètes de tables et de relations.
- Créer des scripts pour la création, le chargement et le peuplement des tables de l'entrepôt de données, en utilisant les différentes sources de données à notre disposition.
- Mettre en place des stratégies de partitionnement et de buckets pour optimiser les performances d'accès aux données.
- Évaluer la performance de l'entrepôt de données en mesurant les temps de réponse des requêtes et en fournissant des graphiques illustrant ces performances.

IV. Rappel Planification Projet

La planification de notre projet CHU Big Data s'étend sur un mois, débutant par la mise en place du projet et l'organisation de l'environnement de travail. Nous consacrons la première semaine à la compréhension du cahier des charges et à la préparation technique. La deuxième semaine est réservée à la modélisation des données et à l'initialisation des jobs d'alimentation, suivi de l'élaboration du premier livrable. La troisième semaine se focalise sur l'optimisation du modèle physique et l'évaluation des performances. Enfin, la dernière semaine est dédiée au peaufinage du second livrable et aux préparatifs de la présentation finale.



TÂCHE	DÉBUT	FIN
Phase 1 : Initiation et Conception	3/4/24	7/4/24
Compréhension du Cahier des Charges et Organisation du Projet	3/4/24	3/4/24
Préparation de l'Environnement de Travail	4/4/24	4/4/24
Conception de Base pour le Référentiel de Données	5/4/24	7/4/24
Phase 2 : Modélisation et Développement du Référentiel	10/4/24	17/4/24
Modélisation des Données et Développement des Jobs d'Alimentation	10/4/24	12/4/24
Élaboration du Livrable 1	13/4/24	17/4/24
Phase 3 : Optimisation et Préparation du Modèle Physique	18/4/24	21/4/24
Développement des Scripts pour la Création des Tables et le Partitionnement	18/4/24	19/4/24
Évaluation de Performance et Ajustements	20/4/24	21/4/24
Phase 4 : Finalisation et Préparation de la Présentation	22/4/24	26/4/24
Peaufinage du Livrable 2 et Préparation des Visualisations pour le Storytelling	22/4/24	24/4/24
Répétition et Finalisation des Supports de Présentation	25/4/24	26/4/24

Figure 1 : GANTT

V. ETL

Actuellement, notre architecture fonctionnelle se limite au Datalake, où toutes nos données brutes sont stockées. Nous devons pleinement exploiter ces ressources en réalisant les traitements nécessaires pour répondre aux exigences des utilisateurs telles que définies dans le cahier des charges.
Afin de réaliser ces traitements, nous allons passer par des ETL, ils vont nous permettre de partir de nos données du DLK, les traiter en fonction de nos besoins pour les stocker finalement dans le DWH.
Pour cela nous avons donc créer des jobs Talend, les voici :

