



Université Chouaib Doukkali
Ecole Nationale des Sciences Appliquées d'El Jadida
Département Télécommunications, Réseaux et Informatique



kaggle



GAME OF
THRONES

Analyse du réseau de Game of Thrones en utilisant Apache GraphX, Neo4j et Spark ML



Réalisé Par :

Elhadi Refki

Reda Makaoui

Oussama Ouhayou

Encadré Par :

Fahd Kalloubi

Table des matières

1	Introduction.....	4
2	Objectif :	4
3	Étapes du Projet :	4
4	Dataset:	5
5	Architecture :	6
6	L'image Docker :	7
7	Importation des JARs :	8
8	Manipulation & Insertion des données:	10
8.1	Récupération et Manipulation des données.....	11
8.2	Insertion des données	12
8.3	Vérification dans Neo4j	13
9	CRUD :	13
9.1	Récupération d'après Neo4j	13
9.2	Exemples de filtrages	16
9.3	ADD	20
9.4	EDIT	21
9.5	DELETE.....	22
10	GraphX :	22
10.1	Récupération des données et création du graph	22
10.2	PageRank Algorithm.....	25
10.3	Connected Components Algorithm.....	26
10.4	Label Propagation Algorithm.....	26
10.5	Triangle Count Algorithm.....	27
10.6	Strongly Connected Algorithm	28
11	Visualisation des résultats GraphX via tableau :	29
12	Spark ML :	32
12.1	Récupération des données	32
12.2	Enrichir le dataset.....	33
12.3	Application de quelques algorithms de prédiction :	36
a.	Prediction du Degree_Centrality :	36
b.	Prediction du PageRank :	37
c.	Prediction du Betweenness_Centrality	37
d.	Prediction du Weighted_Degree	38
12.4	: Clustering du Weight (KMeans).....	39

Table des Figures

Figure 1: Dataset Kaggle	5
Figure 2: Architecture du projet	6
Figure 3 docker-compose.yml file.....	8
Figure 4 le cluster docker	8
Figure 5 version de spark dans zeppelin	8
Figure 6 fichier JAR de connexion de neo4j et spark	9
Figure 7 importation du jar dans zeppelin.....	9
Figure 8 configuration du JAR.....	10
Figure 9 les notebook du Projet.....	10
Figure 10 code de concatenation des fichier CSV	11
Figure 11 stockage dans un nouveau fichier result.csv	12
Figure 12 insertion des données	12
Figure 13 visualisation	13
Figure 14 configuration de l'interpréteur Neo4j	13
Figure 15 connecter l'interpreteur avec le contenuer neo4j.....	14
Figure 16 visualisation dans zeppelin	14
Figure 17 lire des données de neo4j dans zeppelin pour les manipuler	15
Figure 18 créer un graph à partir des données.....	15
Figure 19 visualiser les données en utilisant les CYPHER query.....	16
Figure 20 visualiser les données en utilisant SPARK	16
Figure 21 les nœuds du premier livre	17
Figure 22 les nœuds du 2eme livre.....	17
Figure 23 les nœuds du 3eme livre.....	17
Figure 24 les nœuds du 4eme livre.....	18
Figure 25 les nœuds du dernier livre	18
Figure 26 les personnages qui sont dans tous les livres	19
Figure 27 la personne la plus présente dans la série.....	19
Figure 28 ajouter un noeud.....	20
Figure 29 verifier l'ajout.....	20
Figure 30 modification des données.....	21
Figure 31 vérification de la modification	21
Figure 32 la suppression des données	22
Figure 33 Récupération des données.....	22
Figure 34 créer un Graphe à l'aide de graphX pour les données récupérés.....	23
Figure 35 Affichage des (vertices et Edges)	23
Figure 36 des statistiques sur le graph.....	24
Figure 37 afficher les degrés des noeuds.....	24
Figure 38 PageRank Algorithm.....	25
Figure 39 resultat de l'algorithme.....	25
Figure 40 Connected Components Algorithm.....	26
Figure 41 label propagation algorithme	27
Figure 42 triangle count algorithme	28
Figure 43 strongly connected algorithme	29
Figure 44 resultat de l'algorithme.....	29
Figure 45 Page Rank algorithme	29
Figure 46 connected algorithme	30

Figure 47 Label Propagation Algorithm	30
Figure 48 Triangle Count	31
Figure 49 strongly Connected algorithm	31
Figure 50 Dashboard	32
Figure 51 récupération des données	33
Figure 52 affichage des données	33
Figure 53 resultat.....	35
Figure 54 Prediction du Degree_Centrality.....	36
Figure 55 resultat.....	36
Figure 56 les métriques du model	36
Figure 57 Prediction du PageRank.....	37
Figure 58 les metriques du model	37
Figure 59 Prediction du Betweenness_Centrality	37
Figure 60 resultat.....	38
Figure 61 les metriques du model	38
Figure 62 Prediction du Weighted_Degree	38
Figure 63 resultat.....	38
Figure 64 les métriques	38
Figure 65 le clustering avec Kmeans	39
Figure 66 affichage des clusters	39

1 Introduction

Ce projet s'attache à décortiquer le réseau complexe de Game of Thrones en utilisant des outils technologiques avancés tels qu'Apache GraphX, Neo4j, et Spark ML. En plongeant dans les relations entre les personnages, les maisons nobles, et les événements clés de la série, nous cherchons à élucider les dynamiques politiques et sociales qui façonnent l'intrigue. En combinant la puissance des graphes, des bases de données orientées graphe, et du machine learning, cette analyse promet une exploration approfondie du tissu relationnel unique de l'univers **de Game of Thrones**.

2 Objectif :

Concevoir et mettre en œuvre une solution complète d'analyse de réseau social basée sur le dataset Game of Thrones en utilisant les technologies Spark GraphX, Neo4j, Apache Zeppelin, Spark ML, et en intégrant des visualisations dans un tableau de bord personnalisable. L'objectif est de comprendre les relations entre les personnages de Game of Thrones, d'effectuer des analyses exploratoires, d'appliquer des algorithmes de graphes, et d'appliquer des modèles d'apprentissage automatique pour tirer des insights intéressants du dataset.

3 Étapes du Projet :

Lire le Contenu du Dataset :

- Télécharger le dataset Game of Thrones depuis le lien Kaggle fourni.
- Explorer le dataset pour comprendre sa structure et son contenu.

Connecter Spark GraphX avec Neo4j :

- Suivre la documentation de Neo4j pour intégrer Spark GraphX avec Neo4j.

Importer le Dataset sur Neo4j :

- Utiliser Cypher Query Language pour importer les données du dataset Game of Thrones dans Neo4j.

Analyse Exploratoire avec Apache Zeppelin et GraphX :

- Lire les données depuis Neo4j avec Apache Zeppelin et GraphX.
- Effectuer une analyse exploratoire du réseau social (e.g., degré de centralité, communautés, etc.).

Exécuter 5 Algorithmes de Graphes avec GraphX :

- Choisir cinq algorithmes de graphes (par exemple, PageRank, Label Propagation, etc.).
- Appliquer ces algorithmes sur le réseau social Game of Thrones.
- Visualiser les résultats obtenus.

Créer un Dashboard Personnalisable :

- Utiliser un outil de tableau de bord (par exemple, Tableau, Power BI) pour créer un tableau de bord interactif.
- Intégrer des visualisations basées sur les analyses exploratoires et les résultats des algorithmes de graphes.

Appliquer des Algorithmes ML avec Spark ML :

- Utiliser Spark ML pour appliquer des algorithmes d'apprentissage automatique sur les données Game of Thrones (par exemple, classification des personnages en fonction de certains attributs).

4 Dataset:

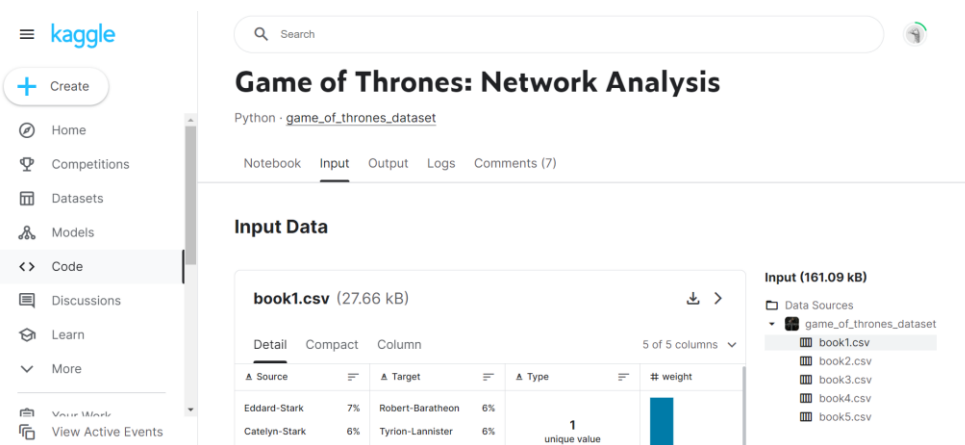


Figure 1: Dataset Kaggle

<https://www.kaggle.com/code/mmmarchetti/game-of-thrones-network-analysis/input>

Cette dataset est utilisé pour réaliser une analyse de réseau sur les personnages de la série de livres "Game of Thrones" de George R.R. Martin. Le jeu de données représente un réseau ou un graphe où les personnages sont des nœuds, et les connexions entre eux (arêtes) représentent les co-occurrences dans un certain contexte dans le texte.

L'analyse de cette dataset permet plusieurs investigations :

- **Relations entre personnages :** Elle permet de comprendre les relations ou interactions entre les personnages en se basant sur leurs co-occurrences. Par exemple, des personnages souvent mentionnés ensemble peuvent signifier des relations, des alliances ou des conflits.
- **Importance des personnages :** En appliquant diverses mesures d'analyse de réseau telles que la centralité degré, la centralité d'intermédierité et le PageRank, cela évalue l'importance ou la centralité des personnages dans le récit. Cela peut montrer qui a le plus d'influence, qui relie différentes parties de l'histoire ou joue un rôle crucial.
- **Évolution des personnages :** Suivre les changements dans l'importance des personnages à travers différents livres offre des perspectives sur les arcs narratifs des personnages, le développement de l'intrigue ou des changements significatifs dans la focalisation narrative.
- **Analyse comparative :** Comparer la centralité des personnages à travers les

livres peut révéler comment leurs rôles et leur importance évoluent avec le temps ou en raison de certains événements dans l'histoire.

Voici un aperçu de ce que représente chaque caractéristique :

- **Source** : Cette caractéristique représente le nœud source dans le réseau, indiquant un personnage impliqué dans une interaction ou une co-occurrence.
- **Target** : Elle indique le nœud cible dans le réseau, représentant un autre personnage impliqué dans la même interaction ou co-occurrence que le personnage source.
- **Type** : Cette caractéristique spécifie le type de relation ou d'arête entre les nœuds source et cible. Dans cette analyse, il est mentionné que toutes les arêtes sont "Non dirigées", indiquant une relation non directionnelle entre les personnages.
- **Weight** : L'attribut de poids représente la force ou la fréquence des interactions ou co-occurrences entre les personnages source et cible. Une valeur de poids plus élevée signifie une relation plus forte ou des co-occurrences plus fréquentes.
- **Book** : Cette caractéristique représente le numéro du livre ou le volume dans la série "Game of Thrones" où ces interactions ou co-occurrences entre les personnages ont été observées.

5 Architecture :

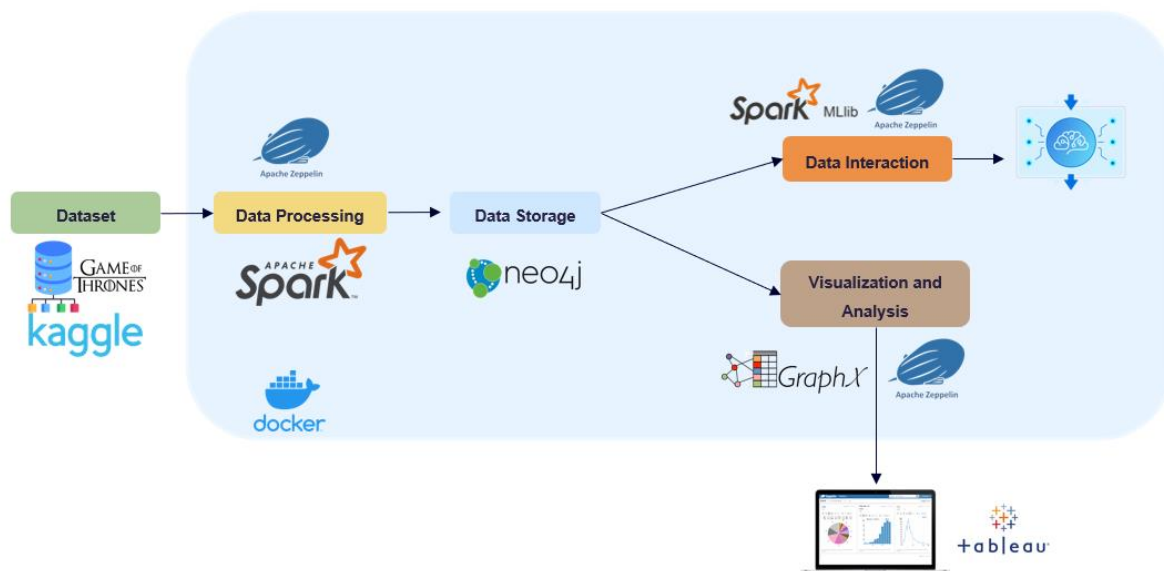


Figure 2: Architecture du projet

- **Collecte de Données depuis Kaggle :**

Les données sources sont collectées depuis Kaggle, où des ensembles de données liés à Game of Thrones sont disponibles. Ces données peuvent inclure des informations sur les personnages, les maisons, les relations, etc.

- **Chargement des Données avec Spark dans Zeppelin :**

Utilisation d'Apache Spark pour le chargement et la manipulation des données. Spark permet de traiter de grands ensembles de données de manière distribuée et efficace. Les

données sont chargées dans Apache Zeppelin, un environnement de notebook interactif.

- **Traitement des Données dans Zeppelin :**

Les données provenant de différents fichiers CSV sont traitées dans Apache Zeppelin. Les étapes de traitement incluent la fusion des fichiers pour rassembler les informations de manière cohérente, le nettoyage des données, et la transformation si nécessaire.

- **Stockage Direct dans Neo4j :**

Les données traitées sont directement stockées dans Neo4j, une base de données orientée graphe. Neo4j est idéal pour représenter et stocker des relations complexes, ce qui est essentiel pour modéliser le réseau de Game of Thrones.

- **Visualisation et Analyse avec GraphX :**

Utilisation d'Apache GraphX, un module de traitement de graphes de Spark, pour visualiser et analyser le réseau complexe de Game of Thrones. GraphX permet de déduire des motifs, des clusters et d'explorer les relations entre les différents éléments du réseau.

