République Algérienne Démocratique et Populaire
Université Abou Bekr Belkaïd – Tlemcen
Faculté des Sciences
Département d'Informatique

Mémoire de fin d'études

Pour l'obtention du diplôme de Master en Informatique

Option : Réseaux et Systèmes Distribués (RSD)

Thème

Vers un cadre de distribution efficace pour les systèmes de gestion de données RDF distribué

Réalisé par :

- Sabah Esmaa Saidi
- Aya Abdellaoui

Présenté le 23 Juin 2025 devant le jury composé de :

M. Badr BENMAMMAR	Président
M. Mohammad MANA	Examinateur
M. Houcine MATALLAH	Encadrant
M. Boumediene SAIDI	Co-encadrant

Année universitaire : 2024--2025

Remerciements

ٱلْحَمْدُ لِلّٰهِ وَٱلشُّكْرُ لِلّٰهِ

Alhamdoulilah, toute louange revient à ***Allah***, qui nous a permis datteindre cette étape importante de notre parcours, qui nous a accordé la patience dans les moments difficiles, la force de persévérer, et qui nous a entourés de personnes bienveillantes tout au long de cette aventure académique.

Nous adressons nos plus profonds remerciements à nos encadrants pour leurs conseils précieux tout au long de ce projet.

Nos sincères remerciements vont à notre encadrant Monsieur **Houcine Matallah** pour sa disponibilité, ses conseils précieux et son accompagnement bienveillant.

Une mention toute particulière à notre co-encadrant Monsieur **Saidi Boumediene**, pour son implication, son écoute, sa rigueur et ses encouragements constants, qui ont fortement contribué à la réussite de ce travail.

Nous remercions également Monsieur **Badr Benmammar** pour avoir accepté de présider le jury, ainsi que Monsieur **Mohammad Mana** pour avoir accepté den être examinateur.

Notre plus profonde gratitude va à nos **parents** pour leur amour inconditionnel, leurs conseils avisés et leur soutien moral et financier constant, qui nous ont permis de poursuivre nos études et de mener à bien ce mémoire.

Enfin, nous remercions chaleureusement nos **amis** et **collègues** pour leur soutien constant, leurs échanges enrichissants et leurs encouragements tout au long de cette aventure académique.

Dédicace

À mes chers **Parents**,
pour leur amour inconditionnel, leurs innombrables sacrifices et leur soutien indéfectible, véritables piliers de mon parcours.

À mes deux frères, **Boumediene** et **Amine**,
dont la présence constante et réconfortante ma accompagnée avec douceur et bienveillance tout au long de cette aventure.

À mes encadrants,
Monsieur Houcine Matallah et **Monsieur Saidi Boumediene**,
pour leurs conseils précieux, leur encadrement attentif et leur disponibilité constante,
qui ont largement contribué à la réussite de ce travail.

À mes meilleures amies,
Lamia, **Douaa** et **Nardjess**, mes sœurs de cœur,
pour leur présence fidèle, leur bienveillance précieuse et leur soutien constant, qui ont été pour moi une source inestimable de force et de réconfort.

À **Fatima**, **Amel**, **Hadjer** et **Khadidja**,
avec qui jai partagé des fous rires et des souvenirs inoubliables, gravés à jamais dans ma mémoire.

Enfin, à ma binôme, **Aya Abdellaoui**,
avec qui j'ai partagé cette expérience avec sérieux, complicité et complémentarité.

Saidi Sabah Esmaa

Dédicace

À mes chers **Parents**,

En témoignage de mon amour infini, de mon respect profond et de ma reconnaissance éternelle.
Merci pour vos sacrifices silencieux, votre patience inlassable et vos prières constantes.
Votre amour est la lumière qui a guidé chacun de mes pas.

À ma **famille** et à mes **amis**,

Pour votre présence réconfortante dans les moments de doute,
vos encouragements sincères et votre soutien indéfectible.

À Monsieur **Houcine Matallah,**

Mon encadrant, pour son accompagnement rigoureux,
sa disponibilité, ses conseils éclairés et sa confiance précieuse.

À Monsieur **Saidi Boumediene,**

Mon co-encadrant, pour sa bienveillance, sa patience
et ses orientations précieuses.

À **Saidi Sabah,**

Ma collaboratrice, pour cette complicité humaine et intellectuelle,
ce chemin partagé, riche d'échanges et de complémentarité.

À toutes celles et ceux qui, de près ou de loin, ont contribué à la réalisation de ce travail, Recevez ici l'expression sincère de ma gratitude.

Abdellaoui Aya

Résumé

L'explosion des données est devenue le centre des préoccupations de plusieurs secteurs, économiques, scientifiques et sociétaux. Dans ce contexte, de nombreux défis sont lancés dans le but de gérer la collection, le traitement et l'exploration de ces données. Les travaux de ce projet de fin d'études s'intègrent dans le cadre du système de gestion de données RDF distribué PQDAG, développé au sein du laboratoire LIAS (ISAE-ENSMA). Notre objectif consistait à analyser les limites du mécanisme de transfert initial de PQDAG, puis à concevoir une solution plus modulaire et performante. Ainsi, nous avons proposé un nouveau framework baptisé **EGFDT** (Enhanced Generic Framework for Data Transfer), permettant de gérer les transferts de données de manière efficace et extensible. Ce framework intègre plusieurs stratégies de transfert efficaces, notamment le *broadcast* et le *shuffle*, qui sont omniprésentes dans le système PQDAG. Toutefois, sa conception générique lui permet de s'intégrer facilement à d'autres systèmes distribués présentant une architecture similaire à celle de PQDAG. L'efficacité de notre solution a été validée expérimentalement à travers une série de tests réalisés sur une infrastructure OpenStack.

Mots-clés : Système de gestion de données, Transfert de données, EGFDT, Scalabilité.

Table des matières

CHAPITRE 1

INTRODUCTION GÉNÉRALE

1.1 Introduction

Le *Resource Description Framework* (RDF) s'est imposé comme la pierre angulaire de la représentation des graphes de connaissances sur le Web. Chaque relation est encodée sous la forme d'un triplet S, P, O : S pour sujet, P pour prédicat, O pour objet. Cette structure élémentaire permet néanmoins de modéliser des connaissances très riches, formant des graphes potentiellement vastes et fortement interconnectés.

Au cours de la dernière décennie, la taille des graphes RDF a connu une croissance exponentielle. En 2009, le LOD (Linked Open Data) cloud contenait environ 100 jeux de données. En 2025, il dépasse les 1500 jeux interconnectés. La taille est passée de quelques millions à plusieurs milliards de triplets par jeu de données [1]. Des graphes de taille industrielle sont désormais courants dans des domaines aussi variés que les sciences de la vie (Bio2RDF [2], UniProt [3]), les encyclopédies ouvertes (DBpedia [4], Wikidata [5]), le balisage commercial du Web (schema.org), les données gouvernementales ouvertes, le patrimoine culturel (Europeana), ainsi que les projets scientifiques et l'IoT. Cette croissance spectaculaire tient à la maturité des standards W3C, à la démocratisation du cloud et à l'essor des graphes de connaissances pour l'intelligence artificielle explicable [9].

Cette croissance a stimulé la recherche et le développement de systèmes de gestion RDF, aussi bien centralisés (des systèmes s'exécutant sur une seule machine, tels que Virtuoso [6], Blazegraph [18] ou Jena TDB [15]) que distribués (des systèmes nécessitant un cluster de machines pour fonctionner, comme TriAD [16], S2RDF [17] ou AdPart [1]). Leur objectif commun est de garantir une exécution efficace des requêtes SPARQL, tout en assurant la scalabilité et la robustesse face à l'augmentation constante du volume de données.

Un trait déterminant des moteurs RDF modernes réside dans le partitionnement du graphe, qui consiste à diviser l'ensemble des triplets en fragments autonomes, répartis sur un ou plusieurs nœuds de calcul. Cette opération vise à limiter les accès distants, réduire les échanges inter-machines et favoriser le parallélisme dans l'exécution des requêtes. Plusieurs stratégies de partitionnement ont été proposées dans la littérature : par exemple, METIS [10] pour minimiser les coupures d'arêtes (edge-cut), MPC [14] pour réduire les coupures sur les propriétés (property-cut), ou encore des approches plus simples comme le hachage ou la répartition en *round-robin*.

L'objectif principal de ces méthodes est de minimiser le nombre de jointures entre partitions, c'est-à-dire éviter les situations où une requête doit accéder à des données réparties sur plusieurs machines. Cela permet de réduire le coût des échanges réseau, qui constitue souvent un facteur limitant dans les systèmes distribués. Hélas, même si un bon partitionnement permet de diminuer le volume global des transferts, le moindre échange réseau introduit une latence significative qui peut annuler les bénéfices attendus.

Malgré les centaines de travaux publiés sur le partitionnement des graphes RDF, la communauté des bases de données et plus particulièrement celle spécialisée dans le RDF accorde encore peu d'attention aux mécanismes concrets de transfert des données entre les nœuds. Or, il ne suffit pas de bien répartir les données : il faut aussi optimiser la façon dont ces données sont échangées au moment de l'exécution.

[1] https://lod-cloud.net/
[2] https://bio2rdf.org/
[3] https://www.uniprot.org/
[4] https://www.dbpedia.org/
[5] https://www.wikidata.org/

Dans ce contexte, on peut distinguer trois grandes catégories de systèmes en fonction de leur stratégie de transfert de données : (1) les systèmes reposant sur des plateformes MPP (Massively Parallel Processing) comme Spark ou Hadoop, qui utilisent des échanges synchrones et centralisés (par exemple via des opérations de *shuffle*) ; (2) les systèmes fondés sur MPI (Message Passing Interface), qui favorisent une communication asynchrone point-à-point, plus fine et potentiellement plus efficace ; et enfin (3) une catégorie de systèmes qui implémentent leurs propres stratégies de transfert optimisées, mais qui restent souvent mal documentées dans la littérature, rendant leur reproduction ou leur analyse difficile. Cette dernière catégorie représente un champ de recherche prometteur, où l'optimisation fine des communications réseau pourrait jouer un rôle déterminant dans la prochaine génération de moteurs RDF distribués.

1.2 Problématique

Les systèmes basés sur Spark et Hadoop, conçus initialement pour le calcul massivement parallèle, ont été largement adoptés pour exécuter des traitements distribués sur de grands volumes de données. Toutefois, leur intégration dans des systèmes de gestion de données RDF n'est pas directe et nécessite une adaptation spécifique. Cette adaptation ajoute non seulement une couche d'abstraction supplémentaire, mais peut également engendrer des coûts computationnels et de mémoire non négligeables. En effet, Spark repose fortement sur la mémoire principale pour atteindre des performances élevées. Or, dans des environnements contraints en ressources ou pour des cas d'usage plus simples, cette dépendance peut constituer une limite, car le déploiement d'un cluster Spark exige des ressources considérables, tant en termes d'infrastructure que de configuration.

De plus, Spark s'appuie sur un modèle de synchronisation globale, essentiel pour garantir la cohérence et la tolérance aux pannes, mais ce mécanisme s'avère coûteux. Il impose des barrières de synchronisation fréquentes entre les nœuds et engendre une latence importante, même lors de transferts de faible volume. L'exemple typique est l'opération de *shuffle*, qui redistribue les données entre les partitions. Cette étape, indispensable dans de nombreuses requêtes SPARQL, devient rapidement un goulet d'étranglement.

À lopposé, MPI (Message Passing Interface) propose un modèle de communication asynchrone, où les processus échangent directement des messages sans coordination globale. Ce paradigme permet de réduire significativement la latence, en supprimant la nécessité de synchronisation entre les tâches. Cependant, cette efficacité a un prix : le modèle MPI suppose un environnement fiable, où les pannes sont rares. Il n'intègre pas nativement de mécanismes de résilience, comme la journalisation ou la réplication automatique. En cas de défaillance d'un nœud, la reprise est manuelle, voire impossible, ce qui rend l'approche difficile à généraliser à des systèmes de production à grande échelle. De plus, la programmation avec MPI est plus complexe, car elle exige une gestion explicite des communications, et une synchronisation fine entre les processus, ce qui accroît la complexité de l'implémentation.

En parallèle, certains systèmes de gestion de données RDF implémentent leurs propres mécanismes de transfert de données, souvent conçus de manière *ad hoc* pour répondre à des besoins spécifiques de performance ou d'optimisation. Néanmoins, ces mécanismes sont rarement décrits en détail dans les publications, et peu de documentation technique est disponible. Cela révèle un angle mort dans la recherche sur les triplestores RDF distribués : alors que l'efficacité du partitionnement est abondamment étudiée, le coût réel des transferts réseau reste sous estimé, alors qu'il peut représenter un facteur déterminant

dans les performances globales d'un système.

C'est dans ce contexte de double contrainte performance et résilience que s'inscrit la contribution de l'équipe PQDAG, rattachée au laboratoire LIAS de l'école d'ingénieurs ISAE-ENSMA, et collaborant activement avec le laboratoire LRIT de l'Université de Tlemcen pour le développement de solutions RDF à grande échelle. L'équipe a conçu un système RDF distribué, baptisé PQDAG, visant à concilier les avantages des approches Spark et MPI. D'un côté, une architecture synchrone basée sur le modèle BSP [19] (Bulk Synchronous Parallel) assure la tolérance aux pannes et la cohérence des traitements, à l'image des systèmes MPP tels que Spark. De l'autre, des mécanismes de communication rapides et légers, directement inspirés de l'approche asynchrone de MPI, permettent de réduire la latence des transferts de données. Le système PQDAG se positionne ainsi comme une solution hybride et pragmatique, adaptée aux environnements contraints, tout en garantissant une exécution efficace des requêtes RDF à grande échelle.

Cependant, les mécanismes de transfert actuellement implémentés dans PQDAG présentent encore certaines limites en termes d'efficacité. Des optimisations supplémentaires sont nécessaires pour affiner la gestion des communications, réduire les goulots d'étranglement et atteindre un meilleur compromis entre performances et robustesse.

1.3 Objectifs

L'objectif de ce projet de fin d'études est, dans un premier temps, de réaliser un état de l'art des systèmes de gestion de données RDF appartenant à deux grandes catégories : ceux basés sur des architectures MPP, comme Spark, et ceux reposant sur MPI, en se concentrant sur les stratégies de transfert de données et leur fonctionnement. Dans un second temps, il s'agira d'analyser en profondeur les mécanismes de transfert déjà implémentés dans PQDAG, ce qui nécessitera un travail important de mise à niveau du système. L'étape suivante consistera à proposer des améliorations ciblées pour ces mécanismes de transfert, tout en veillant à garantir modularité et indépendance vis-à-vis des autres composants du système. Enfin, une validation expérimentale sera menée à l'aide d'une infrastructure OpenStack, en déployant et en orchestrant plusieurs machines virtuelles pour simuler des environnements distribués réels.