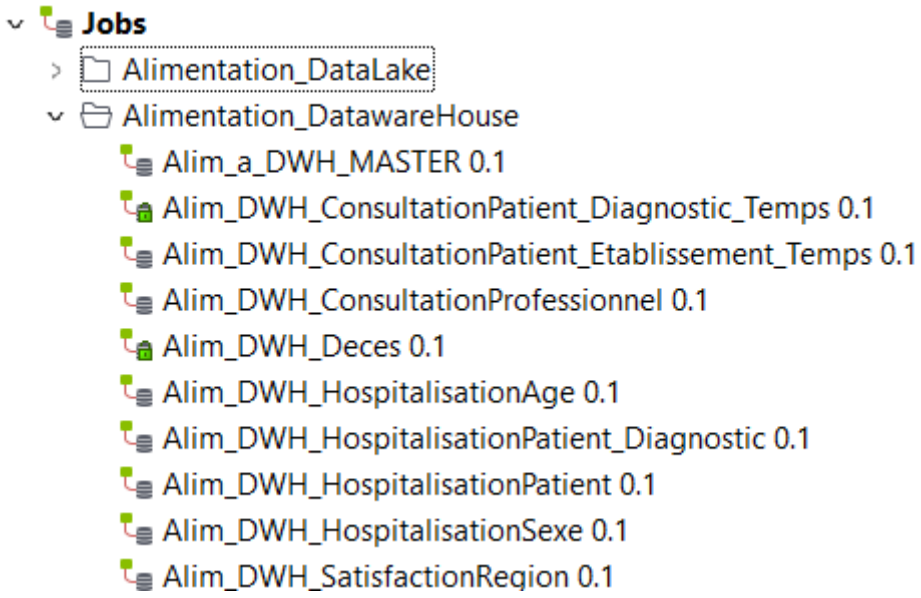


Figure 1 : Jobs DatawareHouse

On y retrouve comme pour le Datalake, le job MASTER, qui permet de lancer tous les traitements en parallèle. Celui-ci s'exécute également automatiquement grâce au planificateur de tâche, il se lance à la suite du DLK.

On y retrouve comme pour le Datalake, le job MASTER, qui permet de lancer tous les traitements en parallèle. Celui-ci s'exécute également automatiquement grâce au planificateur de tâche, et se lance à la suite du DLK, une fois que ce dernier à bien était chargé. Concernant nos autres jobs nous avons fait le choix de segmenter le plus possible nos données d'analyse, afin que chaque table du DWH puisse répondre à un seul besoin utilisateur.

Cette section ne vise pas à détailler le fonctionnement de chaque job, car cela ne serait pas très utile. Notre objectif est plutôt d'identifier et d'expliquer les particularités présentes dans les jobs, répondant ainsi à des besoins de traitement spécifiques.
Contrairement aux ELT où nous avons intégré l'ensemble des données (se référer au livrable 1 pour les arguments), nous allons faire une sélection sur les données utiles ou non dans notre tMap.

Ci-dessous un exemple du tMap du job Décès :

Figure 2 : tMap Job Décès

Notre job dans sa globalité :

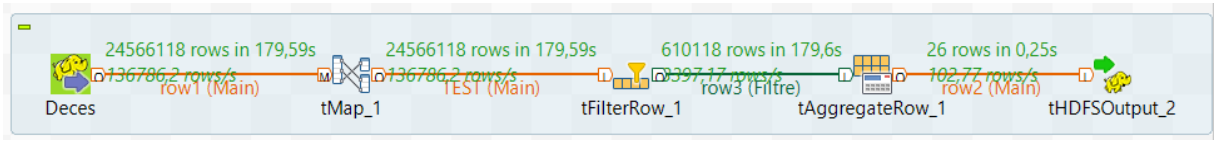


Figure 3 : Job Décès

Comme nous le remarquons, nous mettons en évidence notre décision délibérée de ne sélectionner que les données essentielles au traitement. Cette approche vise à optimiser les performances en réduisant la quantité de données manipulées à chaque étape, ce qui se traduit par des ETL plus rapides.
Dans ce même tMap, pour le job décès, on retrouve un filtre sur les données par rapport à l'année 2019. Les besoins utilisateurs demandent le nombre de décès par région sur l'année 2019. C'est avec la fonction « contains », renvoyant un booléen, que nous allons vérifier les présence de 2019 dans les données du champs « date_deces » :

```
row1.date_deces.contains("2019")
```

Figure 4 : Filtre décès date 2019

Nous allons contenir ce booléen dans un nouveau champ comme on le voit sur le tMap, pour ensuite réaliser un tFilterRow sur ce champ.
On retrouve dans le tFilterRow les conditions suivantes :

Colonne d'entrée	Fonction	Opérateur	Valeur
Boolean_dateDeces	Vide	Vaut	true

Figure 5 : Conditions tFilterRow

Le tFilterRow nous permet de filtrer nos données pour ne conserver que les décès survenus en 2019. Ainsi, nous vérifierons que le booléen est défini sur "true" dans le tFilterRow pour valider le filtre.
A la suite, nous utilisons un tAggregateRow, permettant de réaliser la phase finale de ce job, à savoir compter le nombre de décès par région.

