Finding Connected Components in Graph

Une approche MapReduce pour l'analyse de graphes à grande échelle

Jean-Christophe HAMARD et Dina HOURLIER

Plan de la Présentation

1. Introduction et Contexte

Définition des graphes et des composants connexes • Applications pratiques

2. Objectif de l'Étude

Objectif principal • Cas d'usage concret • Méthode proposée

3. Algorithme CCF (Connected Component Finder)

Vue d'ensemble • CCF-Iterate • Optimisation par tri secondaire • CCF-Dedup

4. Implémentations et Évaluation

Jeux de données • Méthodologie • PySpark (RDD, DataFrame) • Scala (RDD, DataFrame, GraphX)

5. Résultats et Analyse

Comparaison des performances • Facteurs influant • Recommandations

6. Conclusion et perspectives

Synthèse • Améliorations futures • Concept de centralité

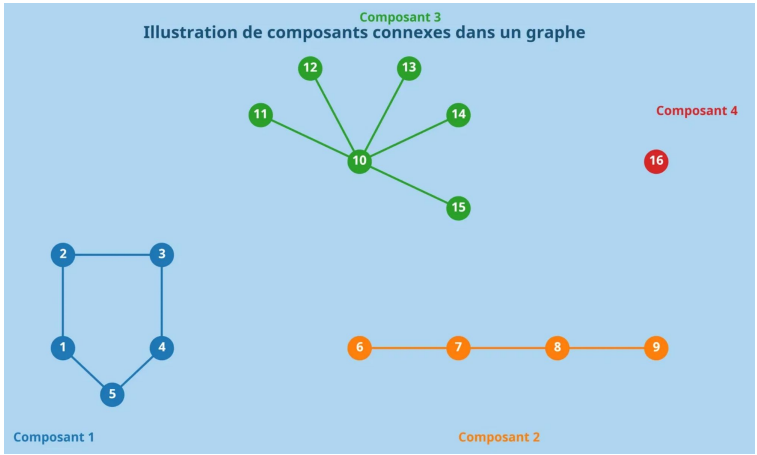
Introduction - Contexte et définitions (1/2)

Qu'est-ce qu'un Graphe ?

Un graphe est une structure mathématique composée de nœuds (ou sommets) et de liens (ou arêtes) qui les connectent. Il est utilisé pour modéliser des relations entre des entités.

Qu'est-ce qu'un Composant Connexe ?

Un composant connexe est un sous-graphe où chaque nœud est accessible à partir de tous les autres nœuds du même sous-graphe via une séquence de liens. En d'autres termes, c'est un groupe de nœuds interconnectés.



Introduction - Contexte et définitions (2/2)

Applications Pratiques

Réseaux Sociaux

Détection de communautés ou de groupes d'amis partageant des intérêts communs et interagissant fréquemment entre eux.

Bases de Données

Recherche de doublons pour regrouper des informations appartenant à la même entité, même si ces données proviennent de sources différentes.

Traitement d'Images

Segmentation d'images en régions connectées pour l'analyse d'objets, la reconnaissance de formes ou la détection de contours.

Objectif de l'étude (1/2)

Objectif Principal

Proposer une méthode efficace et hautement scalable pour trouver tous les composants connexes d'un graphe, en utilisant le modèle de programmation MapReduce pour le traitement parallèle de données massives.

Défis Adressés

Passage à l'échelle

Traitement de graphes contenant des milliards de nœuds et d'arêtes

Contraintes mémoire

Optimisation pour des environnements à mémoire limitée comme le cloud

Distribution

Compatibilité avec les infrastructures distribuées comme Hadoop

Objectif de l'étude (2/2)

Cas d'usage concret

Les auteurs ont utilisé leur méthode pour un moteur de recherche de personnes basé sur des données personnelles américaines (mariages, réseaux sociaux, propriétés, etc.), totalisant des milliards d'enregistrements.

L'objectif était de regrouper toutes les données liées à une même personne (même si ces données proviennent de sources différentes), ce qui revient à trouver les composants connexes dans un graphe géant.

Méthode proposée : CCF

CCF-Iterate: : Propagation des identifiants minimaux à travers le graphe

CCF-Dedup : Suppression des doublons pour optimiser les performances

Convergence : L'algorithme s'arrête lorsqu'aucun nouveau lien n'est découvert

Algorithme CCF - Vue d'ensemble

Connected Component Finder

Le CCF est un algorithme MapReduce itératif conçu pour identifier les composants connexes dans des graphes de très grande taille.

