

# Finding Connected Components in Graph








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

Jean-Christophe HAMARD | Dina HOUPLIER

25 Septembre 2025

# Plan

---

-  Contexte et définitions
-  Objectifs de l'étude
-  Algorithme CCF (Connected Component Finder)
-  Jeux de données et méthodologie
-  Implémentations réalisées
-  Résultats comparatifs et analyse des performances
-  Conclusion et perspectives

# Contexte et définitions




## Qu'est-ce qu'un graphe ?

Structure composée de nœuds (ou sommets) et de liens (ou arêtes) entre eux.

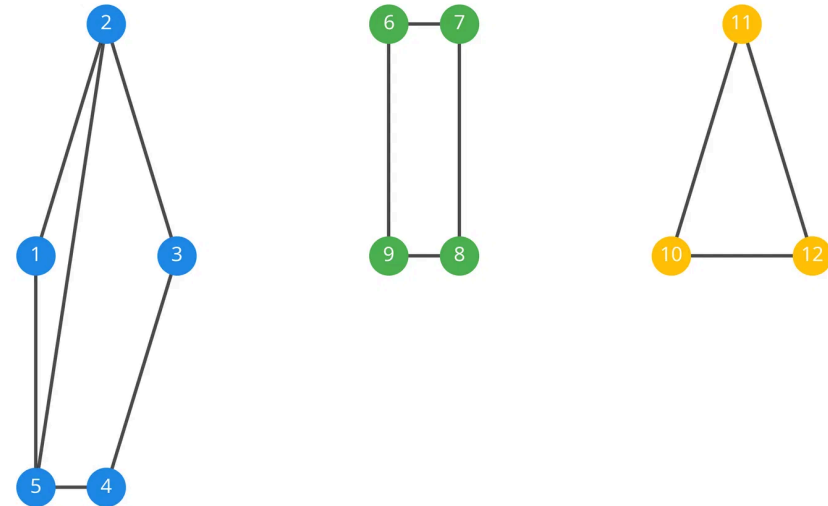
## Qu'est-ce qu'un composant connexe ?

Groupe de nœuds où chaque nœud est accessible à partir des autres via des liens.

## Applications pratiques

-  Réseaux sociaux (groupes d'amis, communautés)
-  Recherche de doublons dans des bases de données
-  Traitement d'images et segmentation

Exemple de graphe avec trois composants connexes



Composant connexe 1

Composant connexe 2

Composant connexe 3

# Objectifs de l'étude

---

## Méthode scalable

Proposer une méthode efficace et hautement scalable pour trouver tous les composants connexes d'un graphe, adaptée aux très grands volumes de données.



## Utilisation de MapReduce

Exploiter le modèle de programmation MapReduce pour le traitement parallèle de données massives via Hadoop et Spark.

### Cas d'usage concret

Moteur de recherche de personnes basé sur des données personnelles américaines (mariages, réseaux sociaux, propriétés, etc.), totalisant des milliards d'enregistrements. L'objectif est de regrouper toutes les données liées à une même personne, même si ces données proviennent de sources différentes.