Pour ce fait, voici le contenant de notre tAggregateRow :

Group by

Colonne de sortie	Position de la colonne d'entrée
Region	Region

Opérations

Colonne de sortie	Fonction	Position de la colonne d'e...	Ignorer les valeurs nu
nb_deces	count	numero_acte_deces	☑

Figure 6 : Conditions tAggregateRow

Nous réalisons un « Group by » sur la région et nous faisons un « count » de « numero_acte_deces » pour chaque région. Et nous stockons le nombre de décès par région dans le champ « nb_deces ».

Une fois nos données chargées dans notre DWH, nous avons bien le nombre de décès par région sur l'année 2019, ci-dessous un extrait :

```
Region;nb_deces
['MARTINIQUE'];3555
['AQUITAINE'];35674
['PAYS DE LA LOIRE'];35546
['GUYANE'];1013
['LIMOUSIN'];9821
['CENTRE'];26436
['LORRAINE'];23434
['ALSACE'];16687
['RHONE-ALPES'];54794
['MIDI-PYRENEES'];30269
['PICARDIE'];17882
["PROVENCE-ALPES-COTE D'AZUR"];52567
['BOURGOGNE'];18970
```

Figure 7 : Extrait données décès DWH

On retrouve dans nos traitements une autre spécificité, dans le job « Alim_DWH_SatisfactionRegion ». Pour ce job nous avons cherché à calculer une moyenne à partir d'une opération "group by" sur les régions (dont nous discuterons plus en détail par la suite). Cependant, nous avons rencontré un problème : les données étaient décimales, mais utilisant des virgules au lieu des points habituels. Cela a posé un problème, car le composant tAggregateRow ne reconnaît pas les virgules comme séparateurs décimaux.

Pour faire face à ce problème, nous avons mis en place une variable dans notre tMap avec une expression qui remplace les virgules par des points :

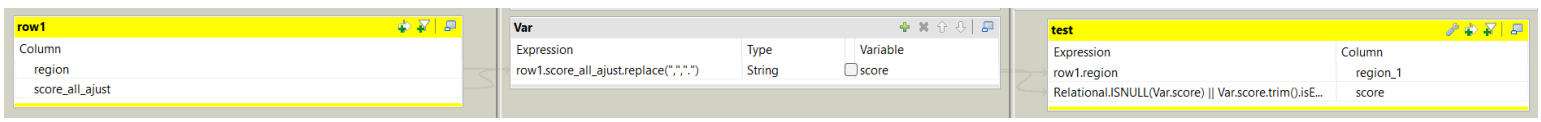


Figure 8 : tMap Satisfaction Region

```
row1.score_all_ajust.replace(",",".")
```

Figure 9 : Variable tMap Satisfaction Region

Par la suite, nous devons convertir notre donnée initialement stockée en tant que chaîne de caractères en format décimal (float) afin de pouvoir calculer une moyenne. Sans cette conversion, cette opération ne serait pas réalisable. Pour ce faire, nous prenons la variable que nous avons mentionnée précédemment et l'utilisons dans l'expression suivante :

```
Relational.ISNULL(Var.score) ||
Var.score.trim().isEmpty() ? null :
Float.parseFloat(Var.score)
```

Figure 10 : Expression Satisfaction Region Conversion String to Float

Le but de cette expression est de gérer les données manquantes ou invalides de manière appropriée lors de la conversion en float dans notre flux de traitement, afin d'éviter toutes erreurs susceptibles de bloquer le traitement.
Finalement, on retrouve un tAggregateRow avec le même fonctionnement que le précédent pour Décès. On vient donc faire un group by, mais cette fois-ci on utilise la fonction « avg » pour faire une moyenne des notes de satisfactions pour chaque région.

Concernant les autres jobs, on retrouve le même schéma : il vise à optimiser les opérations pour ne manipuler que les données nécessaires à l'analyse. Cela inclut des expressions de tri pour filtrer par dates, reformater les dates selon nos besoins ou simplement modifier les caractéristiques de nos données. On y retrouve également des calculs par le biais de fonctions « count » ou « average ». Pour une exploration plus approfondie, le Workspace est disponible sur GitHub. Vous pouvez charger et consulter les différentes tâches.