1. CCF-Iterate

Propagation des identifiants minimaux à travers le graphe. Chaque nœud adopte l'identifiant le plus petit parmi ses voisins.

2. CCF-Dedup

Suppression des doublons générés lors de la phase précédente pour optimiser les performances.

Critère de Convergence

L'algorithme s'arrête lorsqu'aucun nouveau lien n'est découvert, signifiant que tous les nœuds d'un même composant partagent le même identifiant minimal.

Processus itératif

Initialisation : chaque nœud est son propre composant

Propagation des identifiants minimaux (CCF-Iterate)

Suppression des doublons (CCF-Dedup)

Répétition jusqu'à convergence

Résultat : mapping (nœud → identifiant de composant)

Algorithme CCF - CCF-Iterate

Principe de CCF-Iterate

CCF-Iterate est la première phase de l'algorithme CCF, responsable de la propagation des identifiants minimaux à travers le graphe pour identifier les composants connexes.

Étapes détaillées

Transformation des arêtes : Chaque arête (key, value) est transformée en deux sorties : (key, value) et (value, key), rendant le graphe non orienté.

Recherche du minimum : Pour chaque nœud, on cherche l'identifiant minimal parmi lui-même et ses voisins.

Propagation : Si un nœud trouve un identifiant plus petit, il l'adopte et le propage à ses voisins.

```
Map(key, value):
  emit(key, value)
  emit(value, key)

Reduce(key, values):
  minValue = values.next()
  if minValue < key:
    emit(key, minValue)
    for v in values:
      emit(v, minValue)
```

Convergence de l'algorithme

L'algorithme s'arrête lorsqu'aucun nouveau lien n'est découvert, signifiant que tous les nœuds d'un même composant partagent le même identifiant minimal.

Algorithme CCF - Optimisation par tri secondaire

Amélioration de l'efficacité de CCF-Iterate

Version classique

```
reduce(key, <iterable> values)
    min <- key
    for each (value ∈ values)
        if (value < min)
            min <- value
        valueList.add(value)
    if (min < key)
        emit(key, min)
        for each (value ∈
valueList)
            if (min ≠ value)

Counter.NewPair.increment(1)
                emit(value, min)
```

Dans la version standard, pour chaque nœud, l'algorithme doit parcourir toute la liste de ses voisins pour trouver le plus petit identifiant.

Avec tri secondaire

```
map(key, value)
    emit(key, value)
    emit(value, key)

reduce(key, <iterable> values)
    minValue <- values.next()
    if (minValue < key)
        emit(key, minValue)
        for each (value ∈ values)

Counter.NewPair.increment(1)
            emit(value, minValue)
```

Grâce au tri secondaire, les voisins sont déjà triés lors de leur arrivée au Reducer. Le plus petit voisin est directement accessible.

Avantages du tri secondaire

Réduction du temps de traitement pour les nœuds avec de nombreux voisins

Optimisation de l'utilisation de la mémoire pendant la phase de réduction

Particulièrement efficace pour les graphes denses

Le tri secondaire permet d'éviter de parcourir l'ensemble des voisins d'un nœud pour trouver le minimum, ce qui améliore significativement les performances pour les graphes de grande taille.

Algorithme CCF - CCF-Dedup

Objectif de CCF-Dedup

La phase CCF-Dedup supprime les paires dupliquées (nœud, minID) générées lors de la phase CCF-Iterate, optimisant les performances des itérations suivantes en réduisant la surcharge d'E/S.

```
Map(key, value):
  // key = (nœud, minID)
  temp = (entity1, entity2)
  emit(temp, null)

Reduce(key, values):
  // key = (entity1, entity2)
  // MapReduce regroupe automatiquement
  // les clés identiques
  emit(entity1, entity2)
```

Processus de dédoublonnage

1. Réception des paires potentiellement dupliquées
2. Transformation en clés composites
3. Élimination automatique des doublons
4. Production d'un ensemble unique

Exemple :

Entrée :

(1,5), (2,5), (1,5)

Map :

((1,5), null), ((2,5), null), ((1,5), null)

Shuffle & Sort :

((1,5), [null, null]), ((2,5), [null])

Sortie :

(1,5), (2,5)

En résumé :