# Algorithme CCF - Principe

## Vue d'ensemble

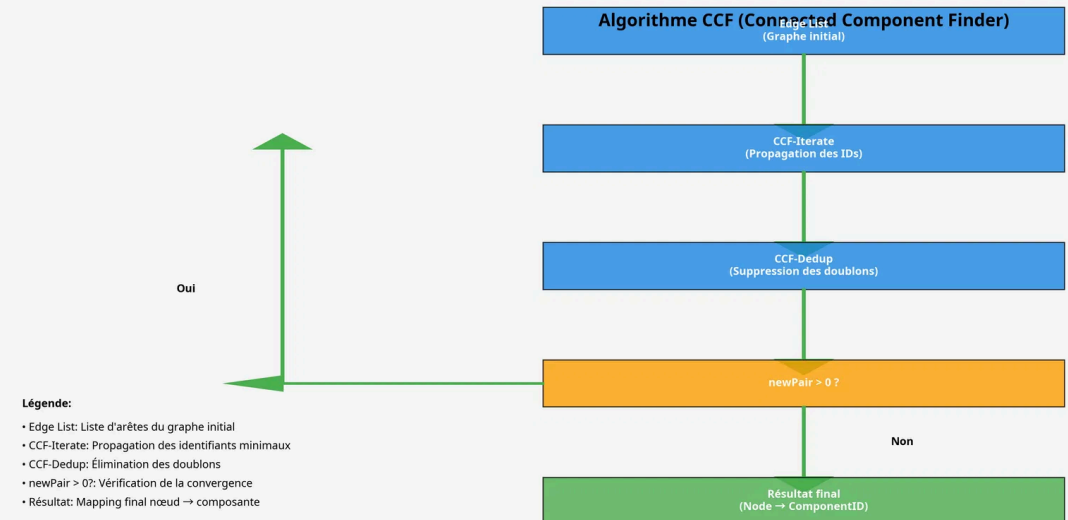
CCF (Connected Component Finder) est un algorithme MapReduce itératif pour identifier les composants connexes dans des graphes de grande taille.

## Étapes principales

- 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, c'est-à-dire quand tous les nœuds d'un même composant partagent le même identifiant minimal.



# Algorithme CCF - Détails

## CCF-Iterate

### → Transformation des arêtes

Chaque arête (key, value) est transformée en deux sorties : (key, value) et (value, key) pour rendre le graphe non orienté.

### → Propagation des identifiants

Pour chaque nœud, on cherche le plus petit identifiant parmi lui-même et ses voisins, puis on le propage.

```
// Pseudo-code simplifié Map(key, value): Emit(key, value) Emit(value, key) Reduce(key, values): min = key for each v in values: if v < min: min = v for each v in values: Emit(v, min)
```

## CCF-Dedup

### → Élimination des doublons

Supprime les paires (nœud, minID) dupliquées pour réduire le volume de données et améliorer les performances.

```
// Pseudo-code simplifié Map(key, value): Emit((key, value), null) Reduce(key, values): Emit(key.entity1, key.entity2)
```

## Optimisation par tri secondaire

Le tri secondaire permet d'obtenir directement le plus petit voisin en première position, évitant ainsi de parcourir toute la liste des voisins pour trouver le minimum.

### ✓ Critère d'arrêt

L'algorithme s'arrête lorsqu'aucun nouveau couple (nœud, minID) n'est généré, indiquant la convergence.

# Jeux de données

Pour cette étude, nous avons utilisé quatre fichiers CSV représentant des graphes de différentes tailles :

Graphe	Nœuds	Arêtes	Fichier
G1	996	3 000	G1_1k.csv
G2	4 983	15 000	G2_5k.csv
G3	7 971	24 000	G3_8k.csv
G4	9 979	30 000	G4_10k.csv

Structure des fichiers CSV :

```
source,target
444,683
129,356
872,195
...
```

# Méthodologie

---

## Environnement d'exécution

- Cluster Spark distribué pour les exécutions parallèles
- Environnement de développement IntelliJ pour Scala
- Jupyter Notebooks pour les implémentations Python

## Métriques d'évaluation

- Temps d'exécution total (en secondes)
- Nombre d'itérations pour atteindre la convergence
- Scalabilité avec l'augmentation de la taille du graphe

## Approches d'implémentation

### PySpark RDD

Implémentation en Python utilisant l'API RDD de Spark pour les transformations de données distribuées.

### PySpark DataFrame

Utilisation de l'API DataFrame de Spark en Python, avec des opérations de type SQL.

### Scala RDD

Implémentation en Scala avec l'API RDD, tirant parti des optimisations de la JVM.

### Scala DataFrame

Utilisation de l'API DataFrame de Spark en Scala, avec des opérations de type SQL.