VI. Automatisation des Processus en Shell

L'automatisation via des scripts shell constitue une composante essentielle dans la gestion des infrastructures de données modernes, telle que Apache Hive sur le Hadoop Distributed File System (HDFS). Ces scripts facilitent non seulement la création et la gestion des tables, mais également leur optimisation à travers le partitionnement et l'utilisation de buckets. Cette automatisation réduit significativement les interventions manuelles, minimise les erreurs, et maximise l'efficacité des traitements. Elle assure également une gestion plus agile des données, une réponse plus rapide aux requêtes, et une adaptation continue aux évolutions des besoins en analyse de données.

Dans le cadre de notre projet, nous avons implanté une automatisation des jobs qui s'activent chaque mois, détaillée dans le livrable 1, pour garantir une mise à jour régulière et systématique des données traitées. L'intégration de processus tels que la surveillance des répertoires pour la détection automatique de nouveaux fichiers, la journalisation des opérations pour un diagnostic précis, et l'application de stratégies de partitionnement et de bucketing, améliore la performance des requêtes et la scalabilité des opérations sur Hive.

Figure 11 : base de données healthcare

1. Scripts principales

Script orchestrateur (main.sh)

Ce script principal, main.sh, orchestre l'exécution des tâches automatisées essentielles à la gestion des données dans un environnement Hive. Il réagit dynamiquement à l'arrivée de nouveaux fichiers de données, assurant ainsi une gestion et une mise à jour continues des tables Hive.

- **Surveillance de Dossier** : Le script utilise la fonction watch_and_process pour surveiller en continu le datawarehouse sur le HDFS spécifié pour l'arrivée de nouveaux fichiers de données. Lorsqu'un nouveau fichier .txt est détecté, le script déclenche automatiquement les scripts de création ou de mise à jour des tables Hive, assurant ainsi que les données les plus récentes sont toujours prises en compte.

```
# Fonction pour surveiller et traiter les nouveaux fichiers
watch_and_process() {
    # Boucle infinie pour surveiller en continu les nouveaux fichiers
    while true; do
        # Détecter les nouveaux fichiers .txt dans le dossier de données local
        new_files=$(find $LOCAL_DATA_PATH -type f -name '*.txt' -mmin -5)

        if [[ ! -z "$new_files" ]]; then
            log_message "New files detected, starting processing..."

            # Déplacer les nouveaux fichiers vers HDFS
            move_txt_files

            # Exécuter chaque script de création/mise à jour de tables
            for script in "${scripts[@]}"; do
                log_message "Updating $script for new data..."
                bash "$LOCAL_SCRIPT_PATH/$script"
            done

            # Exécuter le script de vérification après la mise à jour des tables
            bash "$LOCAL_SCRIPT_PATH/verify_tables.sh"
            log_message "Verification process executed after updating tables."
        else
            log_message "No new files detected."
        fi

        # Attendre un certain temps avant de vérifier à nouveau
        sleep 300  # Temps d'attente en secondes (ici 5 minutes)
    done
}
```

Figure 12 : main.sh

- **Journalisation** : Toutes les activités importantes, y compris la détection de nouveaux fichiers et les opérations sur les tables, sont consignées dans un fichier de log pour faciliter le suivi et le diagnostic en cas de besoin.

Script d'Initialisation (initialisation.sh)

Ce script configure l'environnement nécessaire pour les scripts de création de tables et d'autres opérations.