1.4 Organisation du manuscrit

Le chapitre 2 présente tout d'abord un état de l'art des systèmes de gestion de données RDF distribués, en mettant l'accent sur les protocoles de transfert de données sur lesquels ces systèmes s'appuient. Ensuite, le chapitre 3 décrit la solution existante, PQDAG, en soulignant les limites actuelles de son mécanisme de transfert. Le chapitre 4 s'intéresse alors aux besoins spécifiques de l'équipe PQDAG en matière de transfert de données, et détaille la solution proposée, accompagnée d'une validation expérimentale. Par la suite, le chapitre 5 présente les outils, les langages et les technologies utilisés pour mettre en œuvre notre solution, ainsi que l'exécution concrète des différents composants, illustrée par des captures d'écran. Enfin, le dernier chapitre conclut notre travail et ouvre plusieurs perspectives d'amélioration pour la suite du projet.

CHAPITRE 2

ÉTAT DE L'ART

2.1 Introduction

Dans ce chapitre, nous présentons l'état de l'art des systèmes de gestion de données RDF, en mettant l'accent sur l'aspect du transfert de données, un critère souvent sous-estimé dans les classifications classiques, bien qu'il joue un rôle crucial dans les environnements distribués. La majorité des systèmes existants s'appuie sur des frameworks de calcul massivement parallèle tels qu'Apache Spark, qui adoptent un modèle synchrone. À l'opposé, d'autres approches reposent sur MPI (Message Passing Interface), un modèle asynchrone qui privilégie la communication directe entre les processus pour réduire la latence. Il existe également des systèmes qui implémentent leur propre stratégie de transfert de données, développée *from scratch*, mais ces approches restent relativement rares et peu documentées.

Ce chapitre est structuré comme suit : nous présentons d'abord les principes de fonctionnement de Spark et de MPI, puis nous décrivons quelques systèmes de gestion de données RDF représentatifs basés sur ces deux paradigmes. Enfin, nous discutons des limites propres à chaque catégorie et mettons en lumière la nécessité d'une solution hybride qui offrirait un compromis entre la robustesse du modèle synchrone (Spark) et la rapidité du modèle asynchrone (MPI).

2.2 Apache Spark

Apache Spark est un framework de traitement distribué open-source, largement adopté dans le domaine du Big Data pour sa capacité à exécuter efficacement des traitements sur de très grands volumes de données. Il a été initialement développé en 2009 par Matei Zaharia [1] au sein du laboratoire AMP de l'Université de Californie à Berkeley. Spark a été conçu pour surmonter les limitations du modèle MapReduce proposé par Hadoop [2], notamment en ce qui concerne la latence élevée et la faible efficacité des traitements itératifs et interactifs. Contrairement à MapReduce, Spark adopte un modèle de calcul en mémoire, ce qui permet de réduire considérablement les accès disque, souvent coûteux en temps et en ressources.

2.2.1 Architecture de Spark

L'architecture de Spark repose sur une abstraction de haut niveau appelée RDD (Resilient Distributed Dataset). Les RDD représentent des collections de données réparties sur plusieurs nœuds du cluster, immuables et tolérantes aux pannes. Ce modèle offre une grande flexibilité pour construire des chaînes de traitements complexes, tout en assurant une exécution performante et résiliente.

Le framework Spark est organisé autour d'un noyau central et de plusieurs bibliothèques spécialisées qui étendent ses fonctionnalités :

— **Spark Core**, le cœur du système, assure la gestion de l'exécution distribuée, la planification des tâches, le suivi des ressources et la tolérance aux pannes.
— **Spark SQL**, qui permet de manipuler des données structurées à l'aide de requêtes SQL ou du modèle de données *DataFrame*.

[1] https://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf
[2] https://research.google.com/archive/mapreduce.html

— **Spark Streaming**, conçu pour le traitement de flux de données en temps réel, avec une sémantique de micro-batching.
— **MLlib**, la bibliothèque dédiée à l'apprentissage automatique distribué, offrant des algorithmes de classification, de régression, de clustering et d'autres encore.
— **GraphX**, pour la modélisation, la transformation et l'analyse de graphes à grande échelle.

L'ensemble de ces composants s'appuie sur Spark Core, comme le montre la Figure 2.1, formant un écosystème intégré qui permet de traiter des données de nature variée (structurées, non structurées, en flux, etc.) de manière fluide et performante dans un seul et même environnement distribué.

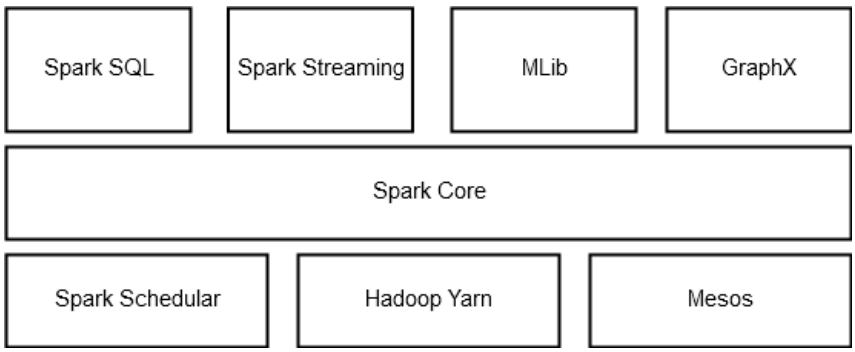

FIG. 2.1 – L'ecosystéme de Spark

2.2.2 Gestion des transferts de données dans Spark

Dans une architecture distribuée comme Spark, l'efficacité des transferts inter-nœuds est déterminante pour les performances globales. Spark orchestre principalement deux mécanismes de transfert fondamentaux : le ***shuffle*** et le ***broadcast***, chacun répondant à des besoins spécifiques de redistribution des données dans un cluster.

Le mécanisme de Shuffle

Le *shuffle* constitue l'une des opérations les plus coûteuses et les plus critiques dans l'exécution d'un programme Spark. Il est déclenché dès qu'une opération requiert une redistribution globale des données entre les partitions du cluster, comme dans les cas suivants : `groupBy`, `reduceByKey`, `join`, `distinct` ou encore `sortByKey`. Le processus de shuffle suit quatre étapes principales :

1. **Écriture intermédiaire** : chaque tâche *map* produit des blocs de sortie pour chaque partition cible et les écrit localement sur le disque.
2. **Sérialisation et compression** : les blocs sont sérialisés et éventuellement compressés pour réduire la taille du transfert.
3. **Transfert réseau** : les données sont transmises aux tâches *reduce* exécutées sur d'autres nœuds.
4. **Lecture et combinaison finale** : chaque tâche *reduce* lit les blocs qui lui sont destinés et les agrège.

Ce mécanisme implique des coûts importants : (1) Accès disques multiples avec une phase décriture en sortie de la tache map et une phase de lecture avant la tache de réduction (combinaison des résultats). (2) Consommation réseau massive, surtout lorsque le

volume de données est important et que les partitions sont nombreuses, ce qui correspond généralement à un nombre élevé de machines dans le cluster.

Spark a connu plusieurs évolutions de son moteur de shuffle, intégrant des algorithmes différents selon les versions :

Type de Shuffle	**Description**
Hash-Based Shuffle (legacy)	Utilisé par défaut jusquà Spark 1.1 Chaque tâche produit un fichier de sortie par tâche cible, ce qui génère un très grand nombre de fichiers intermédiaires et surcharge le système de fichiers.
Sort-Based Shuffle	Introduit à partir de Spark 1.2 et utilisé par défaut depuis Spark 1.6. Trie les enregistrements par clé avant leur envoi, réduisant ainsi le nombre de fichiers produits. Il s'appuie sur des algorithmes de tri partiel (ex. quick sort, merge sort) et une gestion optimisée de la mémoire tampon.
Tungsten Shuffle (Project Tungsten)	Apparu avec Spark 1.4 dans le cadre du projet Tungsten. Optimise la gestion mémoire (usage de la mémoire off-heap) et améliore la performance en réduisant les coûts de sérialisation. Il est intégré au moteur Catalyst de Spark SQL.

TAB. 2.1 – Résumé des principaux mécanismes de shuffle dans Apache Spark

Ces évolutions visent à réduire les coûts liés au shuffle, en optimisant l'usage de la mémoire, la sérialisation et le nombre d'objets manipulés sur disque.

Le mécanisme de Broadcast

Le *broadcast* constitue une alternative légère au shuffle, utilisée lorsque de petits jeux de données doivent être partagés globalement avec tous les nœuds du cluster. C'est notamment le cas dans les jointures entre une grande table et une petite table de dimension (ex. : `broadcast join`). Son fonctionnement suit trois principes fondamentaux :

1. **Diffusion unique** : la donnée est sérialisée une seule fois et diffusée à tous les exécuteurs concernés.
2. **Stockage en mémoire** : chaque nœud conserve une copie locale en mémoire de la donnée diffusée, réduisant ainsi les latences d'accès.
3. **Accès local** : chaque tâche accède directement à l'objet diffusé sans passer par le réseau.

Spark utilise une stratégie appelée *Torrent Broadcast Algorithm*, inspirée des réseaux P2P comme BitTorrent, pour diffuser efficacement les objets broadcastés : le driver divise l'objet à diffuser en plusieurs blocs. Ces blocs sont stockés dans un cache temporaire distribué (via le service *BlockManager*). Les exécuteurs récupèrent les blocs en parallèle, parfois auprès d'autres exécuteurs (et pas uniquement du driver), ce qui permet une diffusion progressive, parallèle et scalable. Ce mécanisme réduit significativement la charge du driver et évite une saturation du réseau lorsque le même objet doit être envoyé à de nombreux nœuds simultanément.

2.3 MPI (Message Passing Interface)

Le standard **MPI (Message Passing Interface)** est une interface de programmation conçue pour le développement d'applications parallèles sur des architectures distribuées. Il a été introduit dans les années 1990 par le *MPI Forum*[3], afin de proposer un modèle standardisé pour le calcul haute performance. Depuis sa première version publiée en 1994 (MPI-1)[4], le standard a évolué à travers plusieurs extensions (MPI-2, MPI-3, MPI-4), ajoutant progressivement des fonctionnalités telles que les entrées/sorties parallèles, les communications unilatérales, ou encore des fonctions collectives non bloquantes[5].

MPI repose sur un paradigme fondé sur le passage explicite de messages entre processus, chacun disposant de sa propre mémoire, ce qui le distingue des modèles à mémoire partagée. Dans ce contexte, chaque processus est identifié par un rang unique, et les communications peuvent être : **(1) point-à-point**, entre deux processus (`MPI_Send`, `MPI_Recv`) ; **(2) collectives**, entre un groupe de processus (ex. `MPI_Bcast`, `MPI_Reduce`)[6].

Le mécanisme `MPI_Bcast`, par exemple, permet à un processus source (appelé *root*) de diffuser un message à l'ensemble des autres processus, ce qui en fait un équivalent fonctionnel du broadcast dans les systèmes Big Data comme Spark.

MPI prend également en charge les communications asynchrones, qui permettent de chevaucher communication et traitement. Les fonctions non bloquantes, telles que `MPI_Isend` ou `MPI_Irecv`, permettent d'initier une communication tout en poursuivant le calcul. Cela améliore la scalabilité et les performances, au prix toutefois d'une programmation plus complexe, nécessitant une gestion explicite de la synchronisation.

2.4 Systèmes de gestion de données basés sur Apache Spark

Comme mentionné dans l'introduction, de nombreux systèmes de gestion de données RDF sont basés sur Spark pour définir l'architecture de leur système. Nous présentons les systèmes les plus représentatifs de cette catégorie.

SparkRDF [3] repose sur une approche de partitionnement vertical pour diviser le graphe RDF en sous-graphes, en créant une table distincte pour chaque propriété (prédicat), ainsi que pour chaque classe. En complément, SparkRDF construit plusieurs types d'index : sur les tables de classes et de propriétés, mais aussi sur les combinaisons classe-propriété, propriété-classe et classe-propriété-classe, afin d'optimiser les jointures fréquentes. Ces index sont ensuite chargés dans une structure de données en mémoire, implémentée sous forme de RDD spécialisés dans Spark. Cette structure permet l'exécution efficace des opérations telles que les jointures et les filtres.

S2RDF [2] est un système de gestion RDF distribué fondé sur l'écosystème Hadoop, utilisant HDFS comme système de fichiers et Parquet comme format de stockage en colonnes. Son schéma de stockage repose sur une version étendue du partitionnement vertical, enrichie par une technique de réduction par *semi-jointures*, appliquée entre chaque paire de tables de propriétés issues des données initiales. Ce schéma engendre un surcoût en termes d'espace, mais présente l'avantage de limiter les transferts réseau uniquement

[3] Site officiel du MPI Forum : `https://www.mpi-forum.org/`
[4] MPI : A Message-Passing Interface Standard, 1994
[5] Gropp et al., "Using Advanced MPI", 2021
[6] Voir la documentation OpenMPI : https ://www.open-mpi.org/doc/

aux données réellement pertinentes pour une jointure donnée impliquant deux prédicats. Dans ce cas, une seule table de semi-jointure, préalablement calculée entre les deux prédicats, est utilisée, ce qui réduit considérablement le volume de données échangées.

2.5 Systèmes de gestion de données basés sur MPI

De nombreux systèmes de gestion de données RDF distribués et plus précisément qui adoptent une approche asynchrone se basent sur MPI pour gérer les mécanismes de transfert de données. Dans ce qui suit, nous présentons les systèmes les plus représentatifs de cette catégorie.

RDF-3X-MPI [4] est un système RDF distribué construit à partir du moteur RDF-3X et s'appuyant sur l'interface de communication parallèle MPI (Message Passing Interface). Après une étape de codage dictionnaire des triplets RDF, les données sont initialement réparties entre les machines selon une fonction de hachage appliquée aux nœuds du graphe. Cette partition initiale est ensuite étendue afin de respecter une propriété dite de garantie à n-sauts (*n-hop guarantee*), c'est-à-dire que tous les nœuds accessibles en au plus n sauts à partir d'un nœud affecté à une partition sont également présents dans cette même partition. Chaque partition est ensuite stockée localement dans une instance de RDF-3X. L'évaluation des motifs de graphe de base (Basic Graph Patterns, BGP) est réalisée de manière autonome sur chaque partition, sans communication inter-nœuds, en supposant que la valeur de n est suffisamment grande pour couvrir les requêtes ciblées. Sinon, un échange des résultats intermédiaires est requis pour poursuivre l'évaluation, ce qui est effectué en utilisant MPI.

TriAD [8] est un système RDF distribué en mémoire, reposant sur une architecture de type *maître-esclave*. Le nœud maître maintient plusieurs structures globales essentielles : un dictionnaire des triplets RDF, un résumé du graphe permettant d'éliminer des résultats intermédiaires non pertinents, ainsi que des statistiques globales basées sur la cardinalité, utilisées pour générer un plan d'exécution optimal des requêtes SPARQL. Le résumé du graphe est construit à partir d'un graphe quotient, obtenu via le partitionnement du graphe RDF à l'aide de l'outil METIS. Chaque partition devient un *super-nœud* dans ce graphe résumé, et les arêtes étiquetées entre super-nœuds représentent des triplets reliant des entités appartenant à différentes partitions. Ce graphe résumé est indexé selon deux permutations : `PSO` et `POS`. Lorsqu'un triplet relie deux partitions différentes sur deux machines différentes, il est dupliqué sur les deux nœuds concernés. Chaque esclave indexe son sous-graphe dans les six permutations standards des triplets (`SPO`, `SOP`, `PSO`, `POS`, `OSP`, `OPS`) pour permettre une évaluation efficace. Lors de l'évaluation d'une requête SPARQL, le maître utilise le résumé de graphe pour identifier les partitions pertinentes, puis envoie cette information aux esclaves afin de restreindre l'espace de recherche, en utilisant MPI. TriAD implémente un algorithme de jointure asynchrone basé sur MPI, ce qui permet une communication efficace entre nœuds sans synchronisation bloquante.