### Scala GraphX

Utilisation de la bibliothèque GraphX de Spark, spécialement conçue pour l'analyse de graphes.



# Implémentation PySpark RDD

## Principe

Implémentation de l'algorithme CCF en utilisant les Resilient Distributed Datasets (RDD) de PySpark, qui permettent un traitement distribué des données.

```
# Extrait de code clé def ccf_iterate(edges): # Rendre le graphe non orienté bidirectional = edges.flatMap(lambda e: [(e[0], e[1]), (e[1], e[0])]) # Grouper par nœud pour obtenir les voisins adjacency = bidirectional.groupByKey().mapValues(list) # Pour chaque nœud, trouver le plus petit ID # et le propager à ses voisins def update_component(node, neighbors): min_id = min([node] + neighbors) return [(node, min_id)] + [(n, min_id) for n in neighbors] new_components = adjacency.flatMap( lambda x: update_component(x[0], x[1]) ) # Supprimer les doublons return new_components.distinct()
```

### ✓ Avantages

- Implémentation simple et intuitive
- Bonne stabilité et convergence rapide (5-6 itérations)
- Contrôle fin sur les transformations de données

## Processus d'exécution

L'algorithme est exécuté de manière itérative jusqu'à convergence, en utilisant un accumulateur pour détecter quand aucun nouveau couple (nœud, minID) n'est généré.

```
# Boucle principale def ccf_correct_implementation(edges_rdd, max_iters=20): # Initialisation components = edges_rdd.flatMap( lambda e: [(e[0], e[0]), (e[1], e[1])] ).distinct() for i in range(max_iters): # Propagation des étiquettes new_components = ccf_iterate(edges_rdd) # Vérification de la convergence if components.subtract(new_components).count() == 0: return components, i+1 components = new_components return components, max_iters
```

### ✗ Inconvénients

- Temps d'exécution modéré (84-158 secondes)
- Moins performant que l'implémentation Scala RDD
- Overhead lié à l'interprétation Python

## Résultats

Graphe	Itérations	Temps (s)
G1	5	84.542
G2	6	162.019
G3	6	156.463
G4	6	158.003

# Implémentation PySpark DataFrame

## Principe

Utilisation des DataFrames Spark, optimisés pour les requêtes de type SQL, avec des opérations comme join, groupBy, agg, et withColumn pour exploiter le moteur d'optimisation de Spark SQL.

```
# Chargement des données edges_df =
spark.read.csv("data/G1_1k.csv", header=True,
schema=schema) # Exemple de logique d'itération
updated_labels = current_and_new_labels \ .withColumn(
"new_label", least(col("current_component_id"),
col("propagated_id"))) \ .select(col("node"),
col("new_label").alias("component_id"))
```

### ✓ Avantages

- Syntaxe concise et déclarative
- Bénéficie des optimisations du moteur Spark SQL
- API plus intuitive pour les utilisateurs SQL

## Résultats

Graphe	Nœuds	Arêtes	Itérations	Temps (s)
G1	996	3000	8	135.066
G2	4983	15000	8	100.337
G3	7971	24000	9	269.291
G4	9979	30000	8	108.962

### ✗ Inconvénients

- Temps d'exécution instables dus aux jointures répétées
- Nombre d'itérations plus élevé que la version RDD
- Performances variables selon la topologie du graphe

## Convergence

L'algorithme converge systématiquement entre 8 et 9 itérations pour les quatre graphes, légèrement plus que l'implémentation RDD (5-6 itérations).