- **Préparation de HDFS** : Assure la création des répertoires nécessaires dans HDFS et ajuste les permissions pour garantir que les données peuvent être correctement stockées et traitées.

```
setup_dir() {
    # Création des dossiers locaux pour logs s'ils n'existent pas
    chmod -R 777 $LOCAL_SCRIPT_PATH
    if [ ! -d "$LOCAL_LOGS_PATH" ]; then
        mkdir -p $LOCAL_LOGS_PATH
        chmod -R 777 $LOCAL_LOGS_PATH
    fi
}

# Création de la base de données si elle n'existe pas
create_healthcare_db() {
    local db_exists=$(hive -e "SHOW DATABASES LIKE 'healthcare';")
    if [[ -z "$db_exists" ]]; then
        hive -e "CREATE DATABASE healthcare;"
        log_message "Database healthcare created."
    else
        log_message "Database healthcare already exists."
    fi
}

# Configuration initiale pour HDFS
setup_hdfs() {
    hdfs dfs -mkdir -p $DATA_PATH
    hdfs dfs -chmod 777 $DATA_PATH
    log_message "HDFS directory setup and permissions set."
}
```

Figure 13 : initialisation.sh / Préparation de HDFS

- **Transfert des Fichiers** : Déplace les fichiers de données du système local vers HDFS pour les rendre accessibles pour traitement.

```
# Déplacement des fichiers .txt vers HDFS
move_txt_files() {
    found_files=false
    for file in $LOCAL_DATA_PATH/*.txt; do
        filename=$(basename "$file")
        if [[ "$filename" =~ ^(Consultation.txt|Deces.txt|Diagnostic.txt|Etablissement
            hdfs dfs -mv "$file" "$DATA_PATH/"
            found_files=true
        fi
    done
    if [ "$found_files" = true ]; then
        log_message "Eligible files have been moved to HDFS."
    else
        log_message "No new eligible files found to move to HDFS."
    fi
}
```

Figure 14 : initialisation.sh / Transfert des fichiers

2. Création et Gestion des Tables Hive

Nos scripts automatisent la création de tables externes et internes pour chaque ensemble de données, facilitant ainsi la gestion des structures de données au sein de notre environnement Hadoop. Ces tables jouent des rôles distincts en fonction des besoins de traitement et d'accès aux données.

- **Création Automatisée de Tables Externes** : Nos scripts orchestrent la création de tables externes pour chaque type de données. Ces tables stockent les données directement sur HDFS, simplifiant ainsi la gestion et assurant un accès rapide et efficace. La création de ces tables est illustrée par nos scripts, qui préparent les répertoires de données, configurent les permissions, et définissent la structure des tables dans Hive.

```
# Création de la table externe
log_message "Creating external table..."
hive_query_external="USE healthcare;
CREATE EXTERNAL TABLE IF NOT EXISTS external_HospitalisationPatient (
    Id_patient INT,
    Date_Entree STRING,
    Jour_Hospitalisation INT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\;'
STORED AS TEXTFILE
LOCATION '$data_directory'
TBLPROPERTIES ('skip.header.line.count'='1');
"
```

Figure 15 : create_HospitalisationPatient.sh / Create External Table

- **Tables Internes pour Optimisation** : Pour les données nécessitant des optimisations de performance, notamment pour améliorer les temps de réponse aux requêtes grâce au partitionnement et au bucketing, nos scripts créent des tables internes. Ces tables internes sont structurées pour maximiser l'efficacité des opérations de lecture et d'écriture.

```
# Création de la table interne avec partitionnement et bucketing
log_message "Creating internal table with partition and buckets..."
hive_query_internal="USE healthcare; CREATE TABLE IF NOT EXISTS HospitalisationPatient (
    Id_patient INT,
    Jour_Hospitalisation INT
)
PARTITIONED BY (Date_Entree STRING)
CLUSTERED BY (Id_patient) INTO 4 BUCKETS
STORED AS ORC;
"
```

Figure 16 : create_HospitalisationPatient.sh / Create Internal Table

- **Utilisation des Tables Externes** : Lorsque les données ne nécessitent pas de partitionnement ou de bucketing, nous utilisons directement les tables externes. Cela réduit la complexité et est particulièrement utile pour les types de données moins volumineux ou pour des requêtes moins fréquentes.

```
# Insertion des données dans la table interne depuis la table externe
log_message "Inserting data into internal table from external table..."
hive -e "USE healthcare;
INSERT OVERWRITE TABLE HospitalisationPatient PARTITION (Date_Entree)
SELECT Id_patient, Jour_Hospitalisation, Date_Entree FROM external_HospitalisationPatient;
"
```