2.6 Discussion

Les systèmes basés sur Spark et Hadoop, conçus pour le calcul massivement parallèle, imposent généralement aux systèmes de gestion de données RDF une couche d'adaptation spécifique afin de pouvoir exploiter efficacement ces frameworks. Par exemple, un système RDF souhaitant fonctionner sur Spark doit adapter sa logique de traitement pour tirer

parti du modèle RDD, et repenser son schéma de stockage en conséquence. Toutefois, un framework comme Spark, fortement dépendant de la mémoire principale, peut s'avérer inadapté à des cas d'usage simples ou contraints en ressources, en raison du coût élevé de déploiement et des exigences en mémoire pour mettre en place un cluster Spark complet. Par ailleurs, le mécanisme de synchronisation utilisé dans ce type de framework est coûteux, introduisant une latence importante, notamment pour assurer la tolérance aux pannes. Même dans Spark, les opérations de transfert comme le shuffle peuvent engendrer une latence non négligeable, même avec de petits volumes de données.

À l'inverse, MPI propose une architecture asynchrone qui permet de réduire la latence, en supprimant le besoin de synchronisation globale. Toutefois, les systèmes de gestion de données RDF basés sur MPI reposent sur l'hypothèse que les pannes sont rares ou inexistantes, ce qui représente une limite importante. En cas de défaillance d'un nœud, l'absence de mécanismes de traçabilité ou de reprise rend la gestion des transferts complexe et potentiellement coûteuse. De plus, la programmation asynchrone sous MPI requiert une expertise technique élevée, et l'adaptation d'un système RDF à ce modèle introduit une forte complexité.

Il existe également des systèmes de gestion de données RDF qui implémentent leurs propres stratégies de transfert, mais la documentation sur ces aspects reste limitée. Cela s'explique par le fait que la communauté des bases de données RDF s'est historiquement peu intéressée à l'optimisation fine des transferts réseau, un aspect pourtant critique pour les performances à grande échelle. C'est dans ce contexte que l'équipe PQDAG a développé un système de gestion de données RDF distribué appelé **PQDAG**, qui vise à proposer un compromis entre les deux approches : d'une part, une architecture synchrone inspirée de Spark pour garantir la tolérance aux pannes, et d'autre part, une rapidité d'exécution inspirée de MPI pour réduire le temps de latence lié aux transferts de données.

2.7 Conclusion

Dans ce chapitre, nous avons présenté quelques systèmes de gestion de données RDF fondés sur les frameworks Spark et MPI, après avoir introduit les principes de fonctionnement de ces deux modèles d'exécution distribuée. Nous avons mis en évidence les différences fondamentales entre l'approche synchrone de Spark et l'approche asynchrone de MPI. La discussion finale a permis de souligner les limites propres à chaque catégorie, qu'il s'agisse des exigences en mémoire et des coûts de synchronisation dans Spark, ou du manque de tolérance aux pannes et de la complexité de programmation dans MPI. Cette analyse a motivé le développement de PQDAG, qui vise à proposer un compromis entre ces deux paradigmes.

Le chapitre suivant sera consacré à la présentation détaillée de PQDAG, ainsi qu'à l'analyse de ses propres limites en matière de transfert de données.

CHAPITRE 3

ÉTUDE DE L'EXISTANT

3.1 Introduction

Dans cette section, nous présentons PQDAG, une solution qui offre un compromis intéressant entre performance et passage à l'échelle. Nous discutons donc de l'architecture de PQDAG, sans approfondir les aspects techniques. Nous présentons également le lien entre PQDAG et le framework Orchestra, initialement utilisé pour implémenter le module de transfert de données. Nous nous inspirons de ces éléments pour proposer notre solution, laquelle sera abordé dans le chapitre suivant.

3.2 PQDAG

PQDAG est un système de gestion de données RDF distribué, basé sur QDAG [11]un système centralisé de gestion de données RDF. PQDAG démontre un excellent compromis entre passage à l'échelle et performance. La figure 3.1 illustre l'architecture globale de PQDAG. Trois modules principaux y sont distingués : (i) le chargeur de données, (ii) l'optimiseur de requêtes, et (iii) le moteur d'exécution.

3.2.1 Chargeur de données

Ce module est responsable du partitionnement et de l'indexation des données RDF. Il prend en entrée un fichier RDF et génère des fragments basés sur la notion de *characteristic sets* [12]. Ces fragments sont ensuite alloués aux différentes machines du cluster à l'aide d'une stratégie d'allocation tel que Metis [10] ou MPC [14]. Sur chaque machine, les fragments sont compressés à l'aide de techniques inspirées de RDF_3X [13], reconnu pour l'efficacité de sa stratégie de compression. Une fois compressés, les fragments sont indexés localement à l'aide d'une structure de type B+Tree.

3.2.2 Optimiseur des requêtes

Le plan d'exécution de la requête est généré dans ce module. Ses principaux composants transforment la requête SPARQL en un plan physique d'exécution, représenté sous la forme d'une séquence d'étoiles de requête. Ce plan est ensuite transmis à la couche d'exécution du système, c'est-à-dire à l'ensemble des machines du cluster. Des méthodes heuristiques sont utilisées pour générer ce plan, afin d'éviter l'énumération exhaustive des plans d'exécution possibles.

3.2.3 Exécuteur

Ce module est chargé de l'exécution du plan généré par l'optimiseur de requêtes, en suivant le modèle Volcano [7]. Ce modèle repose sur le principe de la bufférisation : il consiste à évaluer les étoiles de requête issues du plan d'exécution généré par le module précédent, et à continuer la génération des résultats tant que le tampon (buffer) n'est pas plein. Une fois le tampon rempli, son contenu est consommé pour évaluer l'étoile de requête suivante. Ce mécanisme vise à assurer une utilisation efficace de la mémoire principale, en maintenant un compromis entre l'usage du disque et celui de la mémoire. L'évaluation parallèle des requêtes est abordée dans la section suivante.

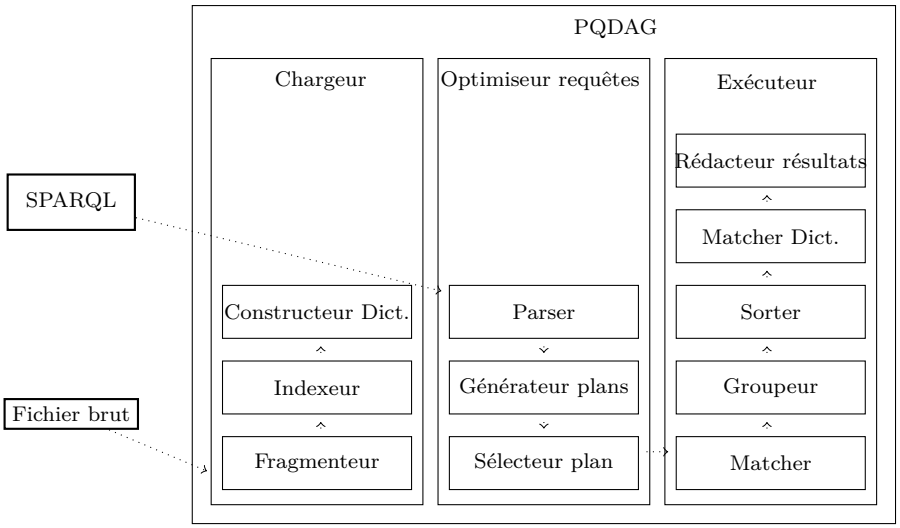

FIG. 3.1 – Architecture du système PQDAG

3.2.4 Évaluation parallèle des requêtes

PQDAG repose sur le modèle BSP (Bulk Synchronous Parallel), conçu principalement pour la définition d'algorithmes parallèles. Ce modèle a été introduit par Leslie Valiant dans les années 1980 [20]. Il repose sur une architecture *shared nothing*, où les machines ne partagent aucune ressource (mémoire, disque), et fonctionnent de manière totalement autonome.

Le fonctionnement du modèle BSP est structuré en super-étapes (voir Figure 3.2), chacune se décomposant en trois phases distinctes :

1. **Calculs locaux** : Chaque machine effectue indépendamment des opérations sur les données qu'elle détient, sans communication avec les autres nœuds.
2. **Communication asynchrone** : Les données intermédiaires produites peuvent être transmises à d'autres machines par l'échange direct de messages. Ces messages ne sont toutefois accessibles qu'à l'issue de la phase de synchronisation, assurant l'absence d'interférences durant les calculs locaux.
3. **Synchronisation globale** : Une barrière de synchronisation garantit que toutes les communications sont terminées avant d'entamer la super-étape suivante. Cette phase assure la coordination du système et garantit la cohérence des données.

Un des points forts du modèle BSP est l'introduction d'un modèle de coût prédictif, permettant d'évaluer le temps nécessaire à l'exécution d'une super-étape à l'aide de la formule suivante :

$$T = w_{\max} + g \cdot h_{\max} + l \tag{3.1}$$

où :

— $w_{\max}$: le temps de calcul local maximal parmi tous les processeurs ;
— $h_{\max}$: le volume maximal de données échangées par une machine ;
— g : Le coût de communication pour un message (g) dépend de la vitesse du réseau ;
— l : la latence de la barrière de synchronisation.

Le modèle BSP présente l'avantage d'un haut niveau de généralité, tout en offrant une bonne prédictibilité des performances. Il est notamment bien adapté aux architectures massivement parallèles et constitue une base conceptuelle pour de nombreux frameworks modernes de calcul distribué, tels que Pregel, Apache Giraph ou BSP4J.

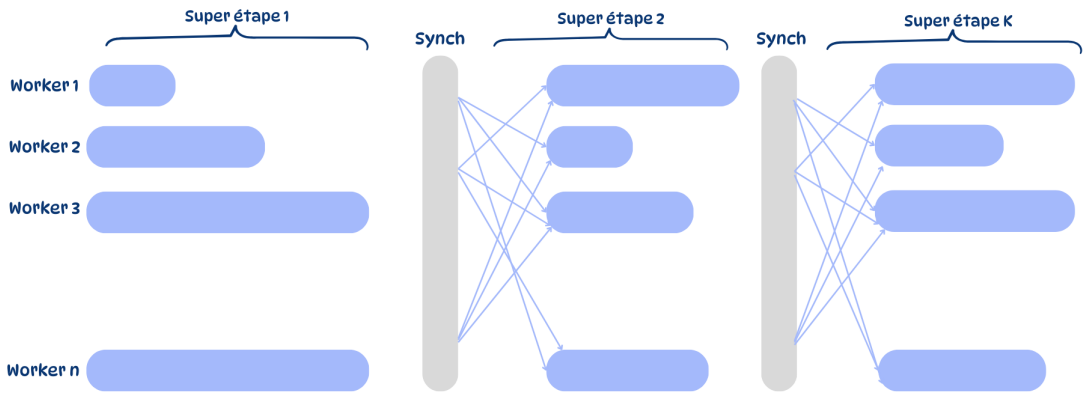

FIG. 3.2 – Apperçu du modèle BSP

PQDAG s'appuie principalement sur le modèle BSP pour l'évaluation distribuée de requêtes SPARQL. L'évaluation commence dès la réception d'une requête SPARQL envoyée par le client au niveau du master. Ce dernier génère un plan d'exécution optimal à l'aide d'heuristiques. Ce plan est représenté sous la forme d'une séquence d'étoiles de requête (sous-requêtes), couvrant l'ensemble de la requête SPARQL initiale. La figure 3.3 illustre l'exécution parallèle d'une requête SPARQL dans PQDAG.

Une fois le plan construit, le plan est diffusé à l'ensemble des workers du cluster. À partir de cette étape, le modèle BSP intervient à travers ses deux phases principales : calcul local et synchronisation. PQDAG suppose initialement que le nombre de super-étapes équivaut au nombre d'étoiles de requête.

1. **Calcul local :** la première super-étape est exécutée simultanément sur l'ensemble des machines. PQDAG tente d'évaluer autant d'étoiles de requête que possible, jusqu'à ce que toutes les machines atteignent une condition de blocage (pour plus de détails, se référer à l'article de *QDAG* [11]). Chaque super-étape i commence par lévaluation de la $i^{\text{ème}}$ étoile de requête, en supposant que les super-étapes précédentes (d'indice inférieur à i) ont déjà produit les résultats nécessaires à la poursuite du traitement.
2. **Synchronisation :** À l'issue de chaque super-étape, des objets de résultats intermédiaires sont produits. Ces objets sont ensuite sérialisés sous forme de blocs de données, qui sont échangés entre les différentes machines du cluster (opération de shuffle). La super-étape suivante ne peut être déclenchée que lorsque tous les blocs de données ont été transmis et reçus par leurs destinataires respectifs.

Lorsque toutes les étoiles de requête ont été évaluées, la requête SPARQL est considérée comme résolue. Dans le meilleur des cas, un nombre réduit de super-étapes peut suffire. Toutefois, dans le pire des cas, autant de super-étapes que d'étoiles de requêtes peuvent être nécessaires. Enfin, les résultats finaux sont renvoyés au client.

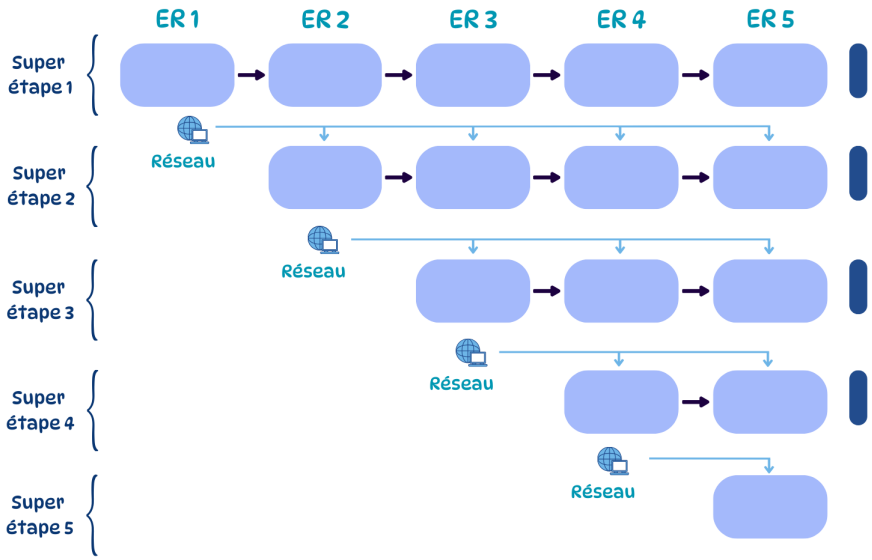

FIG. 3.3 – L'évaluation parallèle des requêtes dans PQDAG

3.2.5 PQDAG et Orchestra

Dans la section précédente, nous avons détaillé l'évaluation parallèle des requêtes dans **PQDAG**. Celle-ci repose explicitement sur des mécanismes de transfert, de sérialisation et de désérialisation, utilisés pour gérer efficacement les échanges de données entre les machines.