# Implémentation Scala RDD

## Principe de l'implémentation

L'implémentation en Scala avec RDD (Resilient Distributed Datasets) offre une approche performante et optimisée pour l'algorithme CCF, tirant parti de la JVM et des optimisations de Spark.

```
// Chargement des données val edges = sc.textFile(path)
.filter(!_startsWith("source")).map(_.split(","))
.map(arr => (arr(0).toInt, arr(1).toInt)) // Dé-
duplication des arêtes val dedupEdges = edges.map {
case (a, b) => if (a < b) (a, b) else (b, a)
}.distinct() // Initialisation des composantes var
components = dedupEdges.flatMap { case (a, b) =>
Seq((a, a), (b, b)) }.reduceByKey((x, _) => x) //
Boucle principale var newComponents = components var
iteration = 0 do { components = newComponents //
Propagation des identifiants minimaux newComponents =
dedupEdges.join(components).map { case (_, (neighbor,
compId)) => (neighbor, compId) }.union(components)
.reduceByKey(math.min) iteration += 1 } while
(components.count() != newComponents.count())
```

### ✓ Avantages

- Performance exceptionnelle (~2.5 secondes)
- Stabilité des temps d'exécution
- Optimisations de la JVM et de Spark
- Idéal pour le traitement distribué

### ! Inconvénients

- Nombre d'itérations légèrement plus élevé (8-10)
- Syntaxe plus complexe que Python

### Performance remarquable

L'implémentation Scala RDD est environ 40 à 60 fois plus rapide que l'équivalent PySpark RDD, et jusqu'à 300 fois plus rapide que Scala DataFrame, ce qui en fait la solution la plus efficace pour l'algorithme CCF sur des graphes de taille moyenne.

# Implémentation Scala DataFrame et GraphX

---

## Scala DataFrame

Implémentation utilisant l'API DataFrame de Spark en Scala, avec des opérations comme join, groupBy et agg.

```
// Extrait de code Scala DataFrame val updatedLabels =  
currentAndNewLabels .withColumn( "new_label",  
least(col("current_component_id"),  
col("propagated_id")) ) .select( col("node"),  
col("new_label").alias("component_id") )
```

### Avantages et inconvénients

- ✓ Bénéficie des APIs haut niveau Scala
- ✗ Très lent : performances 100-300× plus faibles que RDD Scala
- ✗ Temps d'exécution élevé : 350-917 secondes

## Scala GraphX

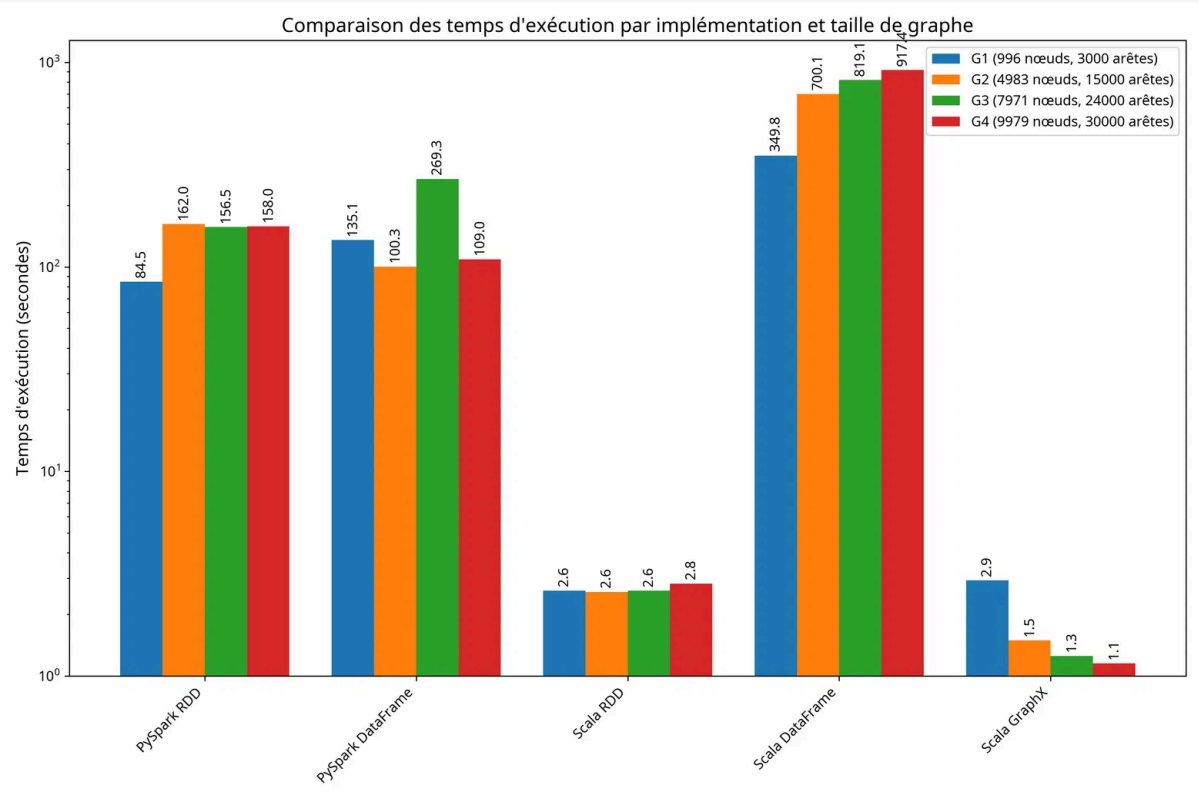
Implémentation utilisant GraphX, la bibliothèque d'analyse de graphes native de Spark, spécifiquement optimisée pour les algorithmes sur les graphes.

```
// Extrait de code Scala GraphX import  
org.apache.spark.graphx._ // Conversion en graphe  
GraphX val graph = Graph(vertices, edges) // Calcul  
des composantes connexes en une ligne val cc =  
graph.connectedComponents() // Résultat val results =  
cc.vertices
```