- **Affichage des Résultats d'Algorithmes dans Tableau :**

Les résultats obtenus à partir de l'analyse avec GraphX peuvent être visualisés de manière plus conviviale en utilisant Tableau. Cette étape permet de présenter les insights gagnés à un public plus large de manière intuitive.

- **Application de Modèles de Machine Learning à partir de Neo4j :**

À partir de la base de données Neo4j, des modèles de machine learning peuvent être appliqués pour tirer des conclusions prédictives. Cela pourrait inclure des prédictions basées sur les relations entre les personnages, les maisons, ou d'autres entités.

6 L'image Docker :

Cet image de configuration Docker Compose définit un ensemble de services pour Apache Zeppelin (0.10.0), Spark (3.1.2), et Neo4j (5) au sein d'un réseau Docker commun. Zeppelin est configuré pour se connecter au Spark Master, et un connecteur Neo4j-Spark est ajouté au répertoire des notebooks. Les services Spark et Neo4j sont également configurés avec les ports nécessaires pour l'interface utilisateur et la communication entre les conteneurs. Le réseau Docker personnalisé facilite la communication entre les services, tandis que les volumes Docker assurent la persistance des données de Neo4j et offrent une flexibilité pour le stockage des notebooks Zeppelin. En résumé, ce fichier permet de déployer un environnement intégré pour l'analyse de données avec Apache Zeppelin, Spark, et Neo4j.


```

1  version: '3'
2
3  services:
4    zeppelin:
5      image: apache/zeppelin:0.10.0
6      ports:
7        - "8080:8080"
8      environment:
9        - ZEPPELIN_SPARK_MASTER=spark://spark-master:7077
10       - ZEPPELIN_NOTEBOOK_DIR=/zeppelin/notebook
11       - ZEPPELIN_INTP_BASIC_AUTH=false
12      volumes:
13        - ./notebook:/zeppelin/notebook
14        - ./jar/neo4j-spark-connector-2.4.5-M2.jar:/zeppelin/notebook/neo4j-spark-connector-2.4.5-M2.jar
15      depends_on:
16        - spark-master
17      extra_hosts:
18        - "spark-master:0.0.0.0"
19      networks:
20        - my-network
21
22    spark-master:
23      image: bitnami/spark:3.1.2
24      ports:
25        - "7077:7077"
26      networks:
27        - my-network
28
29    neo4j:
30      image: docker.io/bitnami/neo4j:5
31      ports:
32        - "7474:7474"
33        - "7473:7473"
34        - "7687:7687"
35      volumes:
36        - 'neo4j_data:/bitnami'
37      networks:
38        - my-network
39
40  networks:
41    my-network:
42      driver: bridge
43  volumes:
44    neo4j_data:
45      driver: local
46      hadoop_namenode:
47      hadoop_datanode:
48      hadoop_historyserver:
49

```

Figure 3 docker-compose.yml file

<input type="checkbox"/>	<div><div></div><div></div><div></div></div>	<div>proj</div> <div>et_got</div>	Running (3/3)	2.49%		3 hours ago	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	<div><div></div><div></div><div></div></div>	<div>zeppelin-1</div> <div>c1bac0e12</div> <div>apache/zeppelin:0.1</div>	Running	1.44%	8080:8080	🔗	3 hours ago	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	<div><div></div><div></div><div></div></div>	<div>spark-ma</div> <div>38dda2179</div> <div>bitnami/spark:3.1.2</div>	Running	0.16%	7077:7077	🔗	3 hours ago	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	<div><div></div><div></div><div></div></div>	<div>neo4j-1</div> <div>d2cfb56653</div> <div>docker.io/bitnami/n</div>	Running	0.89%	7473:7473	🔗	Show all ports (3)	3 hours ago	<div><div></div><div></div><div></div></div>

Figure 4 le cluster docker

7 Importation des JARs :

Pour établir une connexion entre Neo4j et Spark dans Zeppelin, assurez-vous tout d'abord de la version de Spark installée dans Zeppelin. Vous pouvez vérifier cela en exécutant la commande suivante dans un paragraphe Zeppelin :

```
%spark
spark.version
```

```
%spark
sc.version

res18: String = 2.4.5
```

Figure 5 version de spark dans zeppelin

Pour utiliser le connecteur Neo4j-Spark, vous devrez télécharger le JAR correspondant à votre environnement. Voici comment vous pouvez télécharger le JAR depuis la page des releases GitHub :

<https://github.com/neo4j-contrib/neo4j-spark-connector/releases?page=3>

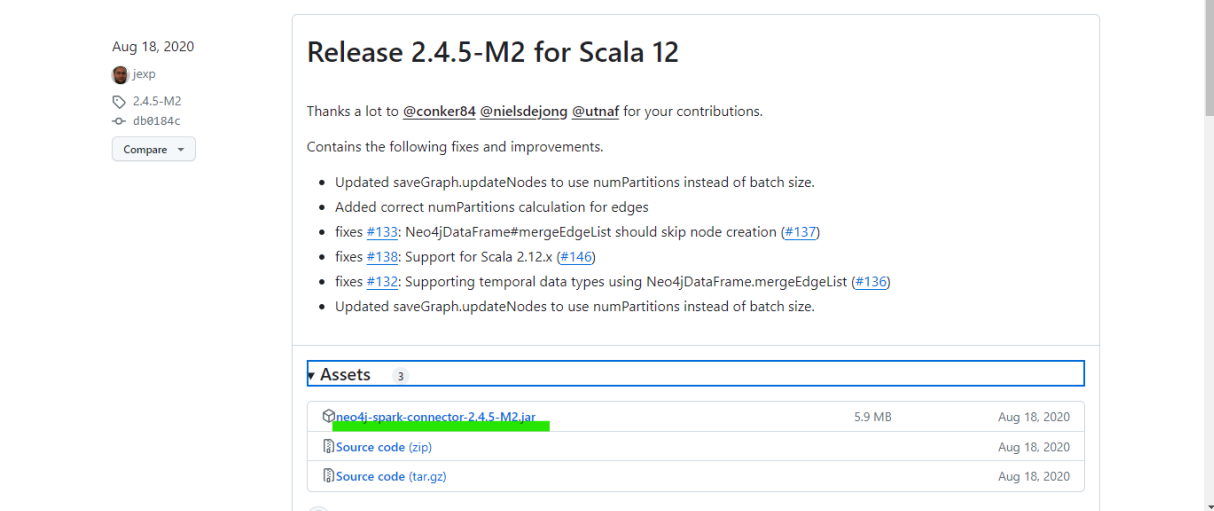


Figure 6 fichier JAR de connexion de neo4j et spark

Si vous souhaitez copier un fichier JAR depuis votre système local vers un conteneur Docker Zeppelin, vous pouvez utiliser la commande docker cp. Assurez-vous que le conteneur Zeppelin est en cours d'exécution.

docker cp chemin_local_du_jar container_id:/opt/zeppelin/interpreter/spark/

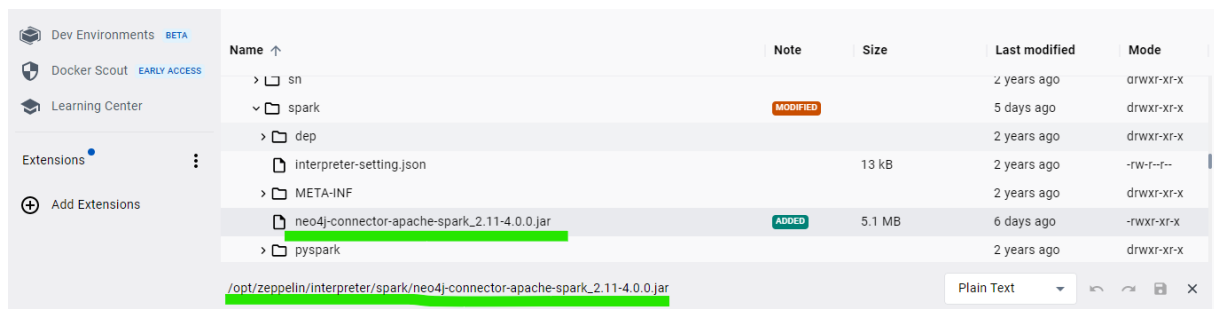


Figure 7 importation du jar dans zeppelin

On passe au chemin de configuration de Zeppelin et on ajoute la configuration du connecteur :

```
cd /opt/zeppelin/conf/
cat zeppelin-env.sh.template
```

On passe à ce chemin et on ajoute cette ligne :

```
# export ZEPPELIN_INTP_CLASSPATH_REMOTE=/opt/zeppelin/interpreter/spark/neo4j-connector-apache-spark_2.11-4.0.0.jar
```

Le JAR file sera present dans le repository de GitHub.

```

10
76 ## Use provided spark installation ##
77 ## defining SPARK_HOME makes Zeppelin run spark interpreter process using spark-submit
78 ##
79 # export SPARK_HOME # (required) When it is defined, load it instead of Zeppelin embedded Spark
80 # export SPARK_SUBMIT_OPTIONS # (optional) extra options to pass to spark submit. eg) "--driver-memory 51
81 # export SPARK_APP_NAME # (optional) The name of spark application.
82 # export SPARK_CONF_DIR # (optional) In the zeppelin interpreter on docker mode, Need to set the lo
83 # export ZEPPELIN_SPARK_USEHIVECONTEXT=true
84 # export ZEPPELIN_INTP_CLASSPATH_REMOTE=/opt/zeppelin/interpreter/spark/neo4j-connector-apache-spark_2.11-4.0.0.jar
85
86 ## Use embedded spark binaries ##

```

Figure 8 configuration du JAR

8 Manipulation & Insertion des données:

➤ Vous trouverez tous les notebooks nécessaires à utiliser dans ce dossier

Nom	Modifié le	Type	Taille
CRUD.	19/12/2023 18:54	Fichier source Jupyter	13 Ko
GraphX.	19/12/2023 21:33	Fichier source Jupyter	15 Ko
GraphXDep.	19/12/2023 18:55	Fichier source Jupyter	1 Ko
Manipulation&Insertion.	19/12/2023 18:55	Fichier source Jupyter	5 Ko
sparkML.	19/12/2023 21:30	Fichier source Jupyter	19 Ko
SparkMLDep.	19/12/2023 18:55	Fichier source Jupyter	1 Ko

Figure 9 les notebook du Projet

- Copiez les fichiers CSV vers le conteneur :

docker cp chemin_local_des_fichiers_csv container_id:/chemin_dans_conteneur/

- Accédez au conteneur Zeppelin en utilisant exec :

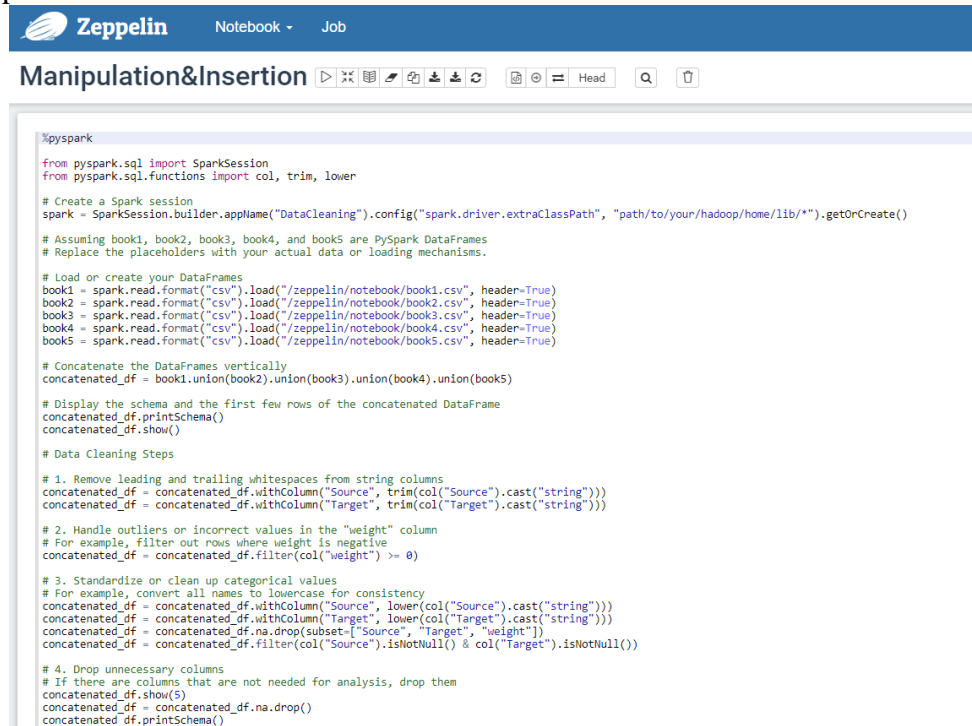
docker exec -it zeeplin_container_id bash

- Déplacez les fichiers CSV vers le répertoire approprié à l'intérieur du conteneur :

mv /chemin_dans_conteneur/fichiers_csv/* /zeppelin/notebook/

8.1 Récupération et Manipulation des données

- Puis concaténer l'ensemble des données (datasets) provenant des cinq livres et faire des manipulations :



```
%pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, trim, lower

# Create a Spark session
spark = SparkSession.builder.appName("DataCleaning").config("spark.driver.extraClassPath", "path/to/your/hadoop/home/lib/*").getOrCreate()

# Assuming book1, book2, book3, book4, and book5 are PySpark DataFrames
# Replace the placeholders with your actual data or loading mechanisms.

# Load or create your DataFrames
book1 = spark.read.format("csv").load("/zeppelin/notebook/book1.csv", header=True)
book2 = spark.read.format("csv").load("/zeppelin/notebook/book2.csv", header=True)
book3 = spark.read.format("csv").load("/zeppelin/notebook/book3.csv", header=True)
book4 = spark.read.format("csv").load("/zeppelin/notebook/book4.csv", header=True)
book5 = spark.read.format("csv").load("/zeppelin/notebook/book5.csv", header=True)

# Concatenate the DataFrames vertically
concatenated_df = book1.union(book2).union(book3).union(book4).union(book5)

# Display the schema and the first few rows of the concatenated DataFrame
concatenated_df.printSchema()
concatenated_df.show()

# Data Cleaning Steps

# 1. Remove leading and trailing whitespaces from string columns
concatenated_df = concatenated_df.withColumn("Source", trim(col("Source")).cast("string"))
concatenated_df = concatenated_df.withColumn("Target", trim(col("Target")).cast("string"))

# 2. Handle outliers or incorrect values in the "weight" column
# For example, filter out rows where weight is negative
concatenated_df = concatenated_df.filter(col("weight") >= 0)

# 3. Standardize or clean up categorical values
# For example, convert all names to lowercase for consistency
concatenated_df = concatenated_df.withColumn("Source", lower(col("Source")).cast("string"))
concatenated_df = concatenated_df.withColumn("Target", lower(col("Target")).cast("string"))
concatenated_df = concatenated_df.na.drop(subset=["Source", "Target", "weight"])
concatenated_df = concatenated_df.filter((col("Source").isNotNull() & col("Target").isNotNull()))

# 4. Drop unnecessary columns
# If there are columns that are not needed for analysis, drop them
concatenated_df.show(5)
concatenated_df = concatenated_df.na.drop()
concatenated_df.printSchema()
```