Cette partie du système s'inspire principalement du framework Orchestra, proposé par Mosharaf Chowdhury [5]. Nous discutons ici des limites de la version actuelle de PQDAG en matière de transfert de données, ainsi que sur les motivations qui ont conduit à la conception de notre solution.

1. **Broadcast :** PQDAG intègre une stratégie de broadcast basée sur un arbre équilibré, dans lequel chaque nœud interne possède un nombre fixe de deux enfants. Cette implémentation, bien que simple, manque de flexibilité et ne constitue pas toujours l'approche optimale. Par exemple, dans le cas de la diffusion d'un bloc de données de petite taille, une stratégie de broadcast linéaire (de type *chain broadcast*) peut s'avérer plus efficace, en réduisant significativement la latence introduite par une structure arborescente.

   Une seconde limite de cette stratégie réside dans le fait que les nœuds-feuilles ne participent pas activement à la diffusion, ce qui limite l'utilisation globale de la bande passante et peut ralentir la propagation de l'information à grande échelle.

2. **Shuffle :** PQDAG intègre une stratégie de transfert de blocs de données entre les différentes machines, appelée *shuffle équitable*, inspirée du framework **Orchestra**. Cette approche attribue la même priorité de transfert, et donc une bande passante équivalente, à l'ensemble des *workers* participant à l'opération de shuffle. Autrement dit, un worker devant transférer 10 blocs de données se voit allouer la même bande passante qu'un autre n'en envoyant qu'un seul. Ce mécanisme, bien que simple, peut s'avérer sous-optimal dans un contexte de synchronisation

BSP, où toutes les machines doivent terminer leurs transferts avant de passer à la super-étape suivante.

Une autre limitation réside dans le fait que le nombre de connexions autorisées, aussi bien au niveau des récepteurs que des émetteurs (tous deux étant des *workers*), est fixé de manière statique et manque de flexibilité. Ce paramètre peut significativement influencer le temps de transfert en affectant la bande passante réellement exploitée par le réseau.

3. **Sérialisation / Désérialisation :** Dans la conception de PQDAG, le mécanisme de sérialisation et de désérialisation a été conçu de manière fortement dépendante de la structure des objets représentant les résultats intermédiaires de PQDAG. Cette dépendance pose plusieurs problèmes : toute modification dans la structure des objets internes de PQDAG nécessite un important travail d'adaptation de cette couche, rendant le système peu flexible et difficile à maintenir. Cette forte liaison entre la couche de sérialisation / désérialisation et la couche d'exécution compromet la modularité de l'architecture, ce qui constitue une faiblesse majeure dans un contexte distribué évolutif.

3.3 Conclusion

Nous avons présenté PQDAG sans entrer dans les détails de mise en œuvre, en mettant l'accent sur son mécanisme d'évaluation parallèle des requêtes, qui constitue l'un de ses aspects centraux. Cette évaluation repose sur le modèle BSP et implique l'utilisation de mécanismes de transfert, de sérialisation et de désérialisation. Nous avons également discuté des limitations actuelles de ces composants, qui représentent aujourd'hui un véritable goulot d'étranglement pour l'évolutivité de PQDAG.

Dans le chapitre suivant, nous analysons les besoins spécifiques de l'équipe PQDAG en matière de transfert de données, de sérialisation et de désérialisation, et présentons la conception de notre solution, ainsi que ses différentes briques fonctionnelles.

CHAPITRE 4

ANALYSE ET CONCEPTION

4.1 Introduction

Dans ce chapitre, nous présentons les besoins identifiés par l'équipe PQDAG en matière de mécanismes de transfert réseau, dans le but de pallier les limitations observées dans le système de transfert de données décrit au chapitre précédent. Pour répondre à ces enjeux, nous introduisons notre solution, nommée *EGFDT* (Enhanced Generic Framework for Data Transfer).

Nous décrivons les différents mécanismes de transfert intégrés dans EGFDT, en mettant l'accent sur leur fonctionnement et leurs objectifs. Enfin, nous proposons une validation expérimentale de notre approche, accompagnée d'une analyse comparative entre les différents algorithmes de transfert dans EGFDT, afin de démontrer l'intérêt et l'efficacité de notre solution.

4.2 Analyse des besoins

Le framework de transfert précédemment déployé dans **PQDAG** n'est pas modulaire et présente plusieurs lacunes en termes de fonctionnalités et de stratégies d'optimisation, comme mentionné dans l'étude de l'existant. Notre objectif principal consiste à proposer une alternative améliorée au framework initial.

Les deux composantes principales que nous allons traiter sont le **Broadcast** et le **Shuffle**, avec pour objectif final de réduire le temps de transfert tout en maximisant l'utilisation de la bande passante. Dans la suite, nous détaillons les besoins spécifiques associés à chacune de ces composantes.

1. **Un Tree Broadcast avec une structure de données flexible :** Il est nécessaire de fournir une flexibilité dans la construction de l'arbre utilisé pour le transfert des données, notamment en permettant d'ajuster dynamiquement le nombre de fils par nœud. Cette capacité est cruciale pour permettre à PQDAG d'adapter sa stratégie de diffusion à la nature du transfert en cours, en tenant compte du contexte opérationnel (nombre de workers, taille des fichiers à diffuser, etc.). Par exemple :
   — Dans **un cas simple**, avec un petit nombre de workers (entre 3 et 5) et des fichiers de taille modérée (ne dépassant pas un certain seuil), la meilleure stratégie consiste à construire un arbre à un seul niveau : le master agit comme racine et envoie directement les données à tous les workers.
   — Dans un **cas plus complexe**, avec plus de 20 workers et des fichiers volumineux, une stratégie plus adaptée consiste à fixer un nombre de fils par nœud supérieur à un, afin de répartir la charge du transfert sur plusieurs niveaux de l'arbre.

   Cette flexibilité doit être accessible en temps réel, afin que le système PQDAG puisse adapter la structure de l'arbre dynamiquement selon les conditions de transfert. Néanmoins, cette approche présente une limite importante : les nœuds feuilles de l'arbre ne participent pas à la redistribution des blocs de données, ce qui signifie qu'une partie des workers peut rester inactive du point de vue du transfert, réduisant ainsi l'efficacité globale dans certains cas.
2. **Une nouvelle stratégie de broadcast plus performante :** L'équipe PQDAG souhaite intégrer une stratégie avancée de diffusion, inspirée de l'approche *Cornet* proposée dans l'architecture Orchestra [5]. Cette stratégie repose sur un protocole

de type BitTorrent, conçu spécifiquement pour les environnements de datacenters. Contrairement à la stratégie de diffusion basée sur une structure en arbre, qui n'exploite pas les ressources réseau des nœuds-feuilles, Cornet permet à chaque nœud participant à la diffusion de continuer à relayer les blocs de données après les avoir reçus. Cette approche maximise l'utilisation de la bande passante globale du cluster et permet une diffusion plus rapide et plus robuste, en particulier dans des environnements à grande échelle.

3. **D'autres stratégies de Shuffle plus performantes :** L'équipe PQDAG a exprimé le besoin d'améliorer le mécanisme de transfert des blocs de données **Shuffle**, en proposant des stratégies plus performantes que la version actuellement utilisée, basée sur un transfert équitable. L'objectif est de concevoir et d'évaluer de nouvelles stratégies en les comparant, en particulier en termes de temps de transfert, à la stratégie équitable existante. Les nouvelles stratégies envisagées doivent intégrer une meilleure flexibilité en ce qui concerne les paramètres impactant le transfert, notamment : le nombre de processus d'envoi et de réception pouvant être exécutés en parallèle dans chaque worker (paramètre absent dans la version précédente). Ces stratégies doivent viser une utilisation optimale de la bande passante disponible, en garantissant qu'à tout instant t, l'ensemble des workers participent activement au transfert, que ce soit en tant qu'émetteurs ou récepteurs de blocs.
4. **Un mécanisme de sérialisation et de désérialisation des données plus performant et générique** : Les deux mécanismes de sérialisation et de désérialisation dépendaient fortement de l'architecture de PQDAG. En revanche, notre objectif est de proposer une version générique, indépendante de l'architecture de PQDAG, permettant d'assurer une couche de sérialisation et de désérialisation efficace.
5. **Généricité des composantes de transfert proposées :** Les fonctionnalités de transfert développées dans le cadre de notre framework doivent être conçues de manière modulaire et indépendante du système PQDAG. L'objectif est de permettre leur réutilisation dans d'autres systèmes de gestion de données nécessitant des mécanismes de diffusion (broadcast) ou d'échange de blocs de données (shuffle). Cette généricité est particulièrement pertinente pour les systèmes distribués fondés sur le modèle BSP, où les phases de communication de données jouent un rôle central.

4.3 EGFDT (Enhanced Generic Framework For Data Transfer)

Nous proposons EGFDT (Enhanced Generic Framework For Data Transfer), un framework générique pour le transfert de données dans un système de gestion de données distribué. EGFDT couvre l'ensemble des besoins exprimés par l'équipe PQDAG et peut être intégré dans PQDAG. Nous avons évalué les performances de notre solution et comparé les différentes stratégies de transfert proposées. Figure 4.1 illustre l'architecture globale d'EGFDT ainsi que les principaux composants de transfert implémentés dans ce framework.

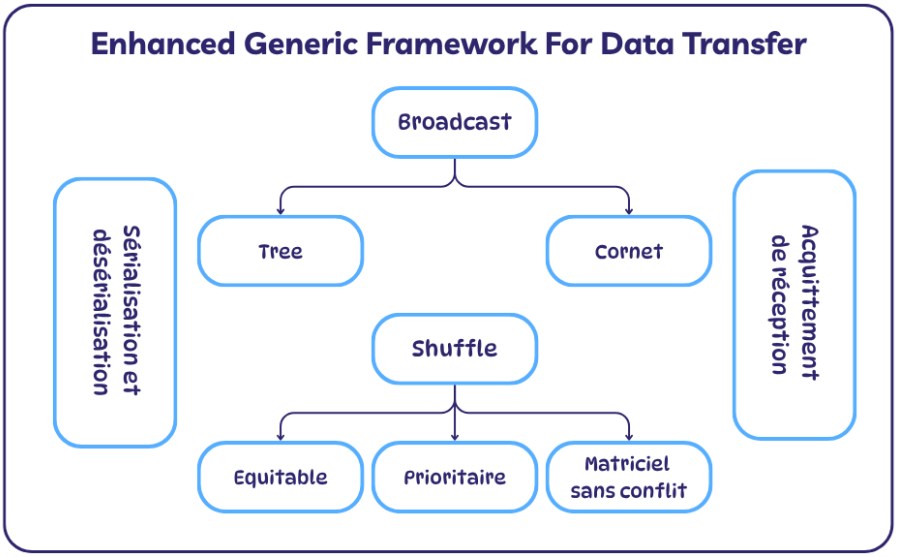

FIG. 4.1 – Architecture modulaire du framework EGFDT pour le transfert de données

4.3.1 Broadcast

Comme solution de diffusion, nous proposons au sein d'EGFDT deux approches : **Tree Broadcast** et **Cornet**. Dans ce qui suit, nous discutons de ces deux solutions, en présentant les avantages et les inconvénients de chacune, ainsi que certains paramètres importants ayant un impact sur leurs performances. Une comparaison en termes de temps de transfert est également fournie.

Tree Broadcast

Cette stratégie repose sur une diffusion en arbre équilibré, avec un nombre de fils d configurable dynamiquement, contrairement à l'approche initialement utilisée dans PQ-DAG, où cette valeur était fixe et où l'arbre était construit dès l'initialisation du système. La Figure 4.2 illustre cette stratégie.

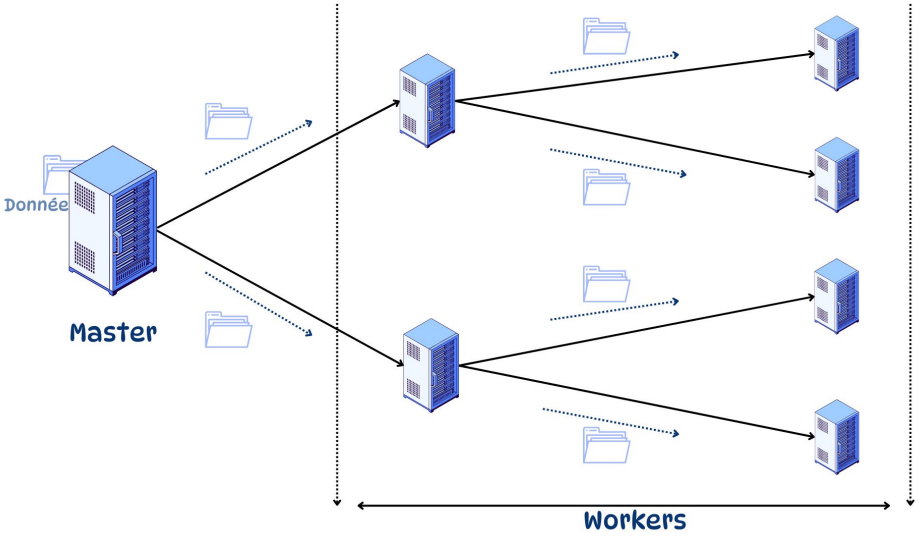

FIG. 4.2 – Exemple de Tree Broadcast avec d = 2

L'arbre est construit à la demande, au moment du transfert, en prenant en entrée le paramètre d ainsi que la liste des workers participant à la diffusion. Le master initie le processus en divisant le fichier à transférer en blocs de données et en les transmettant aux nœuds du premier niveau de l'arbre.

Chaque nœud relaie ensuite les blocs reçus à ses propres fils, ce qui permet de réduire considérablement la charge sur le master. La diffusion se poursuit récursivement jusqu'à ce que l'ensemble des blocs atteigne toutes les feuilles de l'arbre, garantissant ainsi une distribution complète des données.

Scénario d'évaluation : La métrique utilisée dans ces tests est le temps de transfert global. Nous avons évalué la diffusion de fichiers de différentes tailles (10 Mo, 100 Mo et 1 Go) sur un cluster composé de 11 machines (un master et 10 workers), en faisant varier le nombre d'enfants par nœud dans l'arbre de diffusion ($d = 2, 4, 6, 8, 10$).

Comme le montre la Figure 4.3, un $d = 2$ est dans la plupart des cas est le meilleur choix de nombre de fils par nœud. D'où dans notre configuration nous fixons d à 2 et nous laissons aussi le choix à un administrateur de modifier la valeur de d à tout moment pour que la création de l'arbre soit flexible. Il est fort probable que si le nombre de machines utilisées dans le transfert est supérieur à 10, le paramètre d peut changer et que $d = 2$ peut-être n'est pas le meilleur choix.