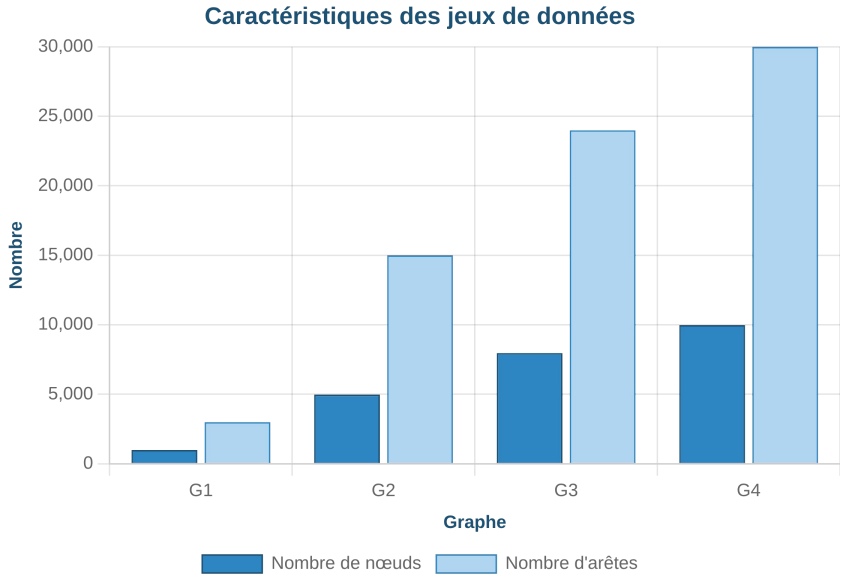
CCF-Dedup agit comme un filtre qui élimine les doublons avant de passer à l'itération suivante, rendant l'algorithme plus rapide et plus léger.

Jeux de Données

Données d'évaluation

Pour évaluer les différentes implémentations de l'algorithme CCF, nous avons utilisé quatre jeux de données de tailles croissantes, représentant des graphes avec différentes caractéristiques.

Graphe	Nœuds	Arêtes	Format
G1	996	3 000	CSV
G2	4 983	15 000	CSV
G3	7 971	24 000	CSV
G4	9 979	30 000	CSV



Chaque fichier est structuré avec deux colonnes : **source** et **target,** représentant les nœuds connectés par une arête.

Exemple: Une ligne 444,683 indique une arête entre le noeud 444 et 683.

Méthodologie d'Évaluation

Approche Expérimentale

Pour évaluer l'efficacité de l'algorithme CCF et comparer différentes implémentations, nous avons adopté une méthodologie rigoureuse basée sur des métriques clés.

Métriques d'évaluation

- **Temps d'exécution:** : Durée totale nécessaire pour identifier tous les composants connexes
- **Nombre d'itérations** : Nombre de cycles nécessaires pour atteindre la convergence
- **Scalabilité** : Evolution des performances avec l’augmentation de la taille du graphe

Implémentations Évaluées

Implémentation	Langage	API
PySpark RDD	Python	RDD
PySpark DataFrame	Python	DataFrame
Scala RDD	Scala	RDD
Scala DataFrame	Scala	DataFrame
Scala GraphX	Scala	GraphX

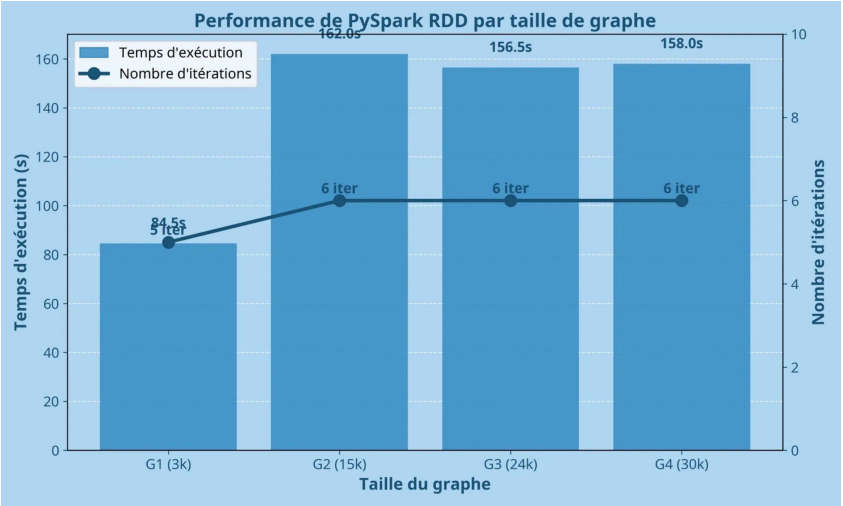
Environnement de Test

Toutes les expérimentations ont été réalisées avec des ressources identiques pour garantir une comparaison équitable entre les différentes implémentations.