Figure 10 code de concatenation des fichier CSV

```
# 2. Handle outliers or incorrect values in the weight column
# For example, filter out rows where weight is negative
concatenated_df = concatenated_df.filter(col("weight") >= 0)

# 3. Standardize or clean up categorical values
# For example, convert all names to lowercase for consistency
concatenated_df = concatenated_df.withColumn("Source", lower(col("Source").cast("string")))
concatenated_df = concatenated_df.withColumn("Target", lower(col("Target").cast("string")))
concatenated_df = concatenated_df.na.drop(subset=["Source", "Target", "weight"])
concatenated_df = concatenated_df.filter(col("Source").isNotNull() & col("Target").isNotNull())

# 4. Drop unnecessary columns
# If there are columns that are not needed for analysis, drop them
concatenated_df.show(5)
concatenated_df = concatenated_df.na.drop()
concatenated_df.printSchema()

# Save the cleaned and concatenated DataFrame to a single CSV file
concatenated_df.coalesce(1).write.mode("overwrite").csv("/zeppelin/notebook/resultUnion.csv", header=True)
# Load the cleaned DataFrame
cleaned_df = spark.read.format("csv").load("/zeppelin/notebook/resultUnion.csv", header=True)
cleaned_df.show(10)
# Load The Cleaned DataFrame
```

root

```
-- Source: string (nullable = true)
-- Target: string (nullable = true)
-- Type: string (nullable = true)
-- weight: string (nullable = true)
-- book: string (nullable = true)
```

Source	Target	Type	weight	book
Addam-Marbrand	Jaime-Lannister	Undirected	3	1
Addam-Marbrand	Tywin-Lannister	Undirected	6	1
Aegon-I-Targaryen	Daenerys-Targaryen	Undirected	5	1
Aegon-I-Targaryen	Eddard-Stark	Undirected	4	1
Aemon-Targaryen-(...)	Alliser-Thorne	Undirected	4	1
Aemon-Targaryen-(...)	Bowen-Marsh	Undirected	4	1
Aemon-Targaryen-(...)	Chett	Undirected	9	1

Took 51 sec. Last updated by anonymous at December 18 2023, 7:09:27 PM.

Figure 11 stockage dans un nouveau fichier result.csv

8.2 Insertion des données

- Insertion de données dans Neo4j avec Spark :

```
import org.neo4j.driver._
import org.neo4j.driver.Values
import scala.io.Source

val uri = "bolt://neo4j:7687"
val user = "neo4j"
val password = "bitnam1"

val driver = GraphDatabase.driver(uri, AuthTokens.basic(user, password))
val session = driver.session()

// Use HDFS file path
val csvFile = "/zeppelin/notebook/resultUnion.csv/part-00000-fa647f61-612d-40ed-81b5-1c8049caa127-c000.csv"
val lines = Source.fromFile(csvFile).getLines().toList

// Extract header and data
val header = lines.head.split(",")
val data = lines.tail.map(_.split(","))

// Insert data into Neo4j with a different graph structure
data.foreach { row =>
  // Check if the row has enough elements
  if (row.length >= 5) {
    val cypherQuery =
      s"""
        MERGE (source:Node1 {name: '${row(0)}'})
        MERGE (target:Node1 {name: '${row(1)}'})
        MERGE (source)-[r:${row(2)} {weight: ${row(3).toInt}, book: ${row(4).toInt}}]->(target)
      """
    session.run(cypherQuery)
  } else {
    println(s"Skipping row: ${row.mkString(", ")} - Insufficient columns")
  }
}

// Close the session and driver when done
session.close()

import org.neo4j.driver._
import org.neo4j.driver.Values
import scala.io.Source

uri: String = bolt://neo4j:7687
user: String = neo4j
password: String = bitnam1
driver: org.neo4j.driver.Driver = org.neo4j.driver.internal.InternalDriver@60ffc672
session: org.neo4j.driver.Session = org.neo4j.driver.internal.InternalSession@272af2a8
```

Figure 12 insertion des données

8.3 Vérification dans Neo4j

- On peut visualiser les résultats stockés dans Neo4j en accédant à l'adresse <http://localhost:7474/browser/>
- Visualiser vos données dans le conteneur neo4j

```
MATCH (n:Node1) return n;
```



Figure 13 visualisation

9 CRUD :

9.1 Récupération d'après Neo4j

- Exécution de commande Neo4j dans zeppelin :
➤ Premièrement si vous n'avez pas Neo4j comme interpréteur dans Zeppelin il faut l'ajouter, et la configurer

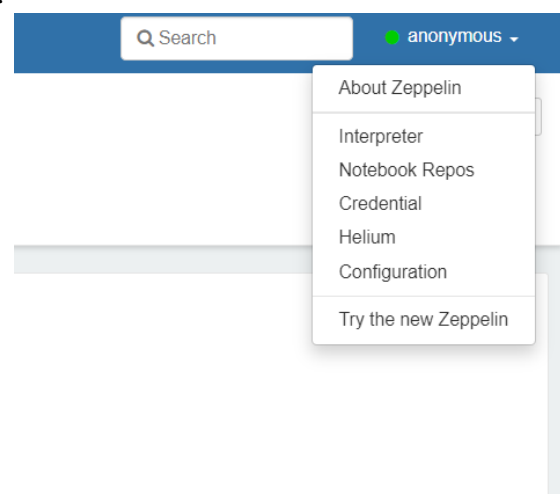


Figure 14 configuration de l'interpréteur Neo4j

neo4j neo4j

Option
The interpreter will be instantiated Globally in shared process

☐ Connect to existing process
☐ Set permission

Properties

Name	Value	Description	Action
neo4j.url	bolt://neo4j:7687	The Neo4j's BOLT url.	<input type="button" value="✕"/>
neo4j.database		The Neo4j target database, if empty use the default db.	<input type="button" value="✕"/>
neo4j.multi.statement	true	Enables the multi statement management, if true it computes multiple queries separated by semicolon.	<input type="button" value="✕"/>
neo4j.auth.type	BASIC	The Neo4j's authentication type (NONE, BASIC).	<input type="button" value="✕"/>
neo4j.auth.user	neo4j	The Neo4j user name.	<input type="button" value="✕"/>
neo4j.auth.password	btname1	The Neo4j user password.	<input type="button" value="✕"/>
neo4j.max.concurrency	50	Max concurrency call from Zeppelin to Neo4j server.	<input type="button" value="✕"/>

Figure 15 connecter l'interpreteur avec le contenuer neo4j

Zeppelin Notebook Job

CRUD

%neo4j
MATCH (n1:Node1)
RETURN n1;

id	label	name
2	Node1	addam-marbrand
3	Node1	jaimelannister
4	Node1	tywin-lannister
5	Node1	aegon-targaryen
6	Node1	daenerys-targaryen
7	Node1	edward-stark
8	Node1	aemon-targaryen-(maester-aemon)
9	Node1	alliser-thorne

Took 4 sec. Last updated by anonymous at December 18 2023, 7:18:50 PM.

Figure 16 visualisation dans zeppelin

En utilisant les requêtes CYPHER dans zeppelin avec l'interpreteur neo4j.

%neo4j

- Récupération des Nodes d'après neo4j :

```
%spark
import org.apache.spark.sql.{SaveMode, SparkSession}

val spark = SparkSession.builder().getOrCreate()

val df1 = (spark.read.format("org.neo4j.spark.DataSource")
  .option("url", "bolt://neo4j:7687")
  .option("authentication.basic.username", "neo4j")
  .option("authentication.basic.password", "bitnami1")
  .option("labels", "Node1")
  .load())

df1 <table>
```

	<id>	<labels>	name
2	[Node1]		addam-marbrand
3	[Node1]		jaime-lannister
4	[Node1]		tywin-lannister
5	[Node1]		aegon-i-targaryen
6	[Node1]		daenerys-targaryen
7	[Node1]		eddard-stark
8	[Node1]		aemon-targaryen(...
9	[Node1]		alliser-thorne
10	[Node1]		bowen-marsh
11	[Node1]		chett
12	[Node1]		clydas
13	[Node1]		jeor-mormont
14	[Node1]		jon-snow
15	[Node1]		samwell-tarly

Took 3 sec. Last updated by anonymous at December 18 2023, 7:20:01 PM.

Figure 17 lire des données de neo4j dans zeppelin pour les manipuler

- Création d'un graph :

```
// CREATION DU GRAPH

import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

// Extract vertices from DataFrame
val vertices: RDD[(VertexId, String)] = df1
  .select("name") // Replace "id" with the actual column name for vertex IDs
  .distinct()
  .rdd
  .map(row => (row.getLong(0), row.getString(1))) // Adjust column indices and types

// Extract edges from DataFrame
val edges: RDD[Edge[String]] = df1
  .select("name") // Replace with actual column names
  .rdd
  .map(row => Edge(row.getLong(0), row.getLong(1), row.getString(2))) // Adjust column indices and types

// Create the GraphX graph
val graph = Graph(vertices, edges)

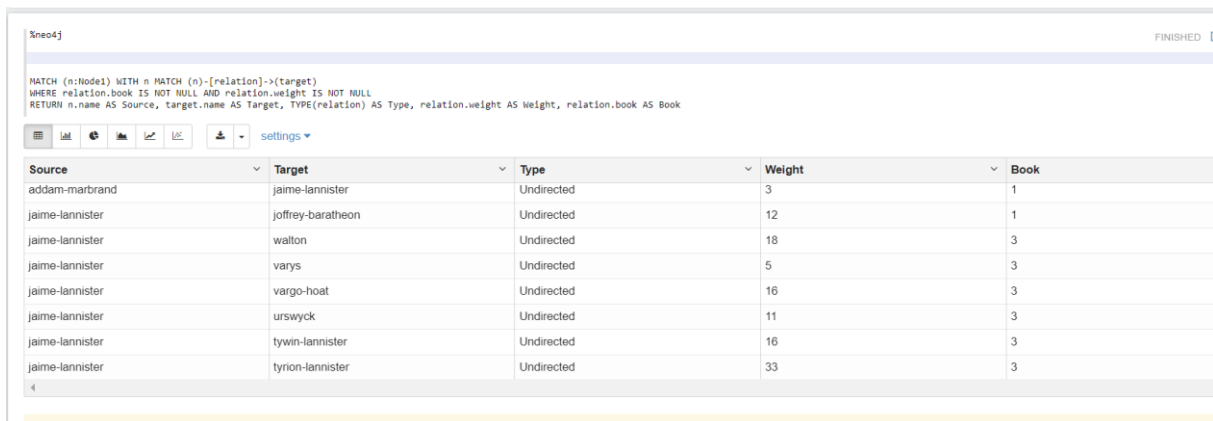
// Now you can apply GraphX algorithms on 'graph'

import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
vertices: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, String)] = MapPartitionsRDD[119] at map at <console>:41
edges: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] = MapPartitionsRDD[125] at map at <console>:47
graph: org.apache.spark.graphx.Graph[String,String] = org.apache.spark.graphx.impl.GraphImpl@69cf5c4e
```

Took 2 sec. Last updated by anonymous at December 18 2023, 7:20:06 PM.

Figure 18 créer un graph à partir des données

- Récupération des données d'après neo4j :



Neo4j

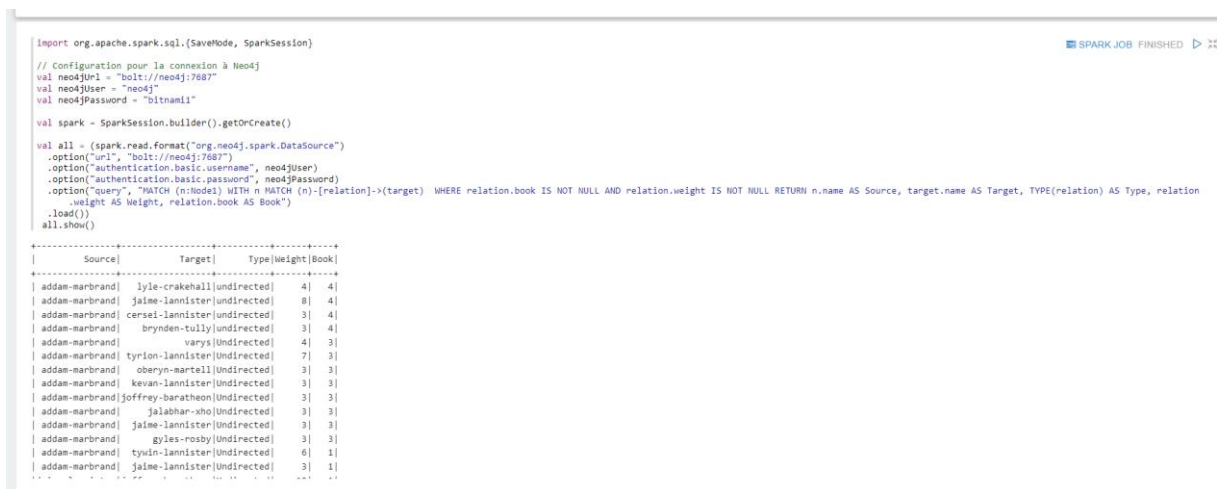
FINISHED

```
MATCH (n:Node1) WITH n MATCH (n)-[relation]->(target)
WHERE relation.book IS NOT NULL AND relation.weight IS NOT NULL
RETURN n.name AS Source, target.name AS Target, TYPE(relation) AS Type, relation.weight AS Weight, relation.book AS Book
```

Source	Target	Type	Weight	Book
addam-marbrand	jaimie-lannister	Undirected	3	1
jaimie-lannister	jooffrey-baratheon	Undirected	12	1
jaimie-lannister	walton	Undirected	18	3
jaimie-lannister	varys	Undirected	5	3
jaimie-lannister	vargo-hoat	Undirected	16	3
jaimie-lannister	urswyck	Undirected	11	3
jaimie-lannister	tywin-lannister	Undirected	16	3
jaimie-lannister	tyrion-lannister	Undirected	33	3

Figure 19 visualiser les données en utilisant les CYPHER query

On rapporte ici les différents nœud et les relation entre eux.



```
import org.apache.spark.sql.{SaveMode, SparkSession}

// Configuration pour la connexion à Neo4j
val neo4jUrl = "bolt://neo4j:7687"
val neo4jUser = "neo4j"
val neo4jPassword = "bitnamli"

val spark = SparkSession.builder().getOrCreate()

val all = (spark.read.format("org.neo4j.spark.DataSource")
  .option("url", "bolt://neo4j:7687")
  .option("authentication.basic.username", neo4jUser)
  .option("authentication.basic.password", neo4jPassword)
  .option("query", "MATCH (n:Node1) WITH n MATCH (n)-[relation]->(target) WHERE relation.book IS NOT NULL AND relation.weight IS NOT NULL RETURN n.name AS Source, target.name AS Target, TYPE(relation) AS Type, relation.weight AS Weight, relation.book AS Book")
  .load()
  .all.show())
```

Source	Target	Type	Weight	Book
addam-marbrand	lyle-craekhall	undirected	4	4
addam-marbrand	jaimie-lannister	undirected	8	4
addam-marbrand	cersei-lannister	undirected	3	4
addam-marbrand	brynden-tully	undirected	3	4
addam-marbrand	varys	undirected	4	3
addam-marbrand	tyrion-lannister	undirected	7	3
addam-marbrand	oberyn-martell	undirected	3	3
addam-marbrand	kevan-lannister	undirected	3	3
addam-marbrand	jooffrey-baratheon	undirected	3	3
addam-marbrand	jalabhar-xho	undirected	3	3
addam-marbrand	jaimie-lannister	undirected	3	3
addam-marbrand	gyles-rosby	undirected	3	3
addam-marbrand	tywin-lannister	undirected	6	1
addam-marbrand	jaimie-lannister	undirected	3	1

Figure 20 visualiser les données en utilisant SPARK

9.2 Exemples de filtrages

- Récupération des données du Book1:

```
// Filter sources with book = 1
val sourcesWithBook1 = all.filter("Book = 1").select("Source", "Book").distinct()

// Show the filtered sources
sourcesWithBook1.show()
```

Source	Book
addam-marbrand	1
jaime-lannister	1
tywin-lannister	1
aegon-i-targaryen	1
daenerys-targaryen	1
edward-stark	1
aemon-targaryen(...)	1
alliser-thorne	1
bowen-marsh	1
chett	1
jeor-mormont	1
jon-snow	1
aerys-ii-targaryen	1
brandon-stark	1

Took 2 sec. Last updated by anonymous at December 18 2023, 7:23:16 PM.