Figure 17 : Figure 15 : create_HospitalisationPatient.sh / Insertion data

- **Gestion dynamique des tables** : Pour maintenir l'exactitude des données, nos scripts incluent des procédures pour supprimer et recréer des tables si nécessaire. Cela garantit que les données dans Hive sont toujours à jour et synchronisées avec les sources de données, éliminant les redondances et les incohérences potentielles.

```
if [[ $table_exists == *"HospitalisationAge"* ]]; then
    log_message "Table HospitalisationAge already exists ..."
    hive -e "USE healthcare; DROP TABLE HospitalisationAge;"
    log_message "Table HospitalisationAge dropped successfully."
fi
```

Figure 18 : Drop Table

3. Partitionnement et Bucketing des Tables

Dans notre approche optimisée, nous combinons le partitionnement et le bucketing pour améliorer les performances des requêtes en segmentant les données de manière efficace et en les distribuant équitablement à travers des buckets. Cette stratégie est appliquée simultanément dans nos scripts de création de tables Hive pour les données nécessitant à la fois une organisation rapide des données et une récupération efficace.

- **Implémentation du Partitionnement et du Bucketing** : Nous configurons les tables Hive pour utiliser à la fois le partitionnement et le bucketing pour maximiser les performances des opérations de lecture et d'écriture. Cela permet des chargements de données plus rapides et des requêtes plus efficaces en travaillant uniquement sur les segments de données nécessaires.

```
SET hive.exec.dynamic.partition = true;
SET hive.exec.dynamic.partition.mode = nonstrict;
SET hive.conf.validation = false;
SET hive.enforce.bucketing = true;
```

Figure 19 : Configuration HIVE pour le Partitionnement Bucketing

- **Tables Optimisées par Partitionnement et Bucketing** : Nous appliquons cette stratégie à plusieurs tables spécifiques, où le partitionnement est réalisé sur des champs clés et le nombre de buckets est défini pour équilibrer la distribution des données.
    - **ConsultationPatient_Diagnostic_Temps** :
        - **Champs de partitionnement** : **Date**
        - **Bucketing** : 4 buckets basés sur **Id_patient**
    - **ConsultationPatient_Etablissement_Temps** :
        - **Champs de partitionnement** : **Date_Entree**
        - **Bucketing** : 4 buckets basés sur **Id_patient**
    - **HospitalisationPatient** :
        - **Champs de partitionnement** : **Date_Entree**
        - **Bucketing** : 4 buckets basés sur **Id_patient**
    - **HospitalisationPatient_Diagnostic** :
        - Champs de partitionnement : **Date_Entree**
        - **Bucketing** : 4 buckets basés sur **Id_patient**

```
# Création de la table interne avec partitionnement et bucketing
log_message "Creating internal table with partition and buckets..."
hive_query_internal="USE healthcare; CREATE TABLE IF NOT EXISTS HospitalisationPatient (
    Id_patient INT,
    Jour_Hospitalisation INT
)
PARTITIONED BY (Date_Entree STRING)
CLUSTERED BY (Id_patient) INTO 4 BUCKETS
STORED AS ORC;
"
```

Figure 20 : Configuration pour le Bucketing et le Partitionnement.

- **Amélioration des Performances avec le Bucketing** : Nous avons significativement réduit les temps de réponse des requêtes grâce à notre stratégie de bucketing. Voici une comparaison des temps de requête avant et après l'implémentation du bucketing :