Implémentation PySpark RDD

Principe

Utilisation des RDD (Resilient Distributed Datasets), une abstraction de données distribuées et tolérantes aux pannes, pour implémenter l'algorithme CCF en Python.



Avantages

- Simplicité d'implémentation
- Contrôle fin des opérations
- Stabilité des performances

Inconvénients

- Verbosité du code
- Performances limitées par Python
- Moins d'optimisations automatiques

Résultats

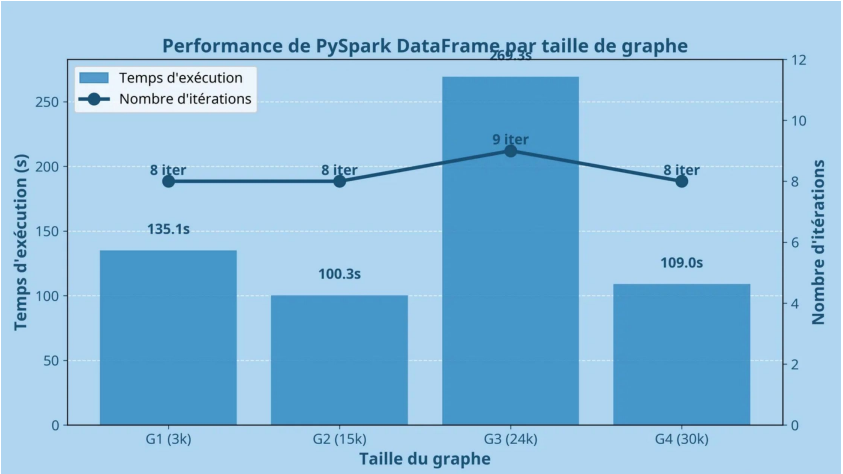
Graphe	Itérations	Temps (s)
G1 (3k arêtes)	5	84.542
G2 (15k arêtes)	6	162.019
G3 (24k arêtes)	6	156.463
G4 (30k arêtes)	6	158.003

Observation : Le temps d'exécution augmente avec la taille du graphe, mais pas de manière parfaitement linéaire. Les graphes G3 et G4 sont traités légèrement plus rapidement que G2 malgré leur taille plus importante.

Implémentation PySpark DataFrame

Principe

Utilisation des DataFrames Spark, une API de haut niveau optimisée pour les requêtes SQL et le traitement structuré des données.



Avantages

- Syntaxe concise et intuitive
- Optimisations Spark SQL
- Facilité d'intégration

Inconvénients

- Temps d'exécution instables
- Plus d'itérations nécessaires

Résultats

Graphe	Itérations	Temps (s)
G1 (3k arêtes)	8	135.066
G2 (15k arêtes)	8	100.337
G3 (24k arêtes)	9	269.291
G4 (30k arêtes)	8	108.962

Observation : Les temps d'exécution sont très variables selon la taille du graphe, avec un pic notable pour G3 (24k arêtes). Cette instabilité est caractéristique de l'implémentation DataFrame en Python.

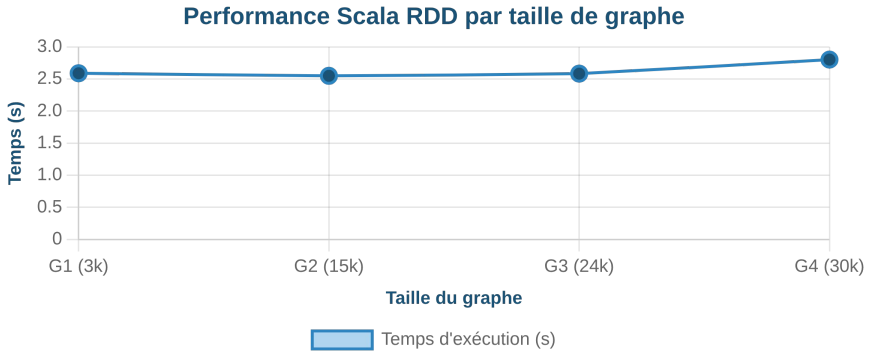
Implémentation Scala RDD

Principe

Implémentation en Scala avec RDD, tirant parti de la JVM et des optimisations Spark pour le traitement distribué des données.