Figure 21 les nœuds du premier livre

- Récupération des données du Book2:

```
// Filter sources with book = 2
val sourcesWithBook1 = all.filter("Book = 2").select("Source", "Book").distinct()

// Show the filtered sources
sourcesWithBook1.show()
```

Source	Book
jaime-lannister	2
tywin-lannister	2
aegon-i-targaryen	2
daenerys-targaryen	2
edward-stark	2
aemon-targaryen(...)	2
alliser-thorne	2
chett	2
jeor-mormont	2
jon-snow	2
aerys-ii-targaryen	2
robert-baratheon	2
aggo	2
drogo	2

Took 1 sec. Last updated by anonymous at December 18 2023, 7:23:23 PM.

Figure 22 les nœuds du 2eme livre

- Récupération des données du Book3:

```
// Filter sources with book = 3
val sourcesWithBook1 = all.filter("Book = 3").select("Source", "Book").distinct()

// Show the filtered sources
sourcesWithBook1.show()
```

Source	Book
Addam-Marbrand	3
Aegon-Frey-(son-o-...)	3
Aegon-I-Targaryen	3
Aegon-Targaryen(...)	3
Aemon-Targaryen(...)	3
Aemon-Targaryen(...)	3
Aerys-II-Targaryen	3
Aggo	3
Alayaya	3
Alerie-Hightower	3
Alester-Florent	3
Alla-Tyrell	3
Alliser-Thorne	3
Amory-Lorch	3

Took 1 sec. Last updated by anonymous at December 13 2023, 9:29:28 PM.

Figure 23 les nœuds du 3eme livre

- Récupération des données du Book4:

```
// Filter sources with book = 1
val sourcesWithBook1 = all.filter("Book = 4").select("Source", "Book").distinct()

// Show the filtered sources
sourcesWithBook1.show()
```

Source	Book
Addam-Marbrand	4
Aegon-Targaryen-(...)	4
Aegon-V-Targaryen	4
Aemon-Targaryen-(...)	4
Aeron-Greyjoy	4
Aerys-II-Targaryen	4
Alla-Tyrell	4
Alleras	4
Alys-Arryn	4
Amerei-Frey	4
Anders-Yronwood	4
Andrey-Dalt	4
Anyia-Waywood	4
Areo-Hotah	4

Took 1 sec. Last updated by anonymous at December 13 2023, 9:29:44 PM.

Figure 24 les nœuds du 4eme livre

- Récupération des données du Book5:

```
// Filter sources with book = 1
val sourcesWithBook1 = all.filter("Book = 5").select("Source", "Book").distinct()

// Show the filtered sources
sourcesWithBook1.show()
```

Source	Book
Aegon-I-Targaryen	5
Aegon-Targaryen-(...)	5
Aemon-Targaryen-(...)	5
Aenys-Frey	5
Aeron-Greyjoy	5
Aerys-II-Targaryen	5
Aggo	5
Alliser-Thorne	5
Alys-Karstark	5
Alysane-Mormont	5
Archibald-Yronwood	5
Areo-Hotah	5
Arianne-Martell	5
Arnolf-Karstark	5

Took 2 sec. Last updated by anonymous at December 13 2023, 9:29:53 PM.

Figure 25 les nœuds du dernier livre

- Les acteurs qui existent dans tous les livres :

```
// Calculate the total number of books
val totalNumberOfBooks = all.select("Book").distinct().count()

// Group by Source and count distinct books
val sourceBookCounts = all.groupBy("Source").agg(countDistinct("Book").alias("DistinctBooks"))

// Filter sources that exist in all books
val sourcesInAllBooks = sourceBookCounts.filter(col("DistinctBooks") === lit(totalNumberOfBooks)).select("Source")

// Show the result
sourcesInAllBooks.show()
val numberOfSourcesInAllBooks = sourcesInAllBooks.count()
val numberOfAllSources = all.select("Source").distinct().count()
// Print the result
println(s"The number of actors that exist in all books is: $numberOfSourcesInAllBooks out of $numberOfAllSources")
```

Source
jaime-lannister
edward-stark
aemon-targaryen-...
jeor-mormont
jon-snow
aerys-ii-targaryen
robert-baratheon
grrinn
tyrion-lannister
arya-stark
bran-stark
catelyn-stark
cersei-lannister
ilyn-payne

Took 4 sec. Last updated by anonymous at December 18 2023, 7:23:59 PM.

Figure 26 les personnages qui sont dans tous les livres

- La personne la plus présente dans la base de données :

```
import org.apache.spark.sql.functions._

// Group by Target and count occurrences
val targetOccurrences = all.groupBy("Target").agg(count("*").alias("Occurrences"))

// Find the target with the most occurrences
val mostFrequentTarget = targetOccurrences.orderBy(desc("Occurrences")).select("Target").first().getString(0)

// Print the result
println(s"The target with the most occurrences is: $mostFrequentTarget")
```

The target with the most occurrences is: tyrion-lannister

```
import org.apache.spark.sql.functions._
targetOccurrences: org.apache.spark.sql.DataFrame = [Target: string, Occurrences: bigint]
mostFrequentTarget: String = tyrion-lannister
```

Took 2 sec. Last updated by anonymous at December 18 2023, 7:24:32 PM.

Figure 27 la personne la plus présente dans la série

9.3 ADD

- Ajoute des données:

```
import org.apache.spark.sql.{SaveMode, SparkSession}
import org.apache.spark.sql.functions._

val spark = SparkSession.builder().getOrCreate()
import spark.implicits._

// Sample data
val data = Seq(
  ("Reda-targarian", "Jaime-Lannister", "Undirected", 3, 1),
  ("oussama-targarian", "Tywin-Lannister", "Undirected", 6, 1),
  ("elhadi-targarian", "Tywin-Lannister", "Undirected", 6, 1)
)

// Define the schema for the DataFrame
val schema = Seq("Source", "Target", "Type", "Weight", "Book")

// Create a DataFrame
val df = spark.createDataFrame(data).toDF(schema: _*)
println(s"Number of entities before insertion: ${all.count()}")
// Save DataFrame to Neo4j
df.write
  .format("org.neo4j.spark.DataSource")
  .mode(SaveMode.ErrorIfExists)
  .option("url", "bolt://neo4j:7687")
  .option("authentication.basic.username", "neo4j")
  .option("authentication.basic.password", "bitnami")
  .option("labels", ":Node1")
  .option("source.labels", ":Node1")
  .option("source.save.mode", "MATCH")
  .option("target.labels", ":Node1")
  .option("target.save.mode", "MATCH")

Number of entities before insertion: 796
import org.apache.spark.sql.{SaveMode, SparkSession}
import org.apache.spark.sql.functions._
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@2131953a
import spark.implicits._
data: Seq[(String, String, String, Int, Int)] = List((Reda-targarian,Jaime-Lannister,Undirected,3,1), (oussama-targarian,Tywin-Lannister,Undirected,6,1)
schema: Seq[String] = List(Source, Target, Type, Weight, Book)
df: org.apache.spark.sql.DataFrame = [Source: string, Target: string ... 3 more fields]
```

Figure 28 ajouter un noeud

- Vérification:

```
import org.apache.spark.sql.{SparkSession, SaveMode}

val spark = SparkSession.builder().getOrCreate()

// Read the data from Neo4j
val neo4jData = spark.read.format("org.neo4j.spark.DataSource")
  .option("url", "bolt://neo4j:7687")
  .option("authentication.basic.username", "neo4j")
  .option("authentication.basic.password", "bitnami")
  .option("labels", ":Node1")
  .load()

// Compare with the original DataFrame
println("Comparison with original data:")
df.show()

// Perform any necessary assertions or checks to verify the correctness
// For example, you can compare the number of rows, schema, etc.
if (neo4jData.count() == df.count()) {
  println("Data was successfully inserted into Neo4j.")
} else {
  println("Data insertion into Neo4j might have issues.")
}

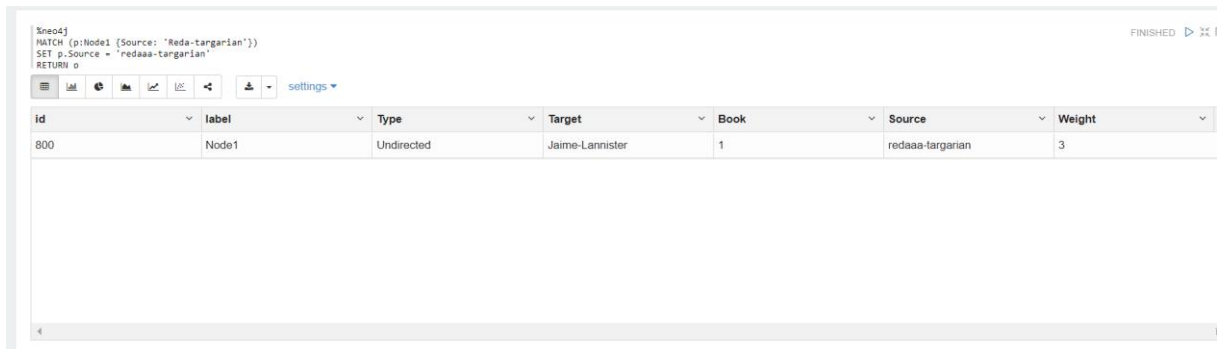
Comparison with original data:
+-----+-----+-----+-----+-----+
| Source | Target | Type | Weight | Book |
+-----+-----+-----+-----+-----+
| Reda-targarian | Jaime-Lannister | Undirected | 3 | 1 |
| oussama-targarian | Tywin-Lannister | Undirected | 6 | 1 |
| elhadi-targarian | Tywin-Lannister | Undirected | 6 | 1 |
+-----+-----+-----+-----+-----+

Data insertion into Neo4j might have issues.
import org.apache.spark.sql.{SparkSession, SaveMode}
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@2131953a
neo4jData: org.apache.spark.sql.DataFrame = [id: bigint, labels: array<string> ... 1 more field]
```

Figure 29 verifier l'ajout

9.4 EDIT

- Edit:



```

Kneo4j
MATCH (p:Node1 {Source: "Reda-targarian"})
SET p.Source = "redaaa-targarian"
RETURN p
  
```

id	label	Type	Target	Book	Source	Weight
800	Node1	Undirected	Jaime-Lannister	1	redaaa-targarian	3

Figure 30 modification des données

- vérification :

```

// Your existing code to create the 'all' DataFrame
val all = (spark.read.format("org.neo4j.spark.DataSource")
  .option("url", "bolt://neo4j:7687")
  .option("authentication.basic.username", "neo4j")
  .option("authentication.basic.password", "bitnami1")
  .option("labels", "Node1")
  .load())

// Retrieve one object (for example, the first row)
val targetRow: Row = all.collect()(0)

// Modify the object (for example, update the "Weight" field)
val modifiedRow = Row.fromSeq(targetRow.toSeq.updated(all.schema.fieldIndex("name"), "reda-targarian"))

// Create a new schema with the same structure as 'all'
val modifiedSchema = StructType(all.schema.fields)

// Create a DataFrame with the modified row
val modifiedDataFrame = spark.createDataFrame(spark.sparkContext.parallelize(Seq(modifiedRow)), modifiedSchema)

// Display the modified row
println("Modified Row:")
modifiedDataFrame.show()
// Display the number of entities before insertion
println(s"Number of entities before insertion: ${all.count()}")

Modified Row:
+-----+-----+
|<id>|<labels>|      name|
+-----+-----+
|  2| [Node1]|reda-targarian|
+-----+-----+

Number of entities before insertion: 796
import org.apache.spark.sql.{SparkSession, Row}
import org.apache.spark.sql.types.StructType
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@2131953a
all: org.apache.spark.sql.DataFrame = [<id>: bigint, <labels>: array<string> ... 1 more field]
targetRow: org.apache.spark.sql.Row = [2,WrappedArray(Node1),addam-marbrand]
modifiedRow: org.apache.spark.sql.Row = [2,WrappedArray(Node1),reda-targarian]
modifiedSchema: org.apache.spark.sql.types.StructType = StructType(StructField(<id>,LongType,false), StructField(<labels>,ArrayType(StringType,true),true), Str
modifiedDataFrame: org.a...
  