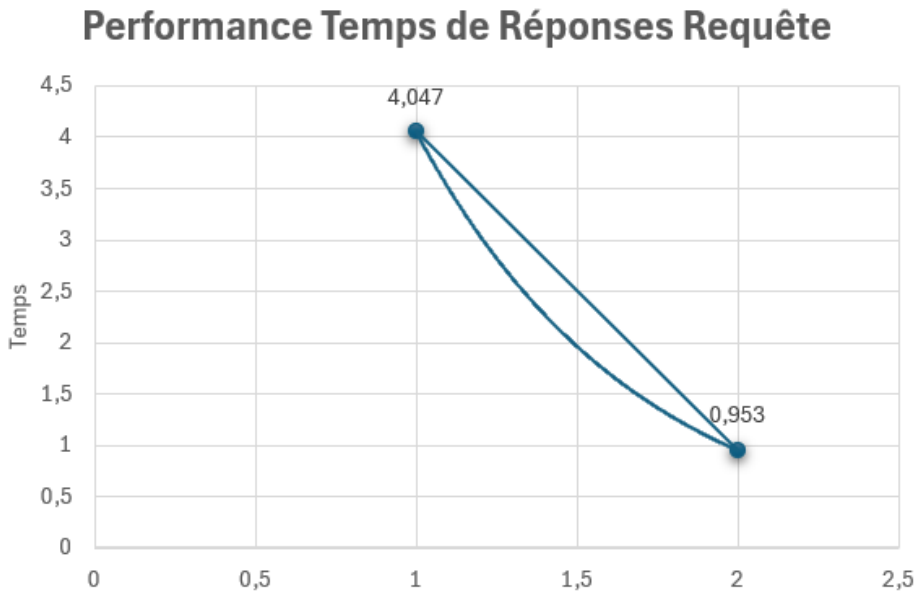


Figure 21 : Graphe temps de réponse

- Avant le Bucketing : *Temps de réponse : 4.047 secondes

```
Time taken: 4.047 seconds, Fetched: 1027157 row(s)
```

Figure 22 : Requête Avant le Bucketing

- Après le Bucketing : *Temps de réponse : 0.953 secondes*

```
Time taken: 0.953 seconds, Fetched: 1027157 row(s)
```

Figure 23 : Requête Après le Bucketing

4. Vérification des Tables

Après la création ou la mise à jour des tables Hive, il est essentiel de valider que les données sont correctement structurées et accessibles. Pour cela, un script de vérification est implémenté pour systématiquement inspecter chaque table afin de confirmer leur intégrité et la précision des données.

- Script de Vérification (verify_tables.sh) : Ce script est exécuté à la conclusion de chaque session de traitement de données pour assurer que toutes les tables ont été correctement créées ou mises à jour et que les données qu'elles contiennent sont complètes et conformes aux attentes. Le script effectue des requêtes ciblées sur chaque table pour vérifier des aspects critiques tels que le nombre d'enregistrements, la présence de toutes les partitions ou buckets attendus, et la validation des champs clés.

- Intégration dans le Processus Automatisé : Le script de vérification est intégré dans le script principal main.sh, étant appelé après chaque cycle de création ou de mise à jour des tables. Cette intégration garantit que toute modification apportée à la structure ou au contenu des données est immédiatement validée, réduisant ainsi le risque d'erreurs ou d'incohérences qui pourraient affecter les analyses ultérieures.

- Journalisation des Résultats : Chaque opération de vérification est consignée dans les fichiers de log dédiés. Ces logs fournissent des détails sur les résultats des vérifications, y compris les succès, les échecs, et les anomalies détectées. Cette documentation est cruciale pour le suivi des performances du système et l'auditabilité des processus.

```bash
# Fonction pour vérifier les données et la structure de la table
verify_table() {
    local table_name=$1
    log_message "Starting verification for table: $table_name"

    # Vérifiez si la table existe
    local table_exists=$(hive -e "USE healthcare; SHOW TABLES LIKE '$table_name';")
    if [[ -z "$table_exists" ]]; then
        log_message "Verification failed: Table $table_name does not exist."
        return
    fi

    # Vérifier la présence de données dans la table
    local has_data=$(hive -e "USE healthcare; SELECT 1 FROM $table_name LIMIT 3;")
    if [[ -n "$has_data" ]]; then
        log_message "Data verification successful for $table_name. Data is present."
    else
        log_message "Data verification warning for $table_name: No data found!"
    fi

    # Vérifiez les partitions si la table est partitionnée
    if [[ "$table_name" == "consultation" || "$table_name" == "deces" || "$table_name" ==
        local partitions=$(hive -e "USE healthcare; SHOW PARTITIONS $table_name;")
        log_message "Partitions in $table_name: $partitions"
    fi
}
```