Résultats

Graphe	Itérations	Temps (s)
G1 (3k arêtes)	8	2.613
G2 (15k arêtes)	10	2.573
G3 (24k arêtes)	10	2.607
G4 (30k arêtes)	10	2.828



Avantages

- Performance exceptionnelle
- Stabilité des temps d'exécution
- Excellente scalabilité

Inconvénients

- Syntaxe plus complexe
- Courbe d'apprentissage plus élevée
- Plus d'itérations nécessaires

Observation clé :
Les temps d'exécution restent remarquablement stables malgré l'augmentation de la taille des graphes.

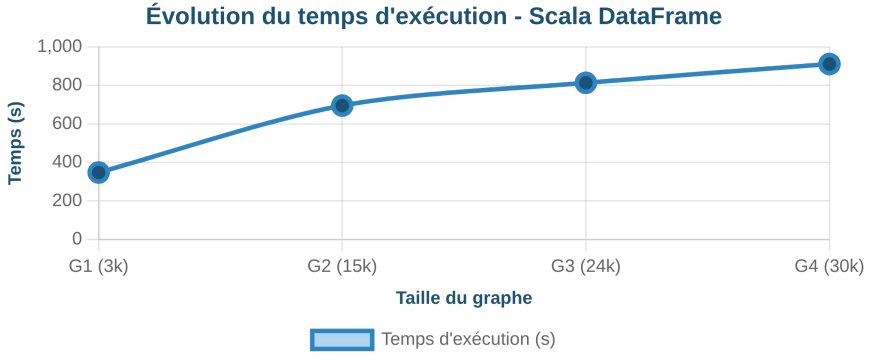
Implémentation Scala DataFrame

Principe

Utilisation de l'API DataFrame de Spark en Scala, une interface de haut niveau pour la manipulation de données structurées.

Résultats

Graphe	Itérations	Temps (s)
G1 (3k arêtes)	8	349.8
G2 (15k arêtes)	10	700.0
G3 (24k arêtes)	10	819.0
G4 (30k arêtes)	10	917.4



Avantages

- API de haut niveau
- Syntaxe déclarative
- Facilité d'intégration

Inconvénients

- Performances très lentes
- Temps croissant avec la taille
- Inadapté aux algorithmes itératifs

Observation : Les temps d'exécution augmentent significativement avec la taille du graphe.

Implémentation Scala GraphX

Principe

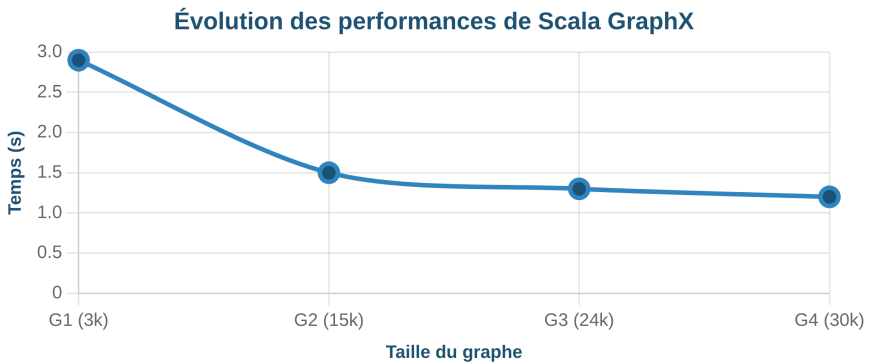
Utilisation de GraphX, bibliothèque native de Spark pour l'analyse de graphes, offrant des algorithmes optimisés pour les opérations sur les graphes.