```

Figure 31 vérification de la modification

9.5 DELETE

- Delete:

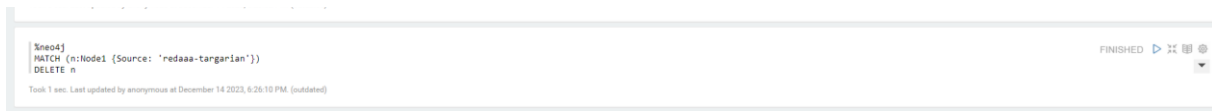


Figure 32 la suppression des données

10 GraphX :

10.1 Récupération des données et création du graph

- Récupération des données :

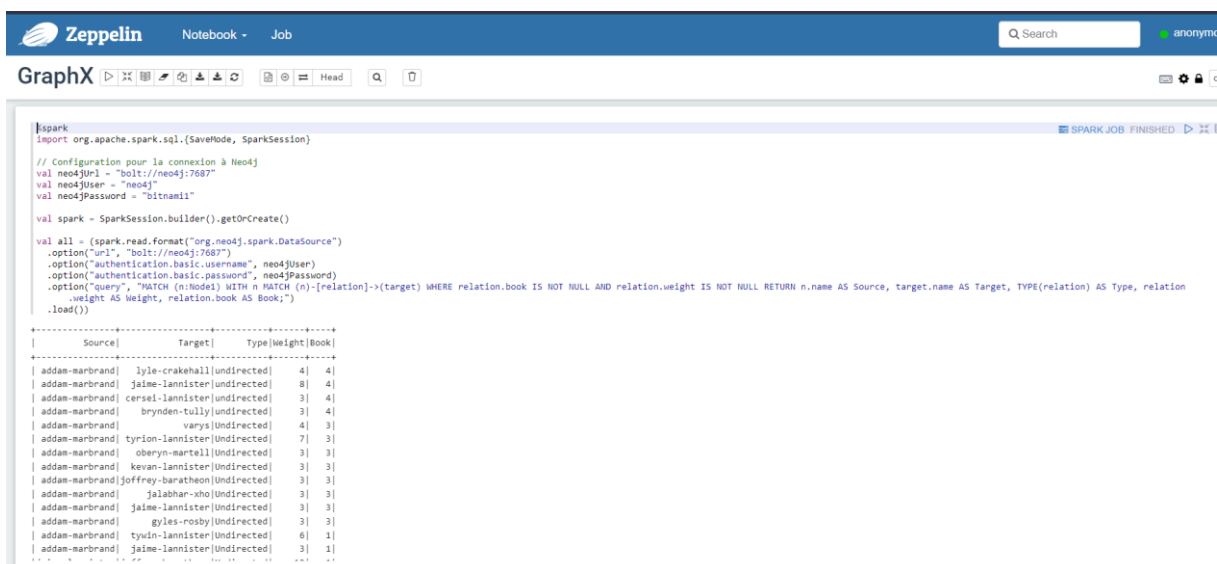


Figure 33 Récupération des données

- Création du Graph en utilisant GraphX :

```
//CREATION DU GRAPH

import org.apache.spark.sql.Row
import org.apache.spark.graphx.{Graph, Edge, VertexId}
import org.apache.spark.rdd.RDD

// Let's create the vertex RDD.
val vertices: RDD[(VertexId, String)] = all
  .selectExpr("explode(array(Target, Source)) as vertex")
  .distinct
  .rdd
  .map(_._1)
  .zipWithIndex
  .map(_._2)

// Now let's define a vertex dataframe because joins are clearer in Spark SQL
val vertexDf = vertices.toDF("id", "node")

// And let's extract the edges and join their vertices with their respective IDs
val edges: RDD[Edge[(String, Long, Long)]] = all
  .join(vertexDf, all("Source") === vertexDf("node"))
  .select(all("Type"), all("Target"), all("Weight").alias("Weight"), all("Book").alias("Book"), vertexDf("id").alias("ids"))
  .join(vertexDf, all("Target") === vertexDf("node"))
  .select("ids", "id", "Type", "Weight", "Book")
  .rdd
  .map { row =>
    Edge(
      row.getAs[Long]("ids"),
      row.getAs[Long]("id"),
      (row.getAs[String]("Type"), row.getAs[Long]("Weight"), row.getAs[Long]("Book"))
    )
  }

// And finally
import org.apache.spark.sql.Row
import org.apache.spark.graphx.{Graph, Edge, VertexId}
import org.apache.spark.rdd.RDD
vertices: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, String)] = MapPartitionsRDD[234] at map at <console>:77
vertexDf: org.apache.spark.sql.DataFrame = [id: bigint, node: string]
edges: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[(String, Long, Long)]] = MapPartitionsRDD[256] at map at <console>:89
```

Figure 34 créer un Graphe à l'aide de graphX pour les données récupérés

- Affichage des nodes et relations (vertices et Edges):

```
//AFFICHAGE DES NODES ET RELATIONS

// Display the first 10 vertices
println("First 10 Vertices:")
graph.vertices.take(10).foreach(println)

// Display the first 10 edges
println("First 10 Edges:")
graph.edges.take(10).foreach(println)
```

```
First 10 Vertices:
(600,())
(200,())
(0,())
(400,())
(401,())
(601,())
(201,())
(1,())
(602,())
(202,())
First 10 Edges:
Edge(275,343,(Undirected,3,1))
Edge(486,489,(Undirected,3,2))
Edge(3,375,(undirected,5,4))
Edge(2,32,(Undirected,24,3))
Edge(2,32,(Undirected,3,1))
Edge(2,32,(Undirected,3,1))
Edge(2,32,(Undirected,3,1))
Edge(2,32,(Undirected,3,1))
Edge(2,32,(Undirected,3,1))
Edge(2,32,(Undirected,3,1))
```

Took 10 sec. Last updated by anonymous at December 18 2023, 7:29:52 PM. (outdated)

Figure 35 Affichage des (vertices et Edges)

- Analyse statistique du Graph en utilisant GraphX :

```
//MANIPULATION DU GRAPH GRAPHX

// Nombre de nœuds et d'arêtes
val numVertices = graph.numVertices
val numEdges = graph.numEdges

// Degré moyen, degré maximum et minimum des nœuds
val avgDegree = graph.degrees.map(_._2).mean()
val maxDegree = graph.degrees.map(_._2).max()
val minDegree = graph.degrees.map(_._2).min()

// Statistiques des poids et des livres
val weightStats = graph.edges.map(_._attr._2).stats()
val bookStats = graph.edges.map(_._attr._3).stats()

// Affichage des résultats
println(s"Nombre de nœuds: $numVertices")
println(s"Nombre d'arêtes: $numEdges")
println(s"Degré moyen: $avgDegree")
println(s"Degré maximum: $maxDegree")
println(s"Degré minimum: $minDegree")

println("Statistiques des poids:")
println(weightStats)

println("Statistiques des livres:")
println(bookStats)

Nombre de nœuds: 796
Nombre d'arêtes: 3908
Degré moyen: 9.819095477386936
Degré maximum: 217
Degré minimum: 1
Statistiques des poids:
(count: 3908, mean: 8.348516, stdev: 11.147510, max: 291.000000, min: 3.000000)
Statistiques des livres:
(count: 3908, mean: 3.015353, stdev: 1.360267, max: 5.000000, min: 1.000000)
numVertices: Long = 796
numEdges: Long = 3908
avgDegree: Double = 9.819095477386936
maxDegree: Int = 217
minDegree: Int = 1
weightStats: org.apache.spark.util.StatCounter = (count: 3908, mean: 8.348516, stdev: 11.147510, max: 291.000000, min: 3.000000)
bookStats: org.apache.spark.util.StatCounter = (count: 3908, mean: 3.015353, stdev: 1.360267, max: 5.000000, min: 1.000000)
```

Figure 36 des statistiques sur le graph

```
val degreeDistribution = graph.degrees.map{ case (vertexId, degree) => (degree, vertexId) }.countByValue()

// Display degree distribution with names in descending order
println("Top 20 Degree Distribution with Names (Descending Order):")
degreeDistribution.toSeq.sortBy { case ((degree, vertexId), count) => -degree }.take(20).foreach { case ((degree, vertexId), count) =>
  val vertexName = vertices.lookup(vertexId).headOption.getOrElse("UnknownVertex")
  println(s"Vertex $vertexName, Degree $degree: $count nodes")
}

Top 20 Degree Distribution with Names (Descending Order):
Vertex Jon-Snow, Degree 438: 1 nodes
Vertex Daenerys-Targaryen, Degree 353: 1 nodes
Vertex Tyrion-Lannister, Degree 349: 1 nodes
Vertex Stannis-Baratheon, Degree 335: 1 nodes
Vertex Cersei-Lannister, Degree 311: 1 nodes
Vertex Theon-Greyjoy, Degree 219: 1 nodes
Vertex Jaime-Lannister, Degree 209: 1 nodes
Vertex Robert-Baratheon, Degree 190: 1 nodes
Vertex Eddard-Stark, Degree 182: 1 nodes
Vertex Arya-Stark, Degree 180: 1 nodes
Vertex Robb-Stark, Degree 176: 1 nodes
Vertex Joffrey-Baratheon, Degree 155: 1 nodes
Vertex Barristan-Selmy, Degree 151: 1 nodes
Vertex Sansa-Stark, Degree 149: 1 nodes
Vertex Bran-Stark, Degree 143: 1 nodes
Vertex Catelyn-Stark, Degree 130: 1 nodes
Vertex Tully-Lannister, Degree 120: 1 nodes

Took 8 sec. Last updated by anonymous at December 15 2023, 11:02:09 PM.
```

Figure 37 afficher les degrés des noeuds

Maintenant on applique les algorithmes sur GraphX :

10.2 PageRank Algorithm

Un algorithme d'analyse de liens utilisé pour mesurer l'importance relative des nœuds dans un graphe orienté ou non orienté. Il attribue des scores aux nœuds en fonction de la structure du graphe, en tenant compte du nombre et de la qualité des liens.

```
import org.apache.spark.sql.types.{StructType, StructField, DoubleType, StringType}

// Run PageRank
val pagerankGraph = PageRank.run(graph, numIter = 10)

val schema = StructType(Seq(
  StructField("vertexname", StringType, nullable = false),
  StructField("PageRank", DoubleType, nullable = false)
))

// Create an empty DataFrame with the defined schema
var pagerankDF = spark.createDataFrame(spark.sparkContext.emptyRDD[Row], schema)

// Get the vertices with their PageRank scores
val pageranks = RDD[(vertexID, Double)] = pagerankGraph.vertices

// Display the top 10 vertices with their PageRank scores in descending order
println("Top 10 Vertices with PageRank Scores (Descending Order):")
pageranks.sortBy(_._2, ascending = false).take(10).foreach { case (vertexID, pagerank) =>
  val vertexname = vertices.lookup(vertexID).headOption.getOrElse("UnknownVertex")
  println(s"Vertex $vertexname, PageRank: $pagerank")
}

// Append data to the DataFrame
val newData = Seq((vertexname, pagerank))
val newRow = spark.createDataFrame(newData).toDF("vertexname", "PageRank")
pagerankDF = pagerankDF.union(newRow)
}

println("-----")
pagerankDF.show()
```

Figure 38 PageRank Algorithm

```
Top 10 Vertices with PageRank Scores (Descending Order):
Vertex tyrion-lannister, PageRank: 33.82024901695369
Vertex stannis-baratheon, PageRank: 21.518088168274776
Vertex tywin-lannister, PageRank: 20.934834220853137
Vertex varys, PageRank: 18.44609375631678
Vertex theon-greyjoy, PageRank: 13.114445571816548
Vertex sansa-stark, PageRank: 11.810659996975042
Vertex walder-frey, PageRank: 9.524869698684245
Vertex robb-stark, PageRank: 8.744489665371315
Vertex samwell-tarly, PageRank: 8.470270331248827
Vertex tommen-baratheon, PageRank: 8.339092035339142
```

Figure 39 resultat de l'algorithme

Top 10 Sommets : Les 10 premiers sommets affichés représentent les nœuds les plus influents du graphe, selon l'algorithme PageRank. Ces sommets ont une probabilité plus élevée d'être visités lors d'un parcours aléatoire du graphe.

Distribution de l'Influence : Les scores PageRank fournissent une mesure de la distribution de l'influence à travers le réseau. Les sommets en haut de la liste ont une influence significative sur d'autres sommets du graphe.

Identification de Points Centraux : Les sommets en haut de la liste, tels que "Tyrion-Lannister" avec un PageRank de 33.82, peuvent être considérés comme des points centraux dans le réseau, indiquant peut-être leur rôle central ou leur importance dans le contexte du graphe.

10.3 Connected Components Algorithm

Un algorithme qui identifie les composantes connexes dans un graphe non orienté. Les composantes connexes sont des sous-ensembles de nœuds dans lesquels chaque nœud est relié à tous les autres par un chemin.

```
import org.apache.spark.graphx.{Graph, VertexId}
import org.apache.spark.graphx.lib.ConnectedComponents
import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}

// Assuming you already have 'spark' as your SparkSession

// Define the schema for connected components DataFrame
val schema = StructType(Seq(
  StructField("VertexName", StringType, nullable = false),
  StructField("ConnectedComponent", LongType, nullable = false)
))

// Create an empty DataFrame with the defined schema
var connectedComponentsDF = spark.createDataFrame(spark.sparkContext.emptyRDD[Row], schema)

// Run ConnectedComponents
val connectedComponentsGraph = ConnectedComponents.run(graph)

// Get the vertices with their connected component identifiers
val components = connectedComponentsGraph.vertices.collect()

// Display the connected component of each vertex and create DataFrame
println("Connected Components:")
components.foreach { case (vertexId, componentId) =>
  val vertexName = vertices.lookup(vertexId).headOption.getOrElse("UnknownVertex")
  println(s"Vertex $vertexName, Connected Component: $componentId")
  // Append data to the DataFrame
  val newData = Seq((vertexName, componentId))
  val newRow = spark.createDataFrame(newData).toDF("VertexName", "ConnectedComponent")
  connectedComponentsDF = connectedComponentsDF.union(newRow)
}

println("-----")
Connected Components:
Vertex softfoot, Connected Component: 0
Vertex zeii, Connected Component: 0
Vertex lyle-crakehall, Connected Component: 0
Vertex balon-greyjoy, Connected Component: 0
Vertex euron-greyjoy, Connected Component: 0
Vertex khal, Connected Component: 0
```

Figure 40 Connected Components Algorithm

Le fait que tous les sommets appartiennent au composant connecté "0" suggère qu'ils sont tous connectés les uns aux autres dans le graphe, directement ou indirectement, formant ainsi un seul ensemble ou une seule composante connectée.