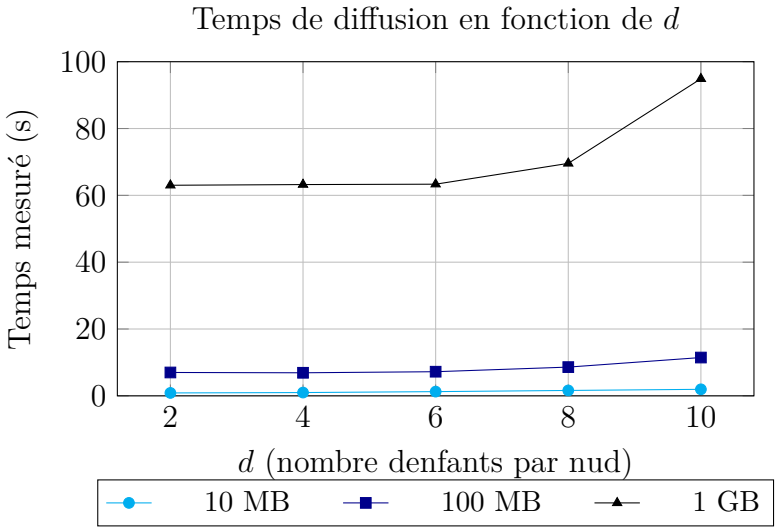

FIG. 4.3 – Temps de diffusion en fonction de d

Cornet

Cornet est un protocole de diffusion issu de l'architecture *Orchestra*, conçu spécifiquement pour les environnements de calcul haut performance. Inspiré du fonctionnement de BitTorrent, il s'en distingue par plusieurs aspects clés. Contrairement à BitTorrent, où certains nœuds peuvent se retirer du processus dès qu'ils ont reçu l'intégralité des données, Cornet impose une participation active de tous les nœuds jusqu'à la fin du transfert. De plus, les données sont fragmentées en blocs de grande taille (4 Mo), ce qui permet de réduire la surcharge induite par la gestion de nombreux petits blocs. Grâce à ces optimisations, Cornet offre des performances significativement supérieures, atteignant jusquà 4,5 fois la vitesse de diffusion observée avec des solutions classiques telles que Hadoop ou BitTorrent [5].

Fonctionnement Dans la suite, nous détaillons le fonctionnement global de Cornet en distinguant trois acteurs clés : le *Master*, les *Workers* et le *Tracker*.

Le master : fragmente l'objet à diffuser en blocs de taille fixe (4 Mo par défaut), puis les distribue aux nœuds participants selon une stratégie *round-robin* pour amorcer rapidement la phase de partage. Il héberge en outre un *tracker* central qui maintient, en temps réel, la table de localisation des blocs.

Le worker : Chaque worker reçoit un sous-ensemble initial de blocs, puis entre en phase d'échanges pair-à-pair pour obtenir les blocs manquants. À chaque réception, il met à jour sa table locale et notifie le tracker. Une fois l'ensemble des blocs acquis, il reconstitue l'objet complet en concaténant les blocs dans l'ordre.

Le tracker : Hébergé par le *master*, le tracker orchestre tous les échanges. Lorsqu'un *worker* demande un bloc, il lui renvoie la liste des pairs qui le possèdent, classée par disponibilité : un nœud ayant déjà reçu l'ensemble de ses blocs est prioritaire sur un nœud encore en cours de réception. Cette politique maximise l'utilisation de la bande passante et réduit la latence globale.

La figure 4.4 illustre la diffusion d'un fichier découpé en quatre blocs, initialement répartis sur les workers selon la stratégie round-robin. Au moment représenté, le *worker 4* a déjà reçu l'intégralité des blocs, tandis que les autres nœuds sont encore en train de récupérer les blocs manquants. Par exemple, le *worker 1* réclame le bloc 2, son dernier bloc

manquant. Comme le bloc 2 est détenu simultanément par les *workers 2* et 4, le tracker privilégie le *worker 4* : ce dernier, ayant terminé sa phase de réception, peut désormais consacrer toute sa bande passante à l'envoi, accélérant ainsi la complétion du transfert pour le *worker 1*.

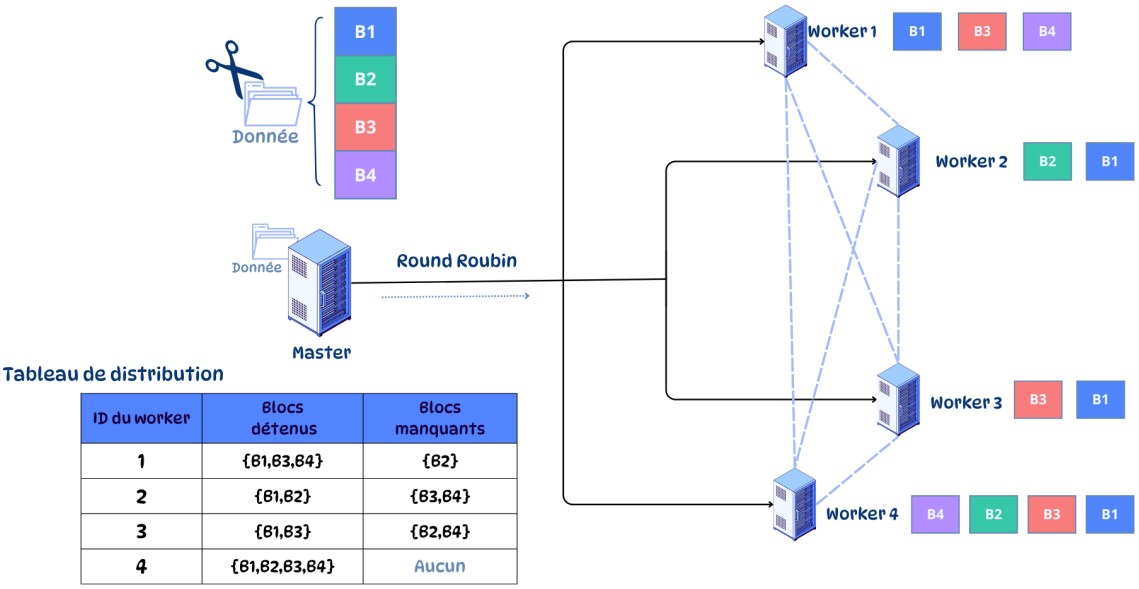

Tableau de distribution

ID du worker	Blocs détenus	Blocs manquants
1	{B1,B3,B4}	{B2}
2	{B1,B2}	{B3,B4}
3	{B1,B3}	{B2,B4}
4	{B1,B2,B3,B4}	Aucun

FIG. 4.4 – Architecture global du protocole Cornet

4.3.2 Tree-Broadcast Vs Cornet

Nous avons comparé le temps de transfert global de deux stratégies de diffusion : (1) la diffusion reposant sur une structure en arbre (avec un facteur $d = 2$) et (2) la stratégie basée sur le protocole Cornet. Pour cette comparaison, nous avons fait varier la taille des fichiers à diffuser. Les résultats, présentés dans la Figure 4.5, permettent de tirer les conclusions suivantes :

Cornet surpasse Tree-Broadcast dans la majorité des cas, notamment grâce à la participation active de la majorité des nœuds tout au long du transfert. Contrairement à la diffusion arborescente, où les nœuds feuilles ne relaient pas les données, Cornet exploite pleinement les capacités de chaque participant, ce qui améliore la bande passante globale. Cependant, dans certains scénarios impliquant des fichiers de petite taille (inférieurs à 50 Mo), la stratégie arborescente s'avère plus efficace. En effet, le coût de latence associé aux consultations fréquentes du tracker dans Cornet devient prédominant et ralentit le transfert global. Ainsi, dans le cadre d'une intégration au sein du système de gestion de données PQDAG, une simple condition basée sur la taille du fichier peut permettre de sélectionner dynamiquement la stratégie de diffusion la plus adaptée.

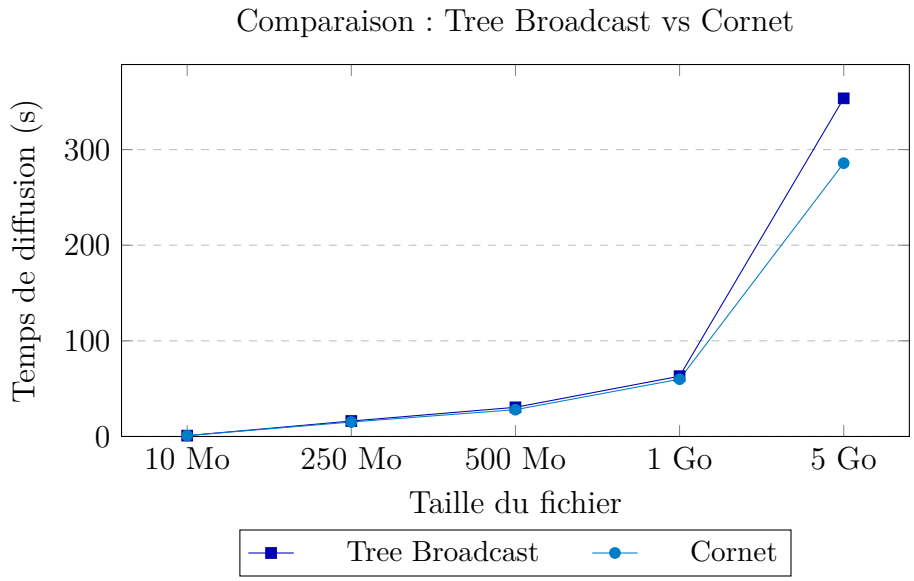

FIG. 4.5 – Comparaison du temps de diffusion entre Tree Broadcast et Cornet

4.3.3 Diagramme de classe pour le Broadcast

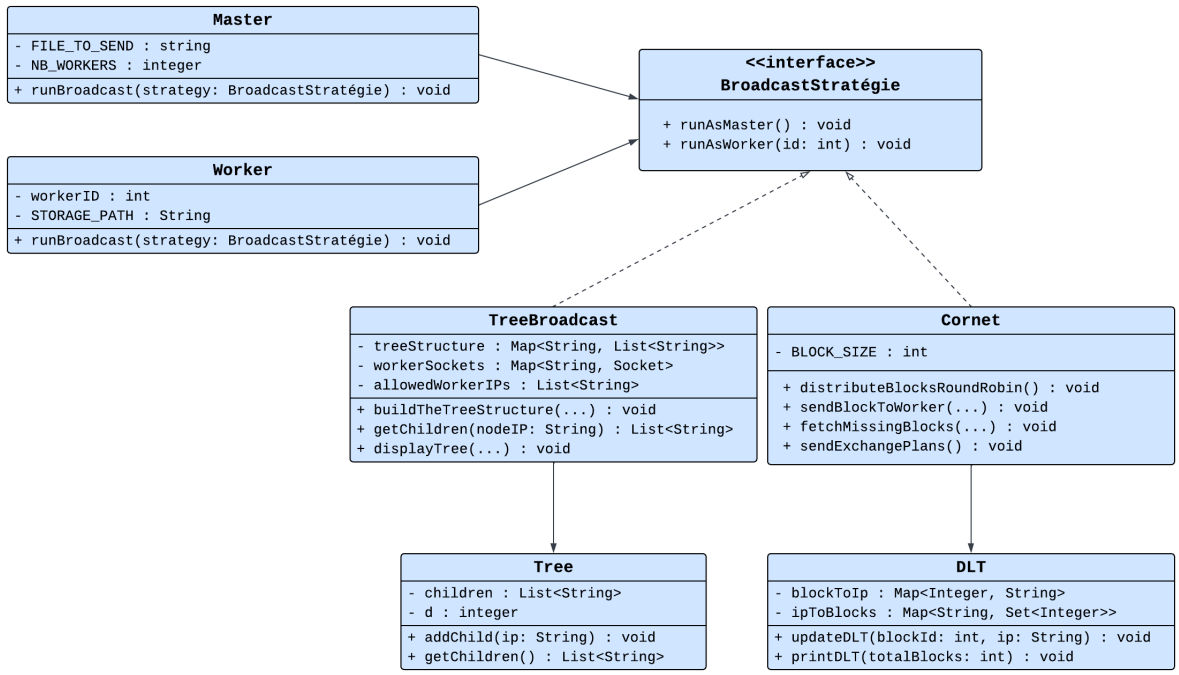

FIG. 4.6 – Diagramme de classes pour les stratégies de diffusion (broadcast)

Le diagramme présenté dans la Figure 4.6 illustre le diagramme de classes adopté pour implémenter les différentes stratégies de diffusion dans notre solution. Les deux classes principales, `Master` et `Worker`, utilisent une instance de `BroadcastStratégie` pour exécuter dynamiquement une stratégie de diffusion sélectionnée. L'interface `BroadcastStratégie` définit deux méthodes principales, `runAsMaster()` et `runAsWorker(int)`, qui sont implémentées par les deux classes concrètes `TreeBroadcast` et `Cornet`. Ces dernières définissent la manière dont les workers exécutent le protocole de diffusion.

`TreeBroadcast` s'appuie sur une structure arborescente modélisée par la classe `Tree`, et construit dynamiquement l'arbre de diffusion à partir des adresses IP des workers. En revanche, `Cornet` repose sur une table de localité des données DLT, encapsulée dans la classe `DLT`.

4.4 Shuffle

Dans cette section, nous présentons les trois stratégies de shuffle que nous avons implémentées : le shuffle équitable, le shuffle prioritaire, et un shuffle basé sur une heuristique dite « *matrice sans conflit* ». La première stratégie était déjà implémentée dans PQDAG ; nous l'avons réimplémentée afin de la comparer au shuffle prioritaire, inspiré dOrchestra, ainsi quà notre nouvelle stratégie, fondée sur une heuristique de type « *matrice sans conflit* ».

4.4.1 Shuffle équitable

La stratégie d'ordonnancement équitable vise à répartir équitablement la bande passante d'un worker entre ses différents flux sortants, sans tenir compte du nombre de blocs à transmettre dans chaque flux. Un flux correspond ici à une connexion réseau entre un émetteur (sender) et un récepteur (receiver). Tous les flux sont donc considérés avec la même priorité.

Le master collecte les besoins de chaque receiver et construit un plan global de réception, organisé en une série d'itérations, qu'il transmet ensuite à chaque receiver. Deux paramètres jouent un rôle essentiel dans ce mécanisme : (1) le nombre de réceptions simultanées autorisées côté receiver ; (2) le nombre d'envois simultanés autorisés côté sender.

Dans une stratégie équitable, ces deux paramètres doivent être symétriques et équilibrés. Une fois son plan local reçu, chaque receiver envoie, à chaque itération, une demande de bloc au sender concerné. Du côté des senders, chaque machine maintient une file d'attente FIFO regroupant les demandes reçues de l'ensemble des receivers. Si le nombre d'envois autorisé est fixé à n, alors le sender peut alors traiter au maximum n demandes à chaque itération.

La figure 4.7 illustre un exemple de cette stratégie. Trois *senders* (S1, S2, S3) transmettent des blocs à trois receivers (R1, R2, R3). Le master élabore un plan global d'ordonnancement représenté sous la forme d'un tableau, où chaque cellule indique l'envoi d'un bloc d'un sender vers un receiver à un instant donné. Ce plan est exécuté en $T = 7$ itérations, représentant le temps total de transfert. Dans cet exemple, ainsi que dans les autres exemples à venir les paramètres sont fixés à une réception et à un envoi par itération, afin d'évaluer de manière claire l'efficacité des différentes stratégies. Lorsqu'un sender reçoit simultanément plusieurs demandes par exemple, à l'instant t_0, S1 reçoit des requêtes de R2 et R3. Dans ce cas, le sender ne peut satisfaire qu'une seule requête, et l'autre doit patienter jusqu'à l'itération suivante.