### Avantages et inconvénients

- ✓ Performances exceptionnelles : 1.15-2.93 secondes
- ✓ Implémentation en une seule ligne de code
- ✓ Optimisé pour les calculs de graphes à grande échelle





# Résultats comparatifs

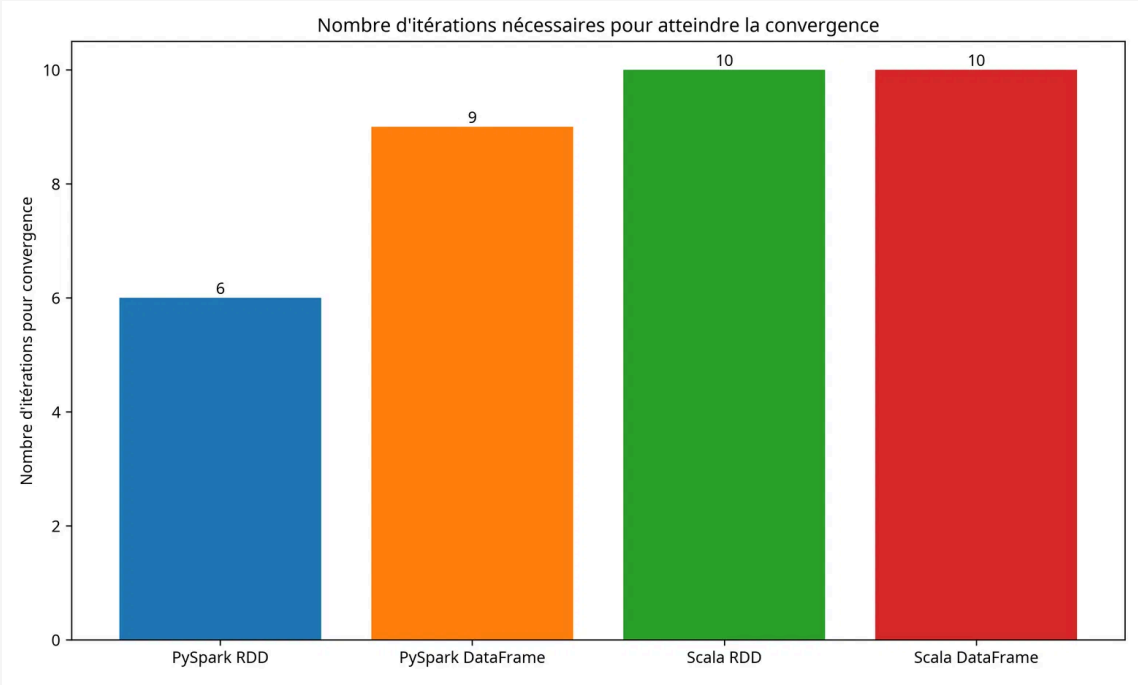


Version	Langage	API	Itérations	Temps (G1)	Temps (G4)	Avantage principal
PySpark RDD	Python	RDD	5-6	84.5 s	158.0 s	Simple, stable
PySpark DataFrame	Python	DataFrame	8-9	135.1 s	109.0 s	Code compact
Scala RDD	Scala	RDD	8-10	2.6 s	2.8 s	Ultra performant
Scala DataFrame	Scala	DataFrame	8-10	349.8 s	917.4 s	Facile à intégrer
Scala GraphX	Scala	GraphX	N/A	2.9 s	1.2 s	Solution optimale

# Analyse des performances

## Facteurs influençant les performances





-  **Langage d'implémentation** : Scala (JVM) offre des performances nettement supérieures à Python pour les opérations intensives.
-  **API utilisée** : Les RDDs sont plus efficaces que les DataFrames pour les algorithmes itératifs comme CCF.
-  **Bibliothèques spécialisées** : GraphX, conçu spécifiquement pour les graphes, offre des optimisations natives pour les composantes connexes.
-  **Topologie du graphe** : La structure du graphe influence le nombre d'itérations nécessaires pour la convergence.



## Forces et faiblesses par approche

Approche	Forces	Faiblesses
PySpark RDD	Simplicité, stabilité	Performance moyenne
PySpark DataFrame	API intuitive, concise	Performances variables
Scala RDD	Très haute performance	Syntaxe plus complexe
Scala DataFrame	API haut niveau	Très lent pour CCF
Scala GraphX	Solution optimale, rapide	Spécifique aux graphes

### Recommandations selon les cas d'usage

-  **Petits graphes, prototypage rapide** : PySpark RDD ou DataFrame pour la facilité de développement