10.4 Label Propagation Algorithm

Un algorithme itératif où les nœuds d'un graphe se voient attribuer des étiquettes en fonction de l'étiquette majoritaire de leurs voisins. Cela se répète jusqu'à ce que chaque nœud ait une étiquette qui est la plus fréquente parmi ses voisins.

```
//3.Label Propagation

import org.apache.spark.graphx.{Graph, VertexId}
import org.apache.spark.graphx.lib.LabelPropagation

val schema = StructType(Seq(
  StructField("VertexName", StringType, nullable = false),
  StructField("communityLabel", DoubleType, nullable = false)
))

// Create an empty DataFrame with the defined schema
var communityDF = spark.createDataFrame(spark.sparkContext.emptyRDD[Row], schema)

// Run Label Propagation with a reasonable number of maxSteps
val maxSteps = 10
val labeledGraph = LabelPropagation.run(graph, maxSteps)

// Get the vertices with their final community labels
val communities = labeledGraph.vertices

// Display the top 20 vertices with their community labels
println("Top 20 Vertices with Community Labels:")
communities.sortBy(_._2, ascending = false).take(20).foreach { case (vertexId, communityLabel) =>
  val vertexName = vertices.lookup(vertexId).headOption.getOrElse("UnknownVertex")
  println(s"Vertex $vertexName, Community Label: $communityLabel")
  // Append data to the DataFrame
  val newData = Seq((vertexName, communityLabel))
  val newRow = spark.createDataFrame(newData).toDF("VertexName", "communityLabel")
  communityDF = communityDF.union(newRow)
}

println("-----")

Top 20 Vertices with Community Labels:
Vertex Raymun-Redbeard, Community Label: 754
Vertex Gerrick-Kingsblood, Community Label: 753
Vertex Torghen-Flint, Community Label: 708
Vertex Brandon-Norrey, Community Label: 707
Vertex Justin-Massey, Community Label: 658
Vertex Alysane-Mormont, Community Label: 657
Vertex Richard-Horpe, Community Label: 657
Vertex Shortear, Community Label: 637
Vertex Puckens, Community Label: 636
Vertex Walgrave, Community Label: 634
Vertex Quill, Community Label: 634
Vertex Rosey, Community Label: 634
Vertex Ralf-Stanhovica, Community Label: 632
```

Figure 41 label propagation algorithm

Diversité des Communautés : Les sommets affichés avec leurs étiquettes de communauté représentent une variété de personnages issus de différentes régions ou factions dans le contexte du graphe. Par exemple, "Raymun-Redbeard" et "Gerrick-Kingsblood" appartiennent à des communautés distinctes (étiquettes 754 et 753).

Top 20 Vertices : Les 20 premiers sommets affichés avec leurs étiquettes de communauté offrent un aperçu des différentes communautés identifiées par l'algorithme.

10.5 Triangle Count Algorithm

Un algorithme qui compte le nombre de triangles dans un graphe. Un triangle est une structure composée de trois nœuds interconnectés, et le comptage de triangles est utile pour comprendre la connectivité et la structure locale d'un graphe.

```
//4. TRIANGLECOUNT

import org.apache.spark.graphx._
import org.apache.spark.graphx.lib.TriangleCount
import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.sql.types.{StructType, StructField, StringType, DoubleType}

// Assuming 'graph' is your original graph

// Define the schema for the triangle count DataFrame
val schema = StructType(Seq(
  StructField("VertexName", StringType, nullable = false),
  StructField("triangleCount", DoubleType, nullable = false)
))

// Create an empty DataFrame with the defined schema
var triangleCountDF = spark.createDataFrame(spark.sparkContext.emptyRDD[Row], schema)

// Run TriangleCount algorithm on the graph
val triangleCountGraph: Graph[Int, (String, Long, Long)] = TriangleCount.run(graph)

// Get the vertices with their triangle counts
val triangleCounts = triangleCountGraph.vertices

// Display the top 20 vertices with their triangle counts
println("Top 20 Vertices with Triangle Counts:")
triangleCounts.sortBy(_._2, ascending = false).take(20).foreach { case (vertexId, triangleCount) =>
  val vertexName = vertices.lookup(vertexId).headOption.getOrElse("UnknownVertex")
  println(s"Vertex $vertexName, Triangle Count: $triangleCount")
  // Append data to the DataFrame
  val newData = Seq((vertexName, triangleCount.toDouble))
  val newRow = spark.createDataFrame(newData).toDF("VertexName", "triangleCount")
  triangleCountDF = triangleCountDF.union(newRow)
}

println("-----")

Top 20 Vertices with Triangle Counts:
Vertex tyrion-lannister, Triangle Count: 659
Vertex Tyrion-Lannister, Triangle Count: 659
Vertex cersei-lannister, Triangle Count: 613
Vertex Cersei-Lannister, Triangle Count: 613
Vertex Jaime-Lannister, Triangle Count: 548
Vertex jaime-lannister, Triangle Count: 548
Vertex joffrey-baratheon, Triangle Count: 515
Vertex Joffrey Baratheon, Triangle Count: 515
```

Figure 42 triangle count algorithm

Sommet le plus Central : Le sommet "tyrion-lannister" est celui qui participe au plus grand nombre de triangles, avec un triangleCount de 659. Cela suggère que Tyrion Lannister est fortement central dans des structures triangulaires du réseau.

Interconnexion entre Personnages : Les sommets les plus élevés dans la liste représentent des personnages qui sont fortement interconnectés avec d'autres personnages du réseau. Par exemple, les membres de la famille Lannister (tyrion-lannister, cersei-lannister, jaime-lannister) et d'autres personnages clés comme sansa-stark, jon-snow, et robert-baratheon sont bien représentés.

Analyse des Communautés : Des groupes de sommets interconnectés peuvent indiquer des communautés ou des groupes fortement liés entre eux. Les membres d'une famille ou les alliés politiques peuvent former des sous-groupes avec une participation triangulaire significative.

10.6 Strongly Connected Algorithm

Un algorithme similaire à celui des composantes connexes, mais appliqué aux graphes orientés. Il identifie des sous-graphes dans lesquels chaque nœud est accessible à partir de tous les autres, formant ainsi des composantes fortement connexes.

```
import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.sql.types.{StructType, StructField, StringType, LongType}

// Assuming 'graph' is your original graph

// Run stronglyConnectedComponents algorithm
val stronglyConnectedComponents: Graph[VertexId, (String, Long, Long)] = graph.stronglyConnectedComponents(5)

// Define the schema for the strongly connected components DataFrame
val schema = StructType(Seq(
  StructField("VertexName", StringType, nullable = false),
  StructField("componentId", LongType, nullable = false),
  StructField("componentName", StringType, nullable = false)
))

// Convert 'vertices' RDD to a DataFrame
val verticesDF = vertices.toDF("VertexId", "VertexName")

// Get the vertices with their component IDs
val componentVertices = stronglyConnectedComponents.vertices.collect()

// Get the mapping between VertexId and VertexName from the 'vertices' DataFrame
val vertexMapping = verticesDF.select("VertexId", "VertexName").as[(Long, String)].collect.toMap

// Create a sequence of rows for DataFrame creation
val rows = componentVertices.map { case (vertexId, componentId) =>
  val vertexName = vertexMapping.getOrElse(vertexId, "UnknownVertex")
  val componentName = vertexMapping.getOrElse(componentId, "UnknownComponent")
  Row(vertexName, componentId, componentName)
}

// Create the DataFrame once using the sequence of rows
val sccDF = spark.createDataFrame(spark.sparkContext.parallelize(rows), schema)

println("Vertices with Strongly Connected Components:")
sccDF.show()
```

Figure 43 strongly connected algorithm

```
Vertices with Strongly Connected Components:
+-----+-----+-----+
| VertexName | componentId | componentName |
+-----+-----+-----+
| softfoot | 600 | softfoot |
| zeil | 200 | zeil |
| lyle-crakehall | 0 | lyle-crakehall |
| balon-greyjoy | 400 | balon-greyjoy |
| euron-greyjoy | 401 | euron-greyjoy |
| kyra | 601 | kyra |
| ygritte | 201 | ygritte |
| addam-marbrand | 1 | addam-marbrand |
| lorent-caswell | 602 | lorent-caswell |
| varamyr | 202 | varamyr |
| dagmer | 402 | dagmer |
| jaime-lannister | 2 | jaime-lannister |
| balon-swann | 403 | balon-swann |
+-----+-----+-----+
```

Figure 44 resultat de l'algorithme

11 Visualisation des résultats GraphX via tableau :

Maintenant on visualiser les résultats des algorithmes sur GraphX :

- PageRank Algorithm (Algorithme PageRank) :

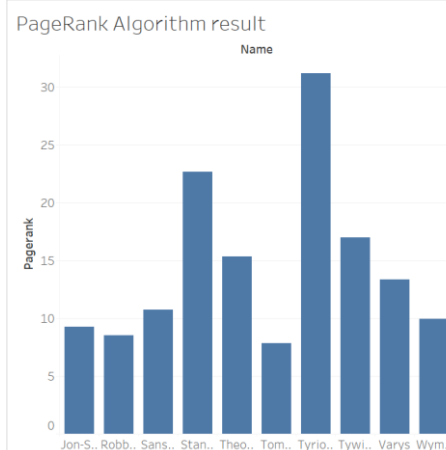


Figure 45 Page Rank algorithm

- Connected Components (Composantes Connexes) :

Connected Components Algorithm result

Name1	
addam-marbrand	0
aegon-frey-(son-...	0
aegon-i-targaryen	0
aegon-targaryen..	0
aegon-v-targary..	0
aemon-targarye..	0
aemon-targarye..	0
aenys-frey	0
aeron-greyjoy	0
aerys-i-targaryen	0
aerys-ii-targaryen	0
aggar	0
aggo	0
alayaya	0
albett	0
alebelly	0
alerie-hightower	0
alester-florent	0
alla-tyrell	0
allar-deem	0
allard-seaworth	0
alleras	0
alliser-thorne	0
alyn	0

Figure 46 connected algorithme

- Label Propagation Algorithm (Algorithme de Propagation d'Étiquettes) :

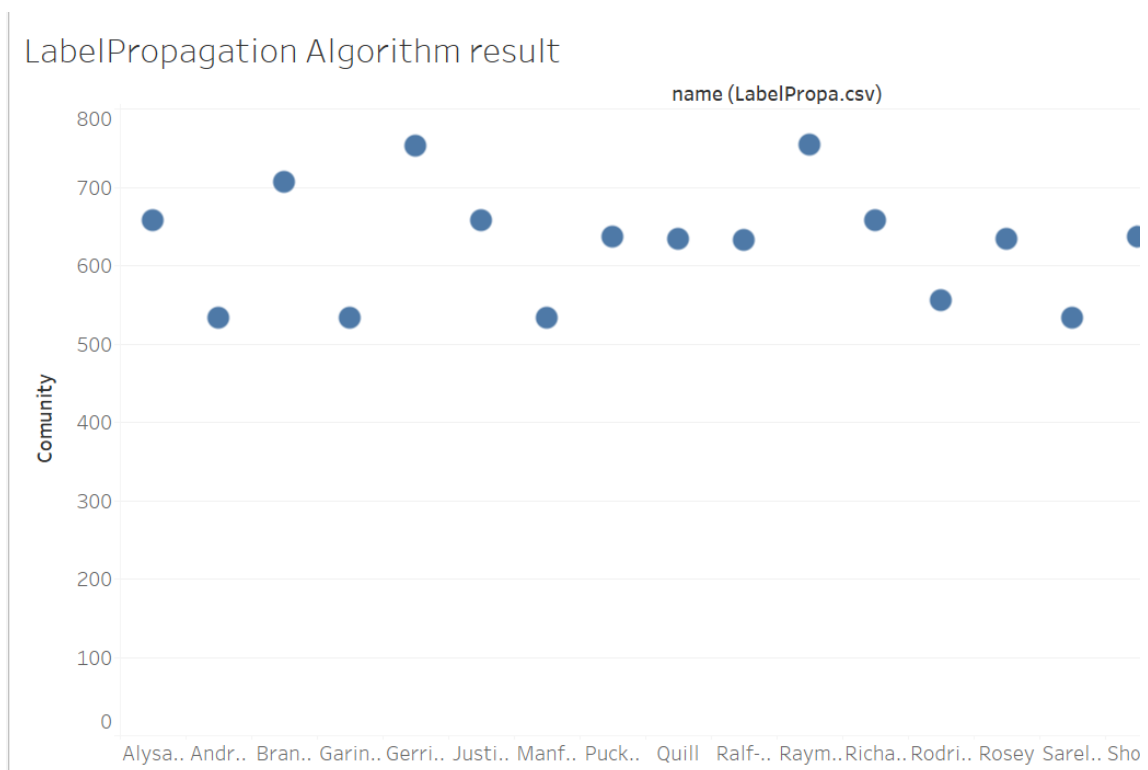


Figure 47 Label Propagation Algorithm

- Triangle Count (Comptage de Triangles) :

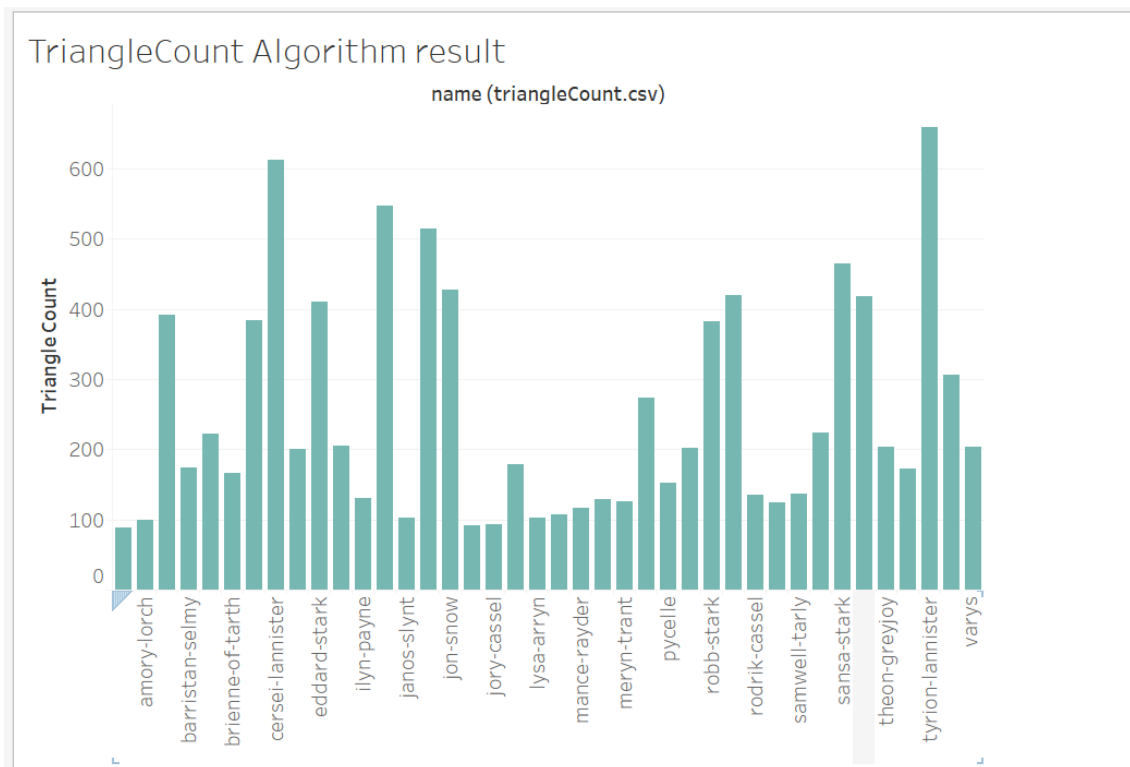


Figure 48 Triangle Count

- Strongly Connected Components (Composantes Fortement Connexes) :