Cette stratégie présente un inconvénient notable : les flux comportant un petit nombre de blocs se terminent rapidement, tandis que les flux plus chargés s'étendent sur plusieurs itérations. Ainsi, même si la bande passante est répartie de manière équilibrée, l'ordonnancement équitable n'assure pas une terminaison simultanée des workers. Or, dans le contexte de PQDAG, la synchronisation des fins de transfert entre machines est essentielle, ce qui justifie l'exploration de stratégies alternatives mieux adaptées.

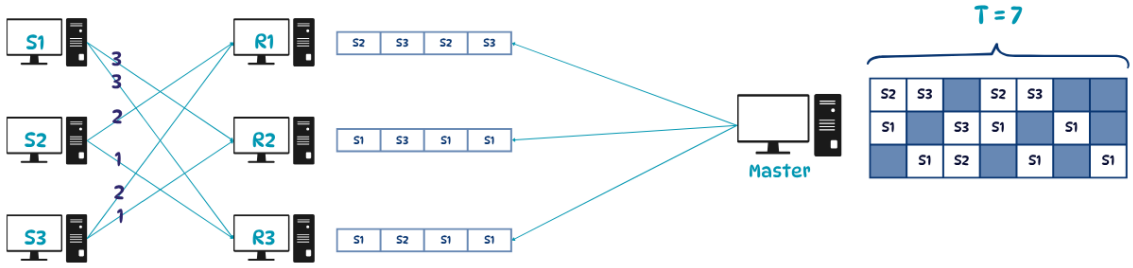

FIG. 4.7 – Shuffle équitable

4.4.2 Shuffle prioritaire

Contrairement à la stratégie équitable, le shuffle prioritaire repose sur une répartition pondérée de la bande passante, en tenant compte de la charge de chaque flux. Un flux ayant davantage de blocs à transmettre ou à recevoir bénéficie d'une priorité plus élevée, ce qui lui permet d'obtenir une part plus importante de la bande passante.

Cette approche vise à améliorer la synchronisation globale en s'assurant que tous les flux achèvent leurs transferts dans un intervalle de temps rapproché. Cela permet d'éviter qu'un worker reste actif beaucoup plus longtemps que les autres, ce qui est particulièrement important dans des systèmes comme PQDAG, où une fin de transfert simultanée entre les machines est recherchée pour optimiser l'efficacité globale.

La figure 4.8 illustre cette stratégie en reprenant le même scénario que pour le shuffle équitable : trois senders (S1, S2, S3) doivent transmettre des blocs à trois receivers (R1, R2, R3). Cependant, contrairement à la stratégie équitable, ici le plan d'ordonnancement est différent. Le master tient compte du nombre de blocs par flux pour générer un plan priorisant les flux les plus chargés.

Le plan est exécuté en 7 itérations (T = 7), comme précédemment, mais la répartition des envois est ajustée pour que S1, dont les flux sont les plus chargés, soit sollicité plus fréquemment dès le début. Cela permet une terminaison plus homogène des envois entre les différents workers.

Cette stratégie constitue un bon compromis entre performance de transfert et terminaison quasi simultanée des workers. En priorisant les flux les plus chargés, elle permet de mieux équilibrer la durée des échanges entre les différentes machines. Toutefois, elle présente une limite importante : lorsqu'un même sender est sollicité simultanément par plusieurs receivers, un seul peut être servi à l'instant considéré, contraignant les autres à attendre l'itération suivante. Cela peut introduire des périodes d'inactivité pour certains receivers, réduisant ainsi le parallélisme potentiel.
De plus, dans l'exemple illustré à la figure 4.8, bien que l'ordonnancement soit plus intelligent que dans la stratégie équitable, le nombre total d'itérations reste inchangé ($T = 7$). Cela montre que le gain en synchronisation ne se traduit pas nécessairement par une amélioration directe du temps global de transfert.
Une stratégie plus efficace consisterait à exploiter de manière optimale les ressources disponibles aussi bien côté senders que receivers à chaque itération, afin de maximiser les transferts parallèles tout en réduisant les temps d'attente inutiles.

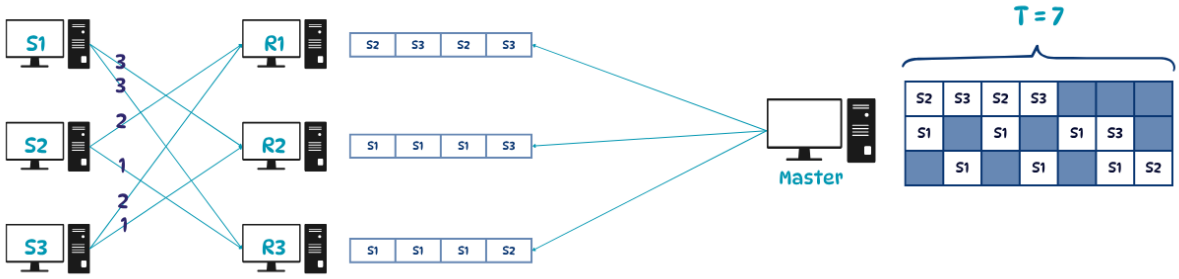

FIG. 4.8 – Shuffle prioritaire

4.4.3 Shuffle matriciel sans conflit

Le Shuffle matriciel sans conflit est une stratégie de transfert que nous avons développée afin de résoudre les limitations observées dans les approches équitable et prioritaire. Elle vise à éviter les conflits d'accès simultané à un même sender tout en maximisant l'utilisation des machines (senders et receivers) à chaque itération.

Le principe clé de cette stratégie repose sur la construction d'une matrice sans conflit, qui représente un plan d'ordonnancement plus efficace que ceux générés par le shuffle prioritaire et équitable. Cette matrice essaie de garantir au maximum qu'à chaque itération, aucun sender n'est sollicité par plus d'un receiver, ce qui élimine les blocages potentiels liés à des demandes concurrentes.

Nous pensons que le problème général d'ordonnancement dans ce contexte est *NP-difficile*, car il s'apparente aux problèmes classiques d'allocation de tâches avec contraintes de ressources. Bien que nous n'ayons pas démontré formellement cette complexité, nous avons conçu une heuristique spécifique, détaillée dans l'algorithme 42.

Algorithm 1 : displayNonConflictPlan

Input : senderPlans : Map(sender → Map(receiver → nb_blocs))
receiverIPs : List, senderIPs : List
Output : matrix : Liste d'étapes sans conflit (receiver → sender)

```
// Étape 1 : Initialisation des files ;
foreach receiver ∈ receiverIPs do
    queue[receiver] ← [] ;
    foreach sender ∈ senderIPs do
        n ← senderPlans[sender][receiver] ;
        for i ← 1 to n do
            queue[receiver].add(sender)
        end
    end
end
// Étape 2 : Construction de la matrice sans conflit ;
matrix ← [] ;
while au moins une queue non vide do
    usedSenders ← {} ;
    step ← {} ;
    foreach receiver ∈ receiverIPs do
        foreach sender ∈ queue[receiver] do
            if sender ∉ usedSenders then
                step[receiver] ← sender ;
                queue[receiver].remove(sender) ;
                usedSenders.add(sender) ;
                break
            end
        end
        if receiver ∉ step then
            step[receiver] ← "-"
        end
    end
    if step.values() == "-" pour tous then
        break
    end
    matrix.add(step)
end
// Étape 3 : Envoi des plans à chaque receiver ;
foreach receiver ∈ receiverIPs do
    plan ← [] ;
    foreach step ∈ matrix do
        sender ← step[receiver] ;
        plan.add(sender)
    end
    envoyer plan via socket à receiver
end
```

L'algorithme 42 décrit le processus de construction de la matrice d'ordonnancement

sans conflit. Il commence par transformer le plan initial, qui associe à chaque sender un nombre de blocs à envoyer vers chaque receiver, en une file de demandes pour chaque receiver. Ces files contiennent une liste ordonnée des senders, répétée selon le nombre de blocs à recevoir. À chaque itération, l'algorithme construit une nouvelle ligne de la matrice en attribuant à chaque receiver le premier sender disponible dans sa file, à condition que ce sender n'ait pas déjà été assigné à un autre receiver au cours de la même itération. Si aucun sender n'est disponible, le receiver reste en attente (indiqué par « - »). Ce processus est répété jusqu'à ce que toutes les files soient vides, produisant ainsi une matrice garantissant une exécution sans conflit, où chaque sender est sollicité au plus une fois par étape.

Dans les trois stratégies présentées, nous avons formulé l'hypothèse suivante afin de mieux étudier le problème : chaque sender et chaque receiver ne peut envoyer ou recevoir quun seul bloc de données à la fois. Cette contrainte simplificatrice nous a permis d'identifier un phénomène clé : dans les stratégies équitable et prioritaire, le temps de transfert global reste inchangé, principalement en raison des demandes simultanées adressées à un même sender. Or, ce dernier, étant limité à un envoi à la fois, devient un point de congestion, entraînant une latence supplémentaire.
Cette observation nous a conduits à concevoir une stratégie alternative, fondée sur une heuristique, qui vise à minimiser les conflits d'accès concurrents aux senders. Cette stratégie repose sur la construction d'une matrice sans conflit, dans laquelle on évite autant que possible d'assigner un même sender à plusieurs receivers lors d'une même itération. Bien que cette approche ne garantisse pas une solution optimale, elle s'est révélée efficace dans plusieurs cas, en réduisant notablement les conflits et donc les délais de transfert.

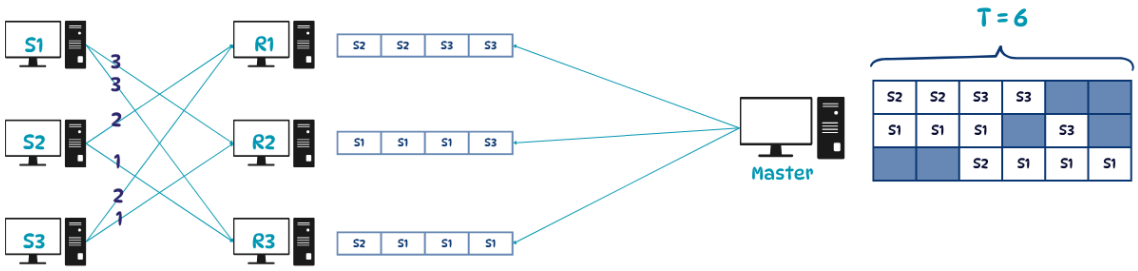

FIG. 4.9 – Shuffle matriciel sans conflit

La figure 4.9 illustre le plan d'ordonnancement obtenu pour le même exemple utilisé précédemment. On y observe que, grâce à l'élimination des conflits, le temps total de transfert est réduit à $T = 6$, contre $T = 7$ pour les stratégies *équitable* et *prioritaire*.

4.4.4 Validation Expérimentale

Nous avons testé le même exemple afin de valider les trois solutions proposées. Les deux premières stratégies, à savoir le *shuffle équitable* et le *shuffle prioritaire*, ont abouti à un temps de transfert global d'environ 16 secondes. En revanche, la troisième stratégie,

basée sur une matrice sans conflit, a permis de réduire ce temps à environ 14 secondes, ce qui constitue une amélioration notable.

4.4.5 Diagramme de classe pour le shuffle

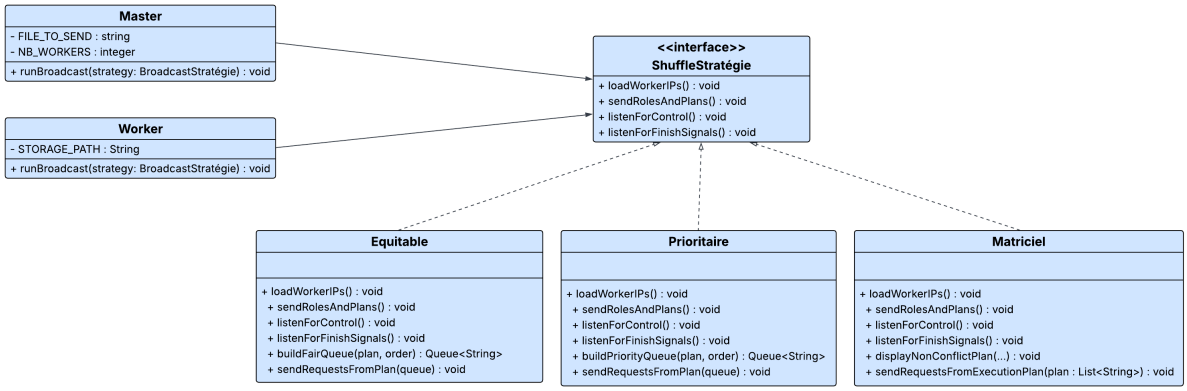

FIG. 4.10 – Diagramme de classes pour les stratégies de transfert (shuffle)

La Figure 4.10 présente le diagramme de classes des différentes stratégies de shuffle. Cette modélisation repose sur l'interface `ShuffleStratégie`, qui définit les quatre méthodes fondamentales utilisées dans toutes les variantes de shuffle : chargement des adresses IP, envoi des rôles et des plans, écoute des instructions, et synchronisation des acquittements de réception. Les classes `Equitable`, `Prioritaire` et `Matriciel` implémentent cette interface en suivant chacune une logique spécifique de construction des plans de réception.

4.4.6 Le reste d'EGFDT

En complément des stratégies de diffusion implémentées, à savoir *Tree Broadcast* et *Cornet*, ainsi que des variantes de *shuffle* (équitable, prioritaire et basé sur une matrice sans conflit), nous avons également intégré un protocole d'accusé de réception visant à garantir la bonne réception des messages critiques par les workers impliqués. Par exemple, dans le cas du protocole de diffusion, chaque worker doit envoyer un accusé de réception (ACK) au master dès qu'il reçoit l'objet diffusé. Le master ne valide l'opération complète qu'après avoir reçu l'ensemble des accusés de réception de tous les workers concernés.

Par ailleurs, nous avons repensé le protocole de sérialisation/désérialisation. Contrairement à l'approche initialement utilisée dans PQDAG, où la sérialisation était étroitement couplée à la structure des objets Java propres à PQDAG, notre nouvelle implémentation repose sur un framework générique. Celui-ci permet une sérialisation totalement indépendante des classes d'objets utilisées. Cette amélioration permet désormais de sérialiser n'importe quelle structure d'objet, et de générer des blocs de données prêts à être diffusés via *broadcast* ou échangés via *shuffle*, sans dépendance à la structure interne de PQDAG.

4.5 Conclusion

Nous avons analysé les besoins de l'équipe PQDAG afin d'améliorer la composante de transfert dans PQDAG. Nous avons conçu et implémenté les différentes composantes du

framework EGFDT, à savoir le *broadcast* avec ses deux stratégies Tree et Cornet, ainsi que le *shuffle* avec ses trois stratégies : équitable, prioritaire et basée sur une matrice sans conflit. Pour chaque mécanisme de transfert, nous l'avons validé expérimentalement afin de démontrer l'efficacité de nos propositions. Dans le chapitre suivant, nous discutons des technologies, du langage et des plateformes utilisés pour mettre en place notre solution.