Figure 24 : verify_tables.sh / Script de Vérification

5. Accès aux Scripts

Tous les scripts utilisés dans ce processus d'automatisation, y compris les scripts d'orchestration, d'initialisation, et de création des tables, sont maintenus et versionnés sur un dépôt GitHub dédié. Ce dépôt permet une collaboration aisée entre les membres de l'équipe et assure une gestion efficace des versions des scripts, facilitant ainsi les mises à jour et les améliorations continues. Le dépôt est accessible publiquement, offrant une transparence totale des processus et des méthodes utilisés dans ce projet.

Le lien vers le dépôt GitHub est le suivant : https://github.com/EnzoMrcli/Projet-Big-Data

VII. Conclusion

Dans ce livrable, nous avons franchi une étape cruciale dans le projet CHU Big Data en concrétisant le chargement de notre entrepôt de données (Data Warehouse). Nous avons mis en place un pipeline ETL permettant de traiter les données selon nos besoins. Cela s'est effectué à travers des jobs Talend pour filtrer, agréger et modifier nos données, ainsi que des scripts SQL automatisés pour créer les tables de notre base de données à partir de nos fichiers .txt. À la suite de la mise en place de l'entrepôt de données, nous prévoyons de créer des tableaux de bord sur Power BI pour visualiser les données que nous avons traitées.
Cette dernière étape viendra finaliser le flux de données de notre architecture Big Data, elle permettra de répondre aux besoins des utilisateurs en leur fournissant des visuels. Ces visuels permettront de donner vie aux données, aidant ainsi nos clients à prendre des décisions éclairées.

VIII. Webographie

https://medium.com/helpshift-engineering/building-a-data-warehouse-with-hive-at-helpshift-part-1-443046df6484

https://www.analyticsvidhya.com/blog/2021/05/hive-a-data-warehouse-in-hadoop-framework/

https://www.lebigdata.fr/apache-hive-definition

https://data-flair.training/blogs/apache-hive-architecture/

https://meritis.fr/hive-et-big-data-exploration-des-concepts/

https://openclassrooms.com/fr/courses/4462426-maitrisez-les-bases-de-donnees-nosql

https://openclassrooms.com/fr/courses/4462426-maitrisez-les-bases-de-donnees-nosql/4462471-maitrisez-le-theoreme-de-cap

https://infodecisionnel.com/data-management/big-data/acid-versus-base/

https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.0/using-hiveql/content/hive_using_materialized_views.html

https://www.talend.com/fr/resources/elt-vs-etl/

https://www.ediservices.com/fr/etl-integration/

https://www.cartelis.com/blog/data-lake-vs-data-warehouse/

https://openclassrooms.com/fr/courses/4467481-creez-votre-data-lake/4467488-identifiez-les-besoins-de-votre-data-lake

https://www.data-transitionnumerique.com/cube-olap-decisionnel-big-data/

https://moodle.cesi.fr/pluginfile.php/24914/mod_resource/content/4/res/Informatique_Desicionnelle.pdf

https://www.edureka.co/blog/videos/etl-using-big-data-talend/

https://www.edureka.co/blog/talend-big-data-tutorial/

https://blog.octo.com/levolution-des-architectures-decisionnelles-avec-big-data/

https://www.cetic.be/Comment-deployer-avec-succes-un-projet-Big-Data

https://www.solution-bi.com/fr/blog/la-modernisation-du-data-warehouse-a-lere-du-big-data

https://docs.microsoft.com/fr-fr/azure/architecture/guide/architecture-styles/big-data

https://docs.microsoft.com/fr-fr/power-bi/fundamentals/power-bi-overview