StronglyConnected Algorithm result

Id V	Id C	
0	0	Abc
1	1	Abc
2	2	Abc
3	3	Abc
4	4	Abc
5	5	Abc
6	6	Abc
7	7	Abc
8	8	Abc
9	9	Abc
10	10	Abc
11	11	Abc
12	12	Abc
13	13	Abc
14	14	Abc
15	15	Abc
16	16	Abc
17	17	Abc
18	18	Abc
19	19	Abc
20	20	Abc
21	21	Abc
22	22	Abc
23	23	Abc
24	24	Abc
25	25	Abc
26	26	Abc
27	27	Abc
28	28	Abc
29	29	Abc

Figure 49 strongly Connected algorithm

- Dashboard :

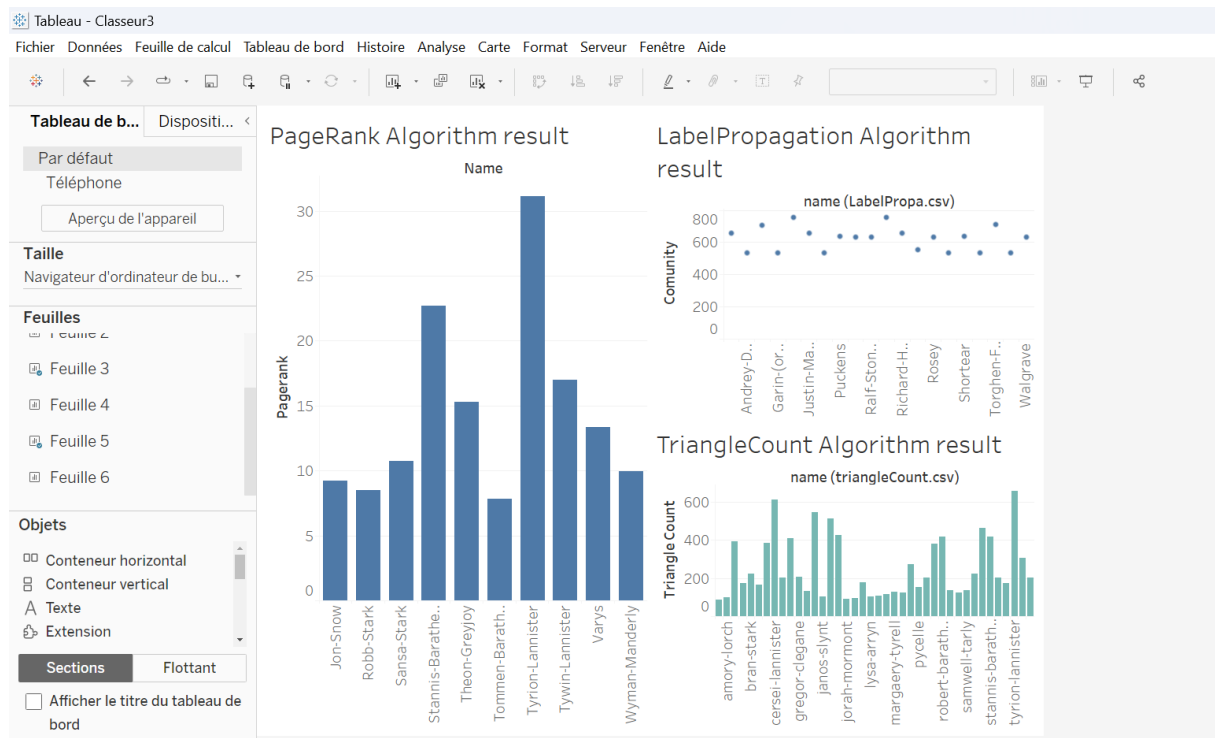


Figure 50 Dashboard

12 Spark ML :

Il faut créer un nouveau notebook pour la partie SparkML

12.1 Récupération des données

- Récupération des données d'après neo4j en utilisant pyspark :

```

%spark.pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[1]") \
    .appName("Neo4jConnection") \
    .getOrCreate()
query = """
MATCH (source)-[relation]->(target)
WHERE (source:Node1 OR target:Node1)
RETURN source.name as Source, target.name as Target, type(relation) as Type, relation.weight as weight, relation.book as book
"""

data = spark.read.format("org.neo4j.spark.DataSource") \
    .option("url", "bolt://neo4j:7687") \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", "neo4j") \
    .option("authentication.basic.password", "bitnami1") \
    .option("query", query) \
    .option("partitions", "1") \
    .load()

```

Took 2 sec. Last updated by anonymous at December 18 2023, 7:51:46 PM.

Figure 51 récupération des données

```

%spark.pyspark
# Vérification des données récupérées depuis Neo4j
data.show()
data.printSchema()

```

Source	Target	Type	weight	book
addam-marbrand	jaime-lannister	Undirected	3	1
addam-marbrand	tywin-lannister	Undirected	6	1
aegon-i-targaryen	daenerys-targaryen	Undirected	5	1
aegon-i-targaryen	edward-stark	Undirected	4	1
aemon-targaryen-...	alliser-thorne	Undirected	4	1
aemon-targaryen-...	bowen-marsh	Undirected	4	1
aemon-targaryen-...	chett	Undirected	9	1
aemon-targaryen-...	clydas	Undirected	5	1
aemon-targaryen-...	jeon-mormont	Undirected	13	1
aemon-targaryen-...	jon-snow	Undirected	34	1
aemon-targaryen-...	samwell-tarly	Undirected	5	1
aerys-ii-targaryen	brandon-stark	Undirected	4	1
aerys-ii-targaryen	edward-stark	Undirected	10	1
aerys-ii-targaryen	gerold-hightower	Undirected	3	1

Took 24 sec. Last updated by anonymous at December 18 2023, 7:52:14 PM.

Figure 52 affichage des données

12.2 Enrichir le dataset

Pour enrichir le jeu de données, vous pouvez ajouter de nouvelles fonctionnalités telles que la similarité des poids ("similarity weight") ainsi que différentes mesures de centralité telles que la centralité degré ("Degree Centrality"), le degré pondéré ("Weighted Degree"), la centralité vectorielle ("Eigenvector Centrality"), le PageRank, et la centralité d'intermédiation ("Betweenness Centrality").

- **Centralité de Degré (Degree Centrality)** : La centralité de degré mesure le nombre de liens qu'un nœud a dans un réseau. Un nœud avec un degré élevé est considéré comme important car il est directement connecté à un grand nombre d'autres nœuds.
- **Degré Pondéré (Weighted Degree)** : Le degré pondéré est similaire à la centralité de degré, mais il prend en compte les poids des liens. Cela signifie qu'au lieu de

simplement compter le nombre de liens, on considère également les valeurs ou les poids associés à ces liens.

- **Centralité de Vecteur Propre (Eigenvector Centrality) :** La centralité de vecteur propre évalue l'importance d'un nœud en tenant compte non seulement de ses liens directs mais aussi de l'importance des nœuds auxquels il est connecté. Les nœuds connectés à d'autres nœuds importants auront une centralité plus élevée.
- **PageRank :** PageRank est une mesure de la importance d'une page Web développée par Google. Dans le contexte des réseaux, PageRank évalue la probabilité qu'un nœud soit visité de manière aléatoire. Les nœuds qui sont liés à beaucoup d'autres nœuds importants ont un score PageRank plus élevé.
- **Centralité d'Intermédierité (Betweenness Centrality) :** La centralité d'intermédierité mesure la fréquence à laquelle un nœud agit en tant qu'intermédiaire sur le chemin le plus court entre deux autres nœuds du réseau. Un nœud avec une centralité d'intermédierité élevée est crucial pour maintenir la communication entre différentes parties du réseau.

```
%spark.pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit, count, when
import pandas as pd
import networkx as nx

# Initialize Spark session
spark = SparkSession.builder.master("local[1]") \
    .appName("Neo4jConnection") \
    .getOrCreate()

# Define your Neo4j query to fetch data
query = """
MATCH (source)-[relation]->(target)
WHERE (source:Node1 OR target:Node1)
RETURN source.name as Source, target.name as Target, type(relation) as Type, relation.weight as weight, relation.book as book
"""

# Load data from Neo4j into a Spark DataFrame
data = spark.read.format("org.neo4j.spark.DataSource") \
    .option("url", "bolt://neo4j:7687") \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", "neo4j") \
    .option("authentication.basic.password", "bitnami") \
    .option("query", query) \
    .option("partitions", "1") \
    .load()

# Add a column for similarity based on the weight of the relationship
data = data.withColumn("Similarity_Weight", when(col("weight") > 5, lit("High"))
    .when((col("weight") <= 5) & (col("weight") > 3), lit("Medium"))
    .otherwise(lit("Low")))

# Calculate Degree for each node (Source and Target)
degrees_source = data.groupBy("Source").agg(count("Target").alias("Source_Degree"))
degrees_target = data.groupBy("Target").agg(count("Source").alias("Target_Degree"))

# Join degree with the main DataFrame
data = data.join(degrees_source, "Source", "left").join(degrees_target, "Target", "left")

# Export Spark DataFrame to Pandas
pandas_data = data.toPandas()

# Create a graph using NetworkX from Source and Target columns
graph = nx.from_pandas_edgelist(pandas_data, 'Source', 'Target', create_using=nx.Graph())
```

```
# Calculate different centrality measures
degree_centrality = nx.degree_centrality(graph)
weighted_degree = dict(graph.degree(weight='weight'))
eigenvector_centrality = nx.eigenvector_centrality_numpy(graph, weight='weight')
pagerank = nx.pagerank(graph, weight='weight')
betweenness_centrality = nx.betweenness_centrality(graph, weight='weight')

# Create Pandas DataFrames for each centrality measure
degree_df = pd.DataFrame(list(degree_centrality.items()), columns=['Node', 'Degree_Centrality'])
weighted_degree_df = pd.DataFrame(list(weighted_degree.items()), columns=['Node', 'Weighted_Degree'])
eigenvector_df = pd.DataFrame(list(eigenvector_centrality.items()), columns=['Node', 'Eigenvector_Centrality'])
pagerank_df = pd.DataFrame(list(pagerank.items()), columns=['Node', 'PageRank'])
betweenness_df = pd.DataFrame(list(betweenness_centrality.items()), columns=['Node', 'Betweenness_Centrality'])

# Convert Pandas DataFrames to Spark DataFrames
degree_spark_df = spark.createDataFrame(degree_df)
weighted_degree_spark_df = spark.createDataFrame(weighted_degree_df)
eigenvector_spark_df = spark.createDataFrame(eigenvector_df)
pagerank_spark_df = spark.createDataFrame(pagerank_df)
betweenness_spark_df = spark.createDataFrame(betweenness_df)

# Join the centrality measures with the original DataFrame
data = data.join(degree_spark_df, data.Source == degree_spark_df.Node, "left").drop("Node")
data = data.join(weighted_degree_spark_df, data.Source == weighted_degree_spark_df.Node, "left").drop("Node")
data = data.join(eigenvector_spark_df, data.Source == eigenvector_spark_df.Node, "left").drop("Node")
data = data.join(pagerank_spark_df, data.Source == pagerank_spark_df.Node, "left").drop("Node")
data = data.join(betweenness_spark_df, data.Source == betweenness_spark_df.Node, "left").drop("Node")

# Fill null values with 0 in case some nodes don't have centrality measures
data = data.fillna(0)

# Display the final DataFrame with centrality features added
data.show()
data.printSchema()
```

Target	Source	Type	weight	book	Similarity_Weight	Source_Degree	Target_Degree	Degree_Centrality	Weighted_Degree	Eigenvector_Centrality	PageRank	Betweenness_Centrality
gretchell	colemon	undirected	5	4	Medium	5	1	0.007547169811320755	6	0.014396674720807114	0.001020322720295135	1.147492172372695...
maddy	colemon	undirected	5	4	Medium	5	2	0.007547169811320755	6	0.014396674720807114	0.001020322720295135	1.147492172372695...
petyr-baelish	colemon	undirected	6	4	High	5	43	0.007547169811320755	6	0.014396674720807114	0.001020322720295135	1.147492172372695...
robert-arryn	colemon	undirected	7	4	High	5	19	0.007547169811320755	6	0.014396674720807114	0.001020322720295135	1.147492172372695...
vardis-egen	colemon	Undirected	4	1	Medium	5	6	0.007547169811320755	6	0.014396674720807114	0.001020322720295135	1.147492172372695...
munch	gariss	Undirected	5	2	Medium	1	1	0.002515723270440...	2	5.007591103638332E-5	0.001060539254437...	0.002515723270440...
melara-hetherspoon	maggy	undirected	3	4	Low	1	4	0.002515723270440...	2	0.008828625680492614	5.16362182620101E-4	0.0
narbert-grandison	benethon-scales	undirected	3	5	Low	2	4	0.002515723270440...	2	0.001209935220879...	4.998085667795787E-4	0.0

Figure 53 resultat

12.3 Application de quelques algorithmes de prédiction :

a. Prediction du Degree_Centrality :

Ce code utilise la régression linéaire pour prédire la centralité degré ('Degree_Centrality') en utilisant différentes caractéristiques ('weight', 'book', 'Source_Degree', 'Target_Degree', 'Weighted_Degree', 'Eigenvector_Centrality', 'PageRank', 'Betweenness_Centrality') du DataFrame. Ensuite, il évalue la performance du modèle en utilisant des métriques telles que MSE, RMSE et R^2 sur les données de test.

```
%spark.pyspark
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml import Pipeline
from pyspark.sql import SparkSession

# Initialise Spark session
spark = SparkSession.builder.master("local[1]") \
    .appName("Predict_Degree_Centrality") \
    .getOrCreate()

# Assume 'data' contient votre DataFrame avec toutes les caractéristiques nécessaires, y compris 'Degree_Centrality'

# Liste des colonnes utilisées comme caractéristiques pour la prédiction de Degree_Centrality
feature_columns = ['weight', 'book', 'Source_Degree', 'Target_Degree', 'Weighted_Degree',
                  'Eigenvector_Centrality', 'PageRank', 'Betweenness_Centrality']

# Utiliser VectorAssembler pour assembler toutes les colonnes de features en une seule colonne
assembler = VectorAssembler(inputCols=feature_columns, outputCol='features')

# Créer un objet LinearRegression
lr = LinearRegression(featuresCol='features', labelCol='Degree_Centrality')

# Créer un pipeline pour assembler les étapes de prétraitement et le modèle
pipeline = Pipeline(stages=[assembler, lr])

# Diviser les données en ensembles d'entraînement et de test (80% pour l'entraînement et 20% pour les tests)
train_data, test_data = data.randomSplit([0.8, 0.2], seed=12345)

# Entraîner le modèle sur les données d'entraînement
model = pipeline.fit(train_data)

# Faire des prédictions sur les données de test
predictions = model.transform(test_data)

# Afficher les prédictions et les valeurs réelles
predictions.select('Degree_Centrality', 'prediction', *feature_columns).show()

from pyspark.ml.evaluation import RegressionEvaluator

# Calculer la MSE
evaluator = RegressionEvaluator(labelCol="Degree_Centrality", predictionCol="prediction", metricName="mse")
mse = evaluator.evaluate(predictions)
print("Mean Squared Error (MSE) on test data = {:.4f}".format(mse))

# Calculer la RMSE
rmse = evaluator.evaluate(predictions, {evaluator.metricName: "rmse"})
print("Root Mean Squared Error (RMSE) on test data = {:.4f}".format(rmse))

# Calculer R² (Coefficient de détermination)
```