CHAPITRE 5

RÉALISATION

5.1 Introduction

Dans ce chapitre, nous présentons la plateforme de déploiement utilisée pour tester notre solution, ainsi que les langages et frameworks employés pour son implémentation. Enfin, nous illustrons le fonctionnement de notre framework à l'aide de captures d'écran issues de son exécution. L'ensemble du code source est disponible sur le dépôt GitHub suivant : < https://github.com/NetworkEngine >

5.2 Environnement de déploiement

5.2.1 OpenStack

Afin d'expérimenter notre framework et de valider les stratégies proposées, nous avons déployé notre propre infrastructure sur une plateforme OpenStack, administrée par l'équipe PQDAG et mise à notre disposition dans le cadre de notre projet de fin d'études. Nous avons créé et configuré par nous-mêmes un ensemble de machines virtuelles (VM), composé d'un nœud `master` et de dix nœuds de calcul `workers`. Chaque machine virtuelle est équipée de 32 Go de mémoire vive et de 800 Go d'espace disque, offrant un environnement adapté à l'évaluation expérimentale de notre solution.

5.2.2 EduVPN

La plateforme étant hébergée sur une infrastructure privée, nous avons utilisé le service eduVPN afin d'établir une connexion sécurisée avec le réseau interne. Un fichier de configuration VPN personnalisé nous a été fourni, Ce fichier nous a permis de nous connecter à distance aux ressources de la plateforme OpenStack via un tunnel sécurisé.

5.2.3 SSH et déploiement automatisé

Pour accéder aux machines virtuelles, nous avons utilisé le protocole SSH (Secure Shell), qui permet d'établir une session distante chiffrée entre deux machines. Ce protocole est largement utilisé pour l'administration de serveurs à distance. Nous avons automatisé à la fois le processus de connexion aux machines virtuelles et le déploiement des composants applicatifs (master et workers) à l'aide d'un script unique, `deploy.sh`. Ce script prend en charge la configuration réseau, le lancement des services, ainsi que la distribution des fichiers nécessaires à l'exécution du framework. L'ensemble du code et des scripts sont disponible sur notre dépôt GitHub.

5.3 Outils et technologies

Dans le cadre du développement de notre framework EGFDT, nous avons mobilisé un ensemble d'outils et de technologies adaptés à nos besoins. Cette section présente les principales technologies utilisées, accompagnées d'une brève description pour chacune d'elles.

5.3.1 Langage utilisé : Java

Le système PQDAG étant principalement développé en Java, l'équipe a naturellement souhaité que le framework EGFDT soit également conçu dans ce langage, afin de garantir une intégration cohérente et sans friction. Java est un langage de programmation orienté objet, multiplateforme, reconnu pour sa robustesse et sa large adoption dans le développement d'applications distribuées. Son architecture repose sur la machine virtuelle Java (JVM), qui permet l'exécution du même code source sur différentes plateformes, sans modification.

Dans notre implémentation, nous avons tiré parti de plusieurs bibliothèques Java essentielles :

— **java.net.Socket** : utilisée pour établir des communications réseau via le protocole TCP entre les différentes machines du système (master, senders et receivers).

— **java.util.concurrent** : fournit des outils avancés pour la gestion du parallélisme, notamment les exécuteurs de threads, les files de requêtes et les mécanismes de synchronisation.

— **Kryo** : bibliothèque de sérialisation Java rapide, légère et hautement performante. Elle permet de convertir efficacement des objets Java en flux binaires (sérialisation) et de les reconstituer ultérieurement (désérialisation), avec un surcoût minimal en termes de mémoire et de temps. Dans notre projet, l'intégration de Kryo nous a permis de découpler complètement la structure des objets PQDAG de l'étape de sérialisation/désérialisation.

5.4 Présentation de notre framework EGFDT

Dans cette section, nous présentons des captures d'écran illustrant les différents mécanismes de transfert réseau, en utilisant notre plateforme expérimentale, composée d'un nœud `master` et de dix nœuds `workers`.

5.4.1 Stratégies de Broadcast

TreeBroadcast

Le processus commence par le lancement du `MasterApp`, comme illustré dans la figure 5.1.

```
ubuntu@master:~$ java -jar MasterApp.jar

🚀 Master démarré sur le port 7506...
```

Fig. 5.1 – Lancement du Master

Dans chaque worker, nous lançons l'application `WorkerApp` (voir figure 5.2).

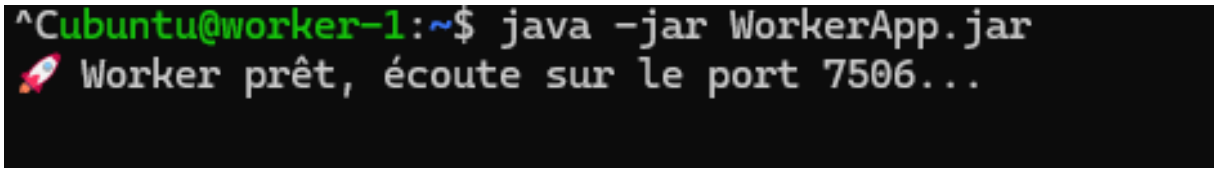

FIG. 5.2 – Lancement des workers

Une fois tous les workers enregistrés, le *master* construit l'arbre de diffusion (avec un nombre de fils d lu à partir du fichier de configuration), comme illustré dans la figure 5.3. Il envoie ensuite à chaque worker des messages contenant l'information sur sa position dans l'arbre, notamment s'il est un nœud intermédiaire ou une feuille. Par exemple, le worker 2 reçoit une instruction indiquant qu'il a trois enfants (voir figure 5.4), tandis que le worker 9 est informé qu'il est une feuille (voir figure 5.5).

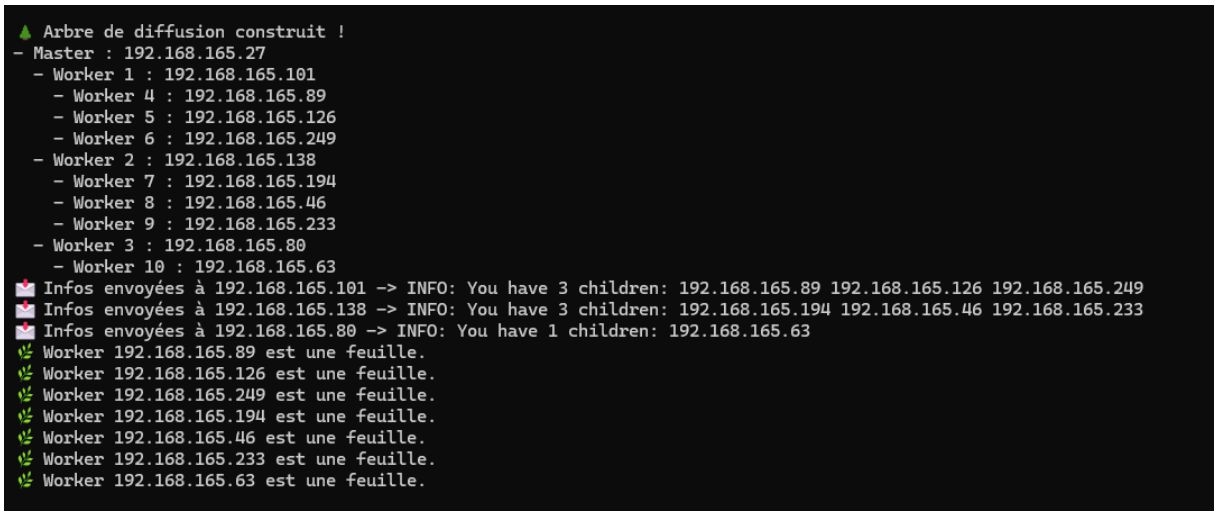

FIG. 5.3 – Arbre de diffusion construit(avec d=3) et informations envoyées aux workers

FIG. 5.4 – Message du Master au Worker 2 : 3 enfants assignés

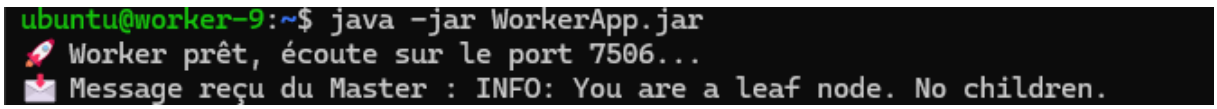

FIG. 5.5 – Message du Master au Worker 9 : nœud feuille (aucun enfant)

Lors de l'exécution du transfert, chaque worker envoie les blocs à ses enfants comme prévu. La figure 5.6 montre l'état final du master après la réception des accusés de réception de l'ensemble des workers.

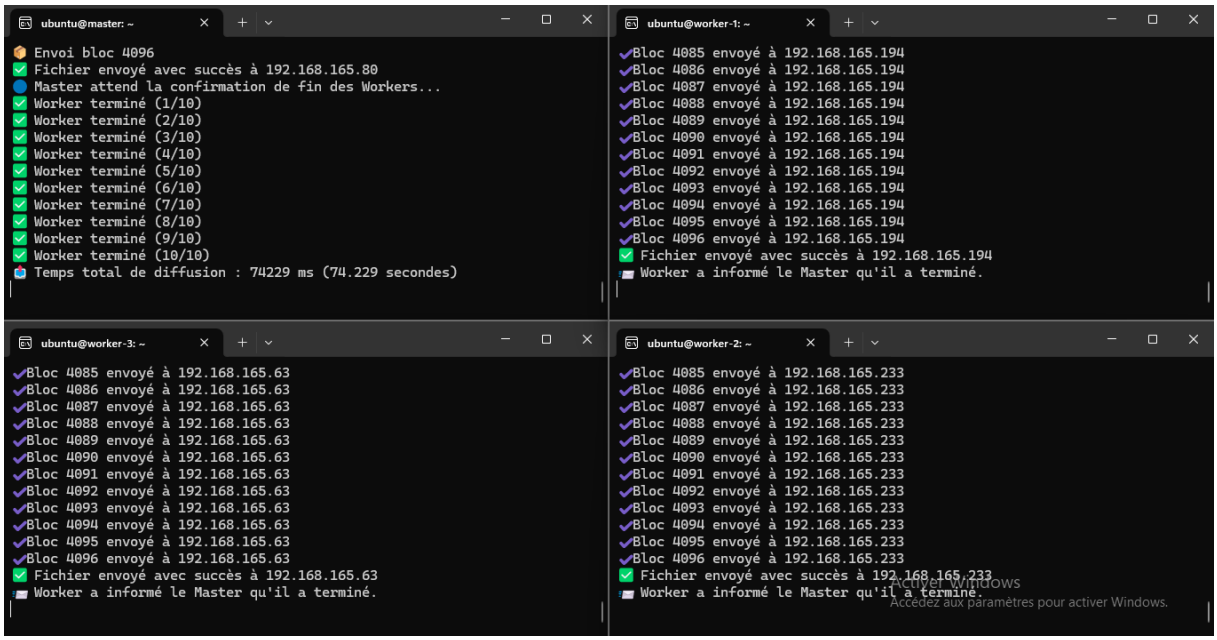

FIG. 5.6 – Diffusion complète Suivi côté Master et 3 Workers

Cornet

Le Master commence par découper le fichier source en plusieurs blocs de taille fixe, définie dans le fichier de configuration. Ces blocs sont ensuite distribués aux différents *workers* selon une politique *round-robin*, de manière à ce que chaque worker reçoive initialement un ou plusieurs blocs distincts. Une fois cette première phase de distribution terminée, le Master construit une table DLT répertoriant, pour chaque worker, la liste des blocs déjà reçus ainsi que ceux qui sont encore manquants (voir figure 5.7).

Table de la DLT :

Adresse IP Worker	Blocs détenus	Blocs manquants
192.168.165.101	[1, 11]	[2, 3, 4, 5, 6, 7, 8, 9, 10]
192.168.165.138	[2]	[1, 3, 4, 5, 6, 7, 8, 9, 10, 11]
192.168.165.80	[3]	[1, 2, 4, 5, 6, 7, 8, 9, 10, 11]
192.168.165.89	[4]	[1, 2, 3, 5, 6, 7, 8, 9, 10, 11]
192.168.165.126	[5]	[1, 2, 3, 4, 6, 7, 8, 9, 10, 11]
192.168.165.249	[6]	[1, 2, 3, 4, 5, 7, 8, 9, 10, 11]
192.168.165.194	[7]	[1, 2, 3, 4, 5, 6, 8, 9, 10, 11]
192.168.165.46	[8]	[1, 2, 3, 4, 5, 6, 7, 9, 10, 11]
192.168.165.233	[9]	[1, 2, 3, 4, 5, 6, 7, 8, 10, 11]
192.168.165.63	[10]	[1, 2, 3, 4, 5, 6, 7, 8, 9, 11]

FIG. 5.7 – Table DLT initiale

Au cours de la phase d'échange entre les machines, chaque réception de bloc par un worker déclenche l'envoi d'une notification au *Master*. Celui-ci met alors à jour la table DLT afin de refléter l'état actuel des blocs détenus et manquants par chaque machine worker. La Figure 5.8 illustre l'évolution de cette table après la réception de 60 notifications, mettant en évidence la propagation progressive des blocs au sein des workers.

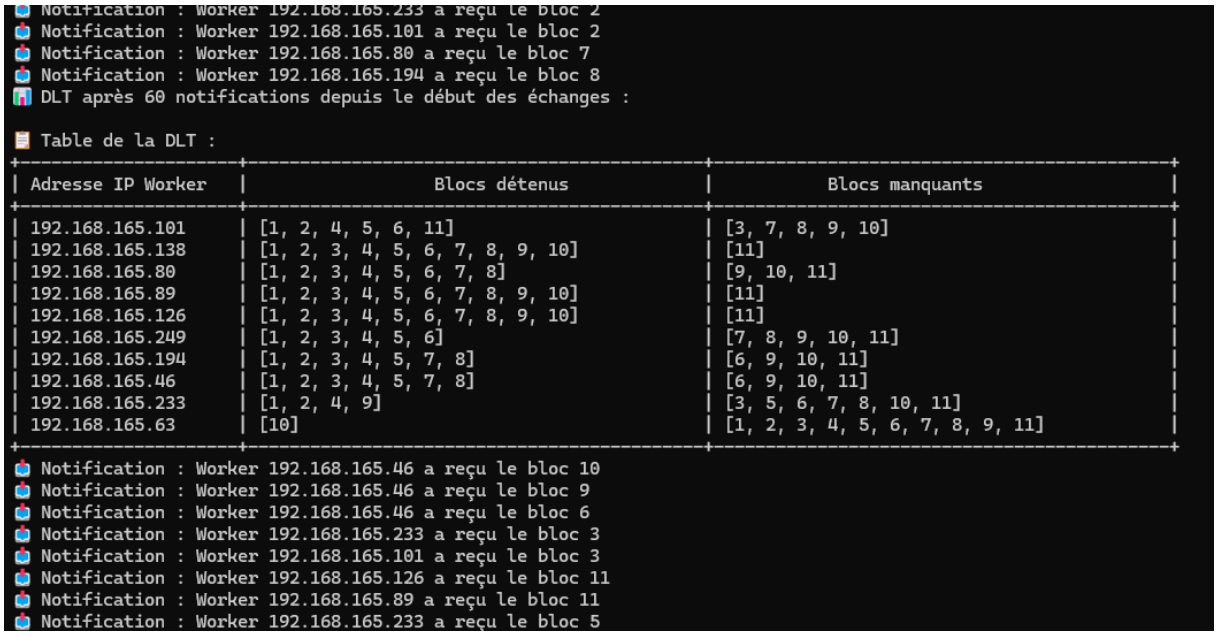

FIG. 5.8 – État intermédiaire de la DLT après 60 notifications d'échange de blocs

Une fois que tous les workers ont reçu l'ensemble des blocs, chacun reconstitue localement le fichier complet. Le master reçoit alors les accusés de réception, envoyés par chaque worker, ce qui marque la complétion du transfert. La figure 5.9 illustre l'état final de la DLT, dans lequel la colonne « Blocs manquants » est vide pour tous les workers, indiquant que tous disposent de l'intégralité des données, avec un temps total de transfert significativement meilleur que celui observé avec TreeBroadcast.