Résultats

Graphe	Temps (s)
G1 (3k arêtes)	2.9
G2 (15k arêtes)	1.5
G3 (24k arêtes)	1.3
G4 (30k arêtes)	1.2

Points forts

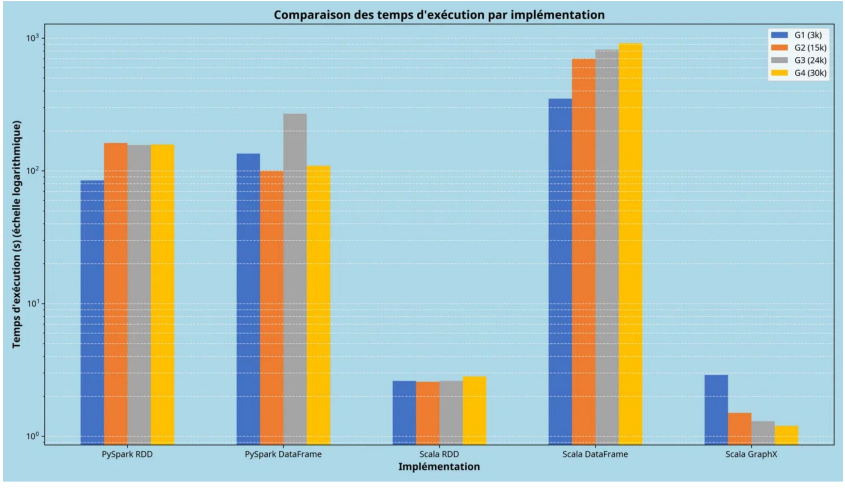
- Performances exceptionnelles
- Code simplifié
- Optimisations natives
- Scalabilité excellente



Résultats Comparatifs

Comparaison des temps d'exécution (échelle logarithmique)

Implémentation	G1 (3k)	G2 (15k)	G3 (24k)	G4 (30k)
PySpark RDD	84.542s	162.019s	156.463s	158.003s
PySpark DataFrame	135.066s	100.337s	269.291s	108.962s
Scala RDD	2.613s	2.573s	2.607s	2.828s
Scala DataFrame	349.8s	700.0s	819.0s	917.4s
Scala GraphX	2.9s	1.5s	1.3s	1.2s



Observation clé

Les implémentations Scala RDD et GraphX sont significativement plus performantes (facteur 30-100x) que les autres approches.

Analyse des Performances

Facteurs influençant les performances

Langage de programmation

Scala (JVM) offre des performances nettement supérieures à Python pour les traitements intensifs.

API utilisée

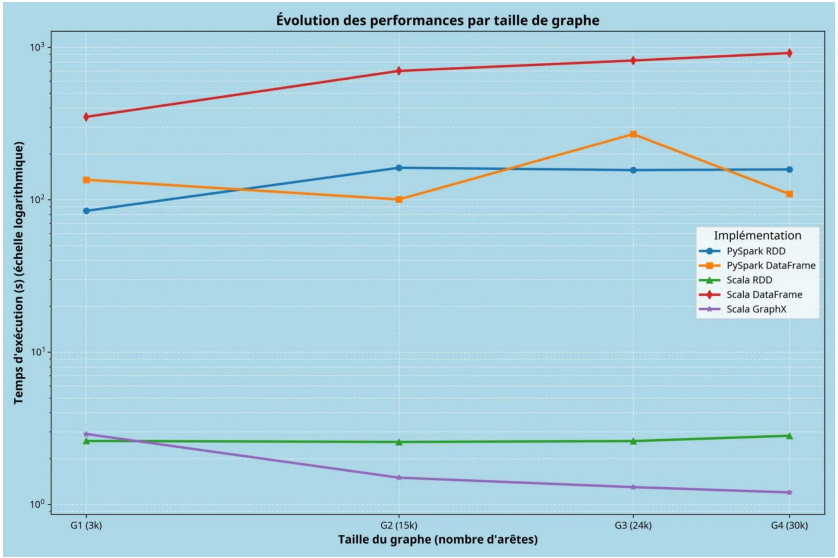
GraphX > RDD > DataFrame pour les algorithmes itératifs sur les graphes.

Optimisations spécifiques

Les bibliothèques spécialisées (GraphX) intègrent des optimisations pour les opérations sur les graphes.

Recommandation

Pour les applications de production traitant des graphes à grande échelle, privilégier Scala GraphX ou Scala RDD.



Conclusion et perspectives

Synthèse des résultats

L'algorithme CCF permet de trouver efficacement les composants connexes dans des graphes de grande taille en utilisant le paradigme MapReduce.

Les implémentations Scala RDD et GraphX offrent les meilleures performances, avec un avantage significatif pour GraphX qui s'améliore avec la taille du graphe.

Recommandations

- Privilégier GraphX pour les graphes de grande taille
- Utiliser Scala RDD pour une meilleure stabilité