Figure 54 Prediction du Degree_Centrality

Degree_Centrality	prediction	weight	book	Source_Degree	Target_Degree	Weighted_Degree	Eigenvector_Centrality	PageRank	Betweenness_Centrality
0.007547169811320755	0.007547169811320...	7	4	5	19	6	0.01439667472909714	0.001020322720295135	1.147492172372695...
0.018867924528301886	0.018867924528301147	8	4	21	9	15	0.013378575325244044	0.002525087193158...	0.003894449146905064
0.012578616352201257	0.012578616352201097	3	1	9	98	10	0.036162700053220545	0.001473004418379...	3.475448911599839E-4
0.018867924528301886	0.01886792452830109	32	3	21	66	15	0.013378575325244044	0.002525087193158...	0.003894449146905064

Figure 55 resultat

Mean Squared Error (MSE) on test data = 0.0000
 Root Mean Squared Error (RMSE) on test data = 0.0000
 R-squared (R^2) on test data = 1.0000

Figure 56 les métriques du model

b. Prediction du PageRank :

```
%spark.pyspark
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import RegressionEvaluator

# Créer un VectorAssembler pour assembler les colonnes de features
feature_columns = ['Source_Degree', 'Target_Degree', 'Degree_Centrality', 'Weighted_Degree',
                  'Eigenvector_Centrality', 'Betweenness_Centrality', 'weight']
assembler = VectorAssembler(inputCols=feature_columns, outputCol='features')

# Créer un modèle DecisionTreeRegressor
dt = DecisionTreeRegressor(featuresCol='features', labelCol='PageRank')

# Créer un pipeline
pipeline = Pipeline(stages=[assembler, dt])

# Diviser les données en ensembles d'entraînement et de test
train_data, test_data = data.randomSplit([0.8, 0.2], seed=12345)

# Entraîner le modèle
model = pipeline.fit(train_data)

# Faire des prédictions sur les données de test
predictions = model.transform(test_data)

# Afficher les prédictions et les valeurs réelles
predictions.select('PageRank', 'prediction', *feature_columns).show()

# Calculer des métriques d'évaluation pour le modèle
evaluator = RegressionEvaluator(labelCol='PageRank', predictionCol='prediction', metricName='rmse')
mse = evaluator.evaluate(predictions)
print("Mean Squared Error (MSE) on test data = {:.4f}".format(mse))
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = {:.4f}".format(rmse))
r2 = evaluator.evaluate(predictions, {evaluator.metricName: "r2"})
print("R-squared (R²) on test data = {:.4f}".format(r2))
```

PageRank	prediction	Source_Degree	Target_Degree	Degree_Centrality	Weighted_Degree	Eigenvector_Centrality	Betweenness_Centrality	weight
0.001020322720295135	0.001160135446517...	5	19	0.007547169811320755	6	0.01439667472900714	1.147492172372695...	7
0.002525087193158...	0.002425692646480608	21	9	0.018867924528301806	15	0.013378575325244044	0.003894449146905064	8
0.001473004418379...	0.001434971445018...	9	98	0.012578616352201257	10	0.036162700053220545	3.475448911599839E-4	3

Figure 57 Prediction du PageRank

Mean Squared Error (MSE) on test data = 0.0002
Root Mean Squared Error (RMSE) on test data = 0.0002
R-squared (R²) on test data = 0.9984

Took 38 sec. Last updated by anonymous at December 19 2023, 8:03:37 PM.

Figure 58 les metriques du model

c. Prediction du Betweenness_Centrality

```
%spark.pyspark
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler
from pyspark.sql import SparkSession

# Initialiser Spark session
spark = SparkSession.builder.master("local[1]") \
    .appName("Betweenness_Prediction_RF") \
    .getOrCreate()

# Assume 'data' contains your prepared DataFrame with all the features and target variable 'Betweenness_Centrality'

# Utiliser VectorAssembler pour assembler toutes les colonnes de features en une seule colonne
feature_columns = ['Source_Degree', 'Target_Degree', 'Degree_Centrality', 'Weighted_Degree',
                  'Eigenvector_Centrality', 'PageRank', 'weight'] # Ajoutez 'weight' comme feature
assembler = VectorAssembler(inputCols=feature_columns, outputCol='features')

# Créer un objet RandomForestRegressor
rf = RandomForestRegressor(featuresCol='features', labelCol='Betweenness_Centrality')

# Créer un pipeline pour assembler les étapes de prétraitement et le modèle
pipeline = Pipeline(stages=[assembler, rf])

# Diviser les données en ensembles d'entraînement et de test (80% pour l'entraînement et 20% pour les tests)
train_data, test_data = data.randomSplit([0.8, 0.2], seed=12345)

# Entraîner le modèle sur les données d'entraînement
model = pipeline.fit(train_data)

# Faire des prédictions sur les données de test
predictions = model.transform(test_data)

# Afficher les prédictions et les valeurs réelles
predictions.select('Betweenness_Centrality', 'prediction', *feature_columns).show()

from pyspark.ml.evaluation import RegressionEvaluator

# Calculer la MSE
evaluator = RegressionEvaluator(labelCol='Betweenness_Centrality', predictionCol='prediction', metricName='mse')
mse = evaluator.evaluate(predictions)
print("Mean Squared Error (MSE) on test data = {:.4f}".format(mse))
```

Figure 59 Prediction du Betweenness_Centrality

Betweenness_Centrality	prediction	Source_Degree	Target_Degree	Degree_Centrality	Weighted_Degree	Eigenvector_Centrality	PageRank	weight
1.147492172372695...	7.59971962848845E-4	5	19	0.007547169811320755	6	0.014396674729097114	0.001820322720295135	7
0.003894449146905064	0.003155416356103...	21	9	0.018867924528301886	15	0.013378575325244044	0.002525087193158...	8
3.475448911599839E-4	9.250218555919745E-4	9	98	0.012578616352201257	10	0.036162700053220545	0.001473004418379...	3
0.003894449146905064	0.003155416356103...	21	66	0.018867924528301886	15	0.013378575325244044	0.002525087193158...	32
0.002589043870249...	0.548209687138108E-4	3	1	0.007547169811320755	6	0.008617798266236121	0.001190758191296...	3
3.475448911599839E-4	9.250218555919745E-4	9	5	0.012578616352201257	10	0.036162700053220545	0.001473004418379...	5
0.003894449146905064	0.003155416356103...	21	14	0.018867924528301886	15	0.013378575325244044	0.002525087193158...	3
0.001442737094176...	0.0008483234820715	23	195	0.03522012578616352	28	0.12110155136394017	0.003534652914421839	8
0.001442737094176...	0.0008483234820715	23	195	0.03522012578616352	28	0.12110155136394017	0.003534652914421839	8

Figure 60 resultat

Mean Squared Error (MSE) on test data = 0.0000
Root Mean Squared Error (RMSE) on test data = 0.0029
R-squared (R²) on test data = 0.9958

Figure 61 les metriques du model

d. Prediction du Weighted_Degree

```
%spark.pyspark
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml import Pipeline
from pyspark.sql import SparkSession

# Initialiser Spark session
spark = SparkSession.builder.master("local[1]") \
    .appName("Predict_Weighted_Degree_RF") \
    .getOrCreate()

# Assume 'data' contains your prepared DataFrame with all the features and target variable 'Weighted_Degree'

# Utiliser VectorAssembler pour assembler toutes les colonnes de features en une seule colonne
feature_columns = ['Source_Degree', 'Target_Degree', 'Degree_Centrality', 'weight', 'Eigenvector_Centrality', 'PageRank', 'Betweenness_Centrality']
assembler = VectorAssembler(inputCols=feature_columns, outputCol='features')

# Créer un objet RandomForestRegressor
rf = RandomForestRegressor(featuresCol='features', labelCol='Weighted_Degree')

# Créer un pipeline pour assembler les étapes de prétraitement et le modèle
pipeline = Pipeline(stages=[assembler, rf])

# Diviser les données en ensembles d'entraînement et de test (80% pour l'entraînement et 20% pour les tests)
train_data, test_data = data.randomSplit([0.8, 0.2], seed=12345)

# Entraîner le modèle sur les données d'entraînement
model = pipeline.fit(train_data)

# Faire des prédictions sur les données de test
predictions = model.transform(test_data)

# Afficher les prédictions et les valeurs réelles
predictions.select('Weighted_Degree', 'prediction', *feature_columns).show()

from pyspark.ml.evaluation import RegressionEvaluator

# Calculer la MSE
evaluator = RegressionEvaluator(labelCol="Weighted_Degree", predictionCol="prediction", metricName="mse")
mse = evaluator.evaluate(predictions)
print("Mean Squared Error (MSE) on test data = {:.4f}".format(mse))

# Calculer la RMSE
rmse = evaluator.evaluate(predictions, {evaluator.metricName: "rmse"})
print("Root Mean Squared Error (RMSE) on test data = {:.4f}".format(rmse))

# Calculer R² (Coefficient de détermination)
r2 = evaluator.evaluate(predictions, {evaluator.metricName: "r2"})
print("R-squared (R²) on test data = {:.4f}".format(r2))
```

Figure 62 Prediction du Weighted_Degree

Weighted_Degree	prediction	Source_Degree	Target_Degree	Degree_Centrality	weight	Eigenvector_Centrality	PageRank	Betweenness_Centrality
6	6.030636191608099	5	19	0.007547169811320755	7	0.014396674729097114	0.001820322720295135	1.147492172372695...
15	15.14306052676994	21	9	0.018867924528301886	8	0.013378575325244044	0.002525087193158...	0.003894449146905064
10	9.195151632722043	9	98	0.012578616352201257	3	0.036162700053220545	0.001473004418379...	3.475448911599839E-4
15	15.14306052676994	21	66	0.018867924528301886	32	0.013378575325244044	0.002525087193158...	0.003894449146905064
6	6.322240717691403	3	1	0.007547169811320755	3	0.008617798266236121	0.001190758191296...	0.002589043870249...
10	9.195151632722043	9	5	0.012578616352201257	5	0.036162700053220545	0.001473004418379...	3.475448911599839E-4
15	15.14306052676994	21	14	0.018867924528301886	3	0.013378575325244044	0.002525087193158...	0.003894449146905064
28	27.071408453140116	23	195	0.03522012578616352	8	0.12110155136394017	0.003534652914421839	0.001442737094176...
28	27.071408453140116	23	195	0.03522012578616352	8	0.12110155136394017	0.003534652914421839	0.001442737094176...
5	4.399844492212945	5	3	0.006289308176100628	4	0.00353168880907368	0.632889054466047E-4	0.0
5	4.399844492212945	5	3	0.006289308176100628	3	0.00353168880907368	0.632889054466047E-4	0.0

Figure 63 resultat

Mean Squared Error (MSE) on test data = 0.5610
Root Mean Squared Error (RMSE) on test data = 0.7490
R-squared (R²) on test data = 0.9995

Took 1 min 17 sec. Last updated by anonymous at December 19 2023, 8:41:16 PM.

Figure 64 les metriques

12.4: Clustering du Weight (KMeans)

```
%pyspark

from pyspark.sql import SparkSession
from pyspark.ml.clustering import KMeans
from pyspark.ml.feature import VectorAssembler
from pyspark.sql.functions import col

# Initialiser Spark session
spark = SparkSession.builder.master("local[1]").appName("Weight_Clustering").getOrCreate()

# Convertir la colonne "weight" en type numérique (double)
data = data.withColumn("weight", col("weight").cast("double"))

# Supprimer les lignes avec des valeurs nulles dans la colonne 'weight'
data = data.dropna(subset=["weight"])

# Assembler les caractéristiques en un vecteur, en traitant les valeurs nulles
assembler = VectorAssembler(inputCols=["weight"], outputCol="features", handleInvalid="keep")
assembled_df = assembler.transform(data)

# Créer le modèle K-Means avec k=3 clusters
kmeans = KMeans(featuresCol="features", k=3)

# Entraîner le modèle K-Means
kmeans_model = kmeans.fit(assembled_df)

# Faire des prédictions sur les données
predictions = kmeans_model.transform(assembled_df)

# Afficher les données d'origine et les prédictions du K-Means
result = predictions.select("Source", "Target", "weight", "prediction")
result.show(100)

result.coalesce(1).write.csv('/zeppelin/notebook/result3.csv', header=True)
```

Figure 65 le clustering avec Kmeans

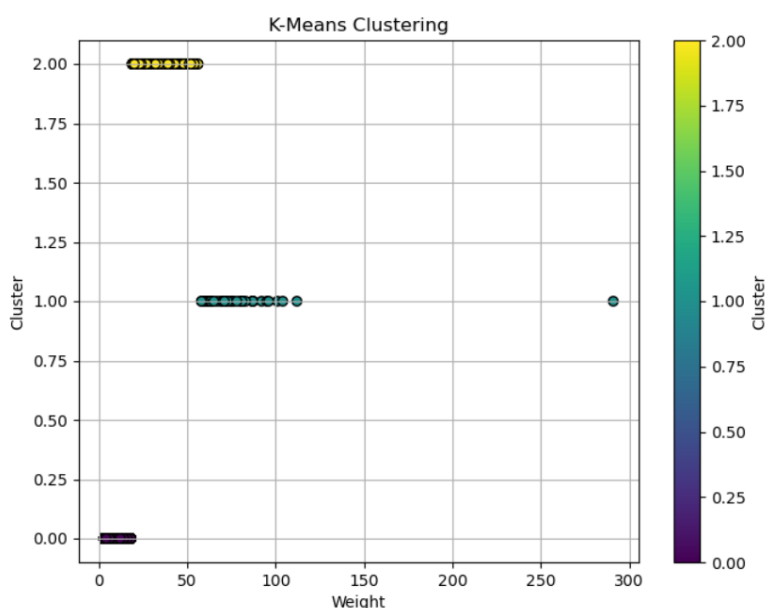
```
%python
import pandas as pd
import matplotlib.pyplot as plt

# Load the saved CSV file into a Pandas DataFrame
result_pandas = pd.read_csv('/zeppelin/notebook/result3.csv/part-00000-fbb41e2a-e96e-4507-aeaa-28e1a1717cae-c000.csv')

# Extract relevant columns for visualization
x = result_pandas['weight']
y = result_pandas['prediction']

# Plotting the scatter plot
plt.figure(figsize=(8, 6))
plt.scatter(x, y, c=y, cmap='viridis', edgecolors='k')
plt.title('K-Means Clustering')
plt.xlabel('Weight')
plt.ylabel('Cluster')
plt.colorbar(label='Cluster')
plt.grid(True)
plt.show()
z.show(plt)
```

Figure 66 affichage des clusters



On a trouvé la présence de 3 clusters des personnes et une valeur aberrant.