-  **Graphes moyens à grands** : Scala RDD pour un excellent équilibre performance/flexibilité
-  **Graphes très grands ou en production** : Scala GraphX pour des performances optimales
-  **Graphes géants (milliards de nœuds)** : MapReduce natif pour une scalabilité extrême

# Conclusion et perspectives

---

## Synthèse des résultats

Notre étude comparative des différentes implémentations de l'algorithme CCF (Connected Component Finder) a démontré des écarts significatifs de performance selon les approches utilisées. L'algorithme CCF s'est révélé efficace pour identifier les composants connexes dans des graphes de différentes tailles.

## Recommandations selon les cas d'usage

- ✓ **Scala GraphX** est la solution optimale pour les graphes de toute taille, offrant des performances exceptionnelles et une implémentation simplifiée.
- ✓ **Scala RDD** est une excellente alternative lorsque GraphX n'est pas disponible, avec des performances très proches.
- ✓ **PySpark RDD** offre un bon compromis entre facilité d'implémentation et performance pour les équipes travaillant principalement en Python.
- ✓ **MapReduce natif** reste pertinent pour des graphes extrêmement volumineux nécessitant une infrastructure Hadoop dédiée.

## Perspectives d'amélioration

Les travaux futurs pourraient explorer l'optimisation des implémentations DataFrame, l'application à des graphes encore plus volumineux, et l'intégration avec d'autres algorithmes d'analyse de graphes pour des cas d'usage plus complexes comme la détection de communautés ou l'analyse de centralité.

**Merci pour votre attention !**