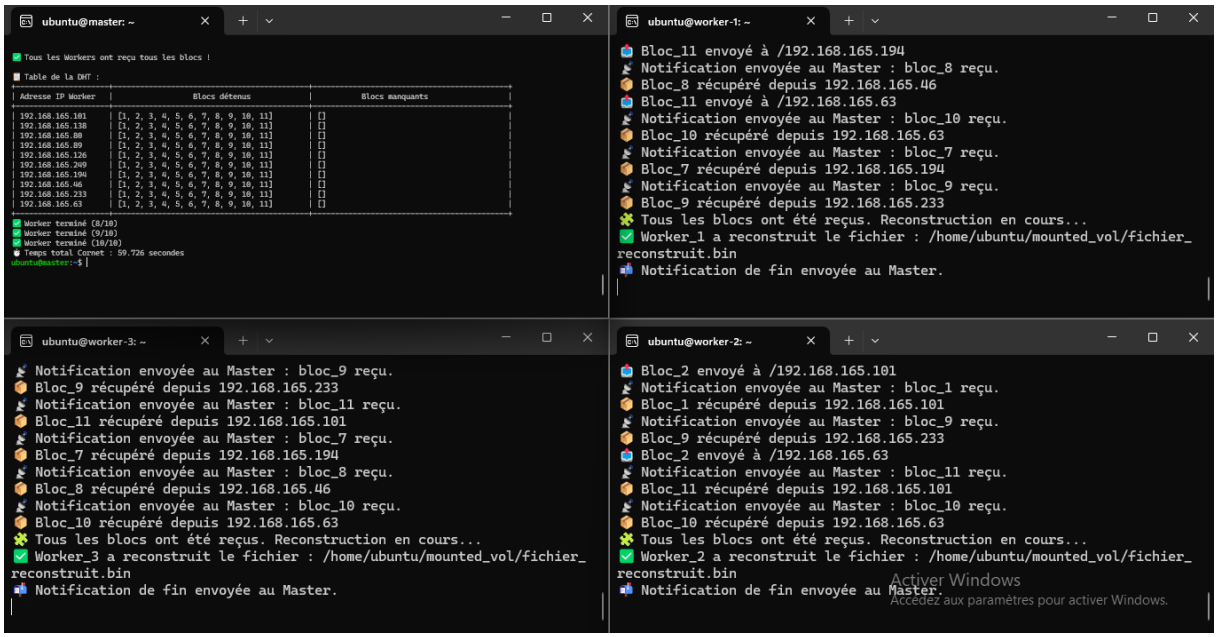

FIG. 5.9 – Fin du transfert

5.4.2 Shuffle

Nous présentons ici l'exécution du même exemple que celui utilisé dans le chapitre consacré à l'analyse et à la conception, en utilisant une configuration composée d'un

master et de six *workers* (trois *senders* et trois *receivers*).

Shuffle Équitable

Le processus commence par la génération d'un plan global par le *master*, où chaque *sender* connaît les destinataires (receivers) pour chacun de ses blocs (figure 5.10).

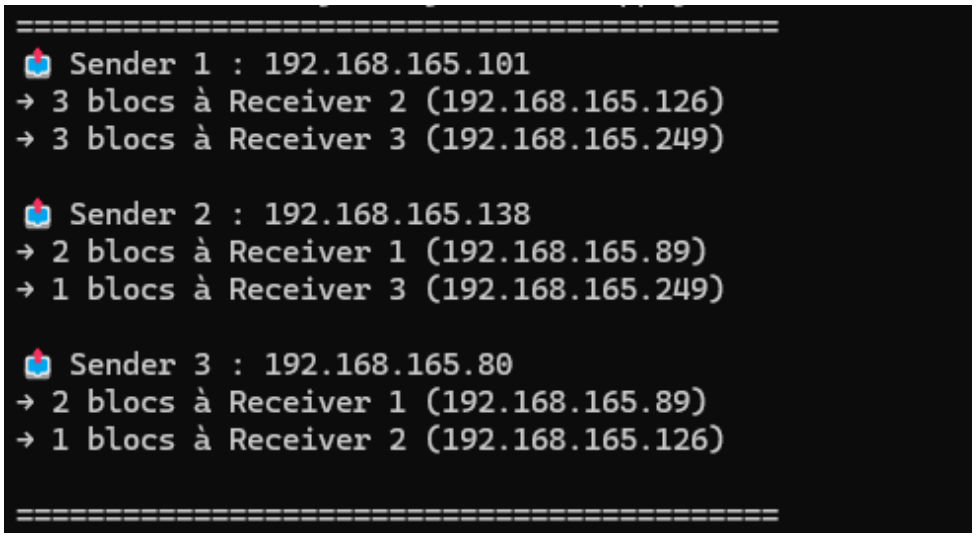

FIG. 5.10 – Plan global généré par le master

Ensuite, le master distribue les rôles : chaque machine reçoit l'instruction d'agir en tant que *sender* ou *receiver*. Chaque machine worker affiche alors le rôle qui lui est attribué (Voir les figures 5.11 et 5.12).

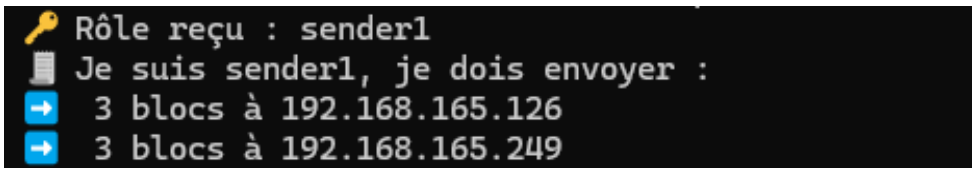

FIG. 5.11 – Exemple d'un sender

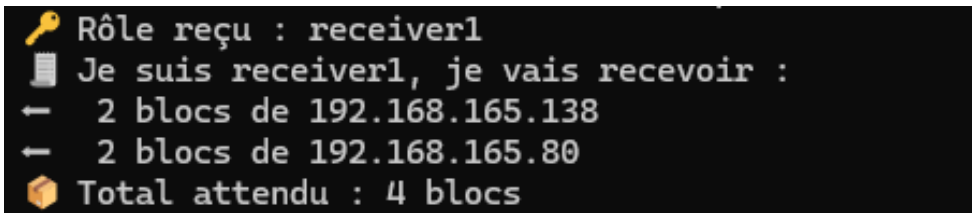

FIG. 5.12 – Exemple d'un receiver

Côté *receiver*, chaque machine construit une file de réception selon une stratégie équitable, qui répartit les blocs reçus de manière équilibrée entre les senders tout en respectant un ordre strict (Voir la figure 5.13).

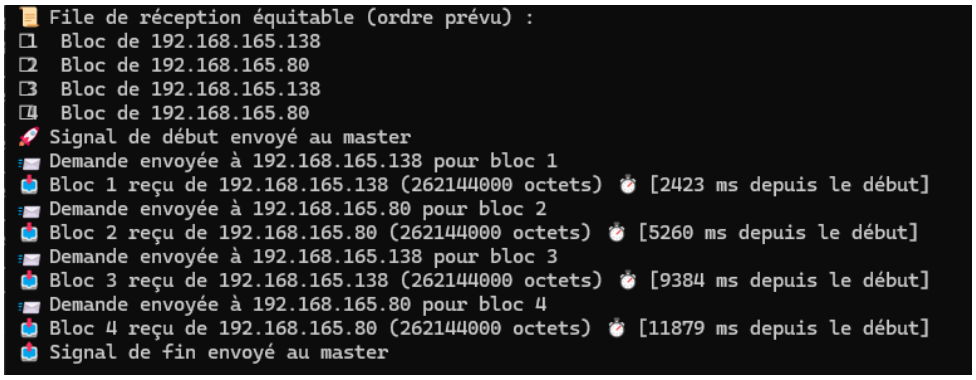

FIG. 5.13 – Construction de la file équitable et envoi des demandes aux senders

Chaque receiver envoie ensuite les requêtes aux senders selon cette file. Les senders, de leur côté, reçoivent les requêtes et envoient les blocs dans l'ordre d'arrivée (FIFO). Une fois tous les blocs reçus, chaque receiver envoie un accusé de réception au master. Le master, après réception des trois accusés de réception, conclut que le transfert est terminé. Le transfert est complété en 16,379 secondes.

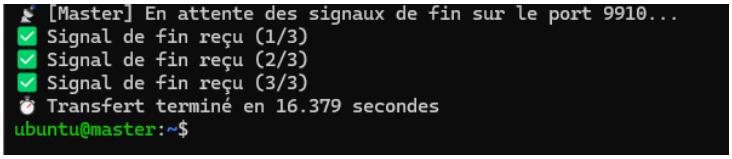

FIG. 5.14 – temps total du transfert équitable

Shuffle Prioritaire

Le processus reste identique à celui du *shuffle équitable*, nous n'en détaillons donc pas à nouveau les étapes. La seule différence réside dans le plan de réception, qui suit ici une stratégie basée sur la priorité. La figure 5.15 illustre un exemple de réception du côté d'un *receiver*, où le sender `192.168.165.101` est prioritaire, car il doit transmettre trois blocs, soit plus que les autres.

Bien que cette approche introduise une logique de priorité dans le traitement des envois, le temps de transfert total observé reste très proche de celui obtenu avec la stratégie équitable.

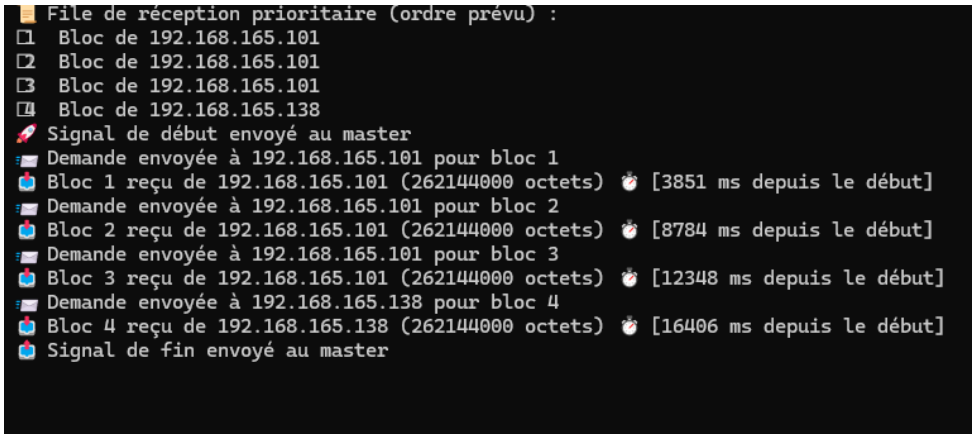

FIG. 5.15 – Exemple de réception prioritaire

Shuffle Matriciel Sans Conflit

Le *master* commence par construire un plan d'exécution basé sur une matrice sans conflit, dans laquelle chaque ligne représente une itération du transfert, et chaque colonne correspond à un plan d'ordonnancement d'un *receiver*, comme illustré dans la figure 5.16.

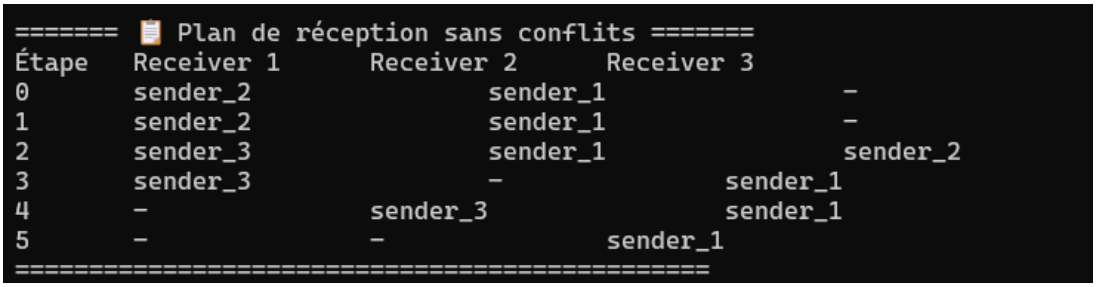

FIG. 5.16 – Plan d'exécution sans conflit généré par le master

Chaque *receiver* reçoit ensuite la colonne du plan qui lui est dédiée et construit localement sa propre file d'exécution, comme illustré dans la figure 5.17.

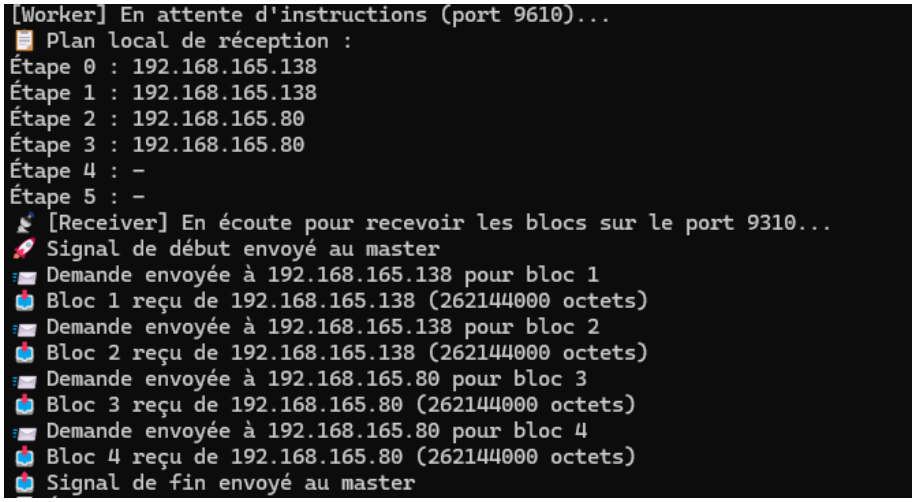

FIG. 5.17 – Exécution locale chez un receiver selon le plan sans conflit

Les *receivers* envoient alors leurs demandes aux *senders* en respectant strictement l'ordre de réception défini dans le plan d'exécution.

5.5 Conclusion

Dans ce chapitre, nous avons présenté l'ensemble des briques nécessaires à la conception, au développement et au déploiement du framework EGFDT. Nous avons illustré, à l'aide de captures d'écran, l'exécution des différents mécanismes de transfert proposés. Le chapitre suivant viendra clore notre projet de fin d'études en dressant une synthèse des contributions apportées et en proposant quelques perspectives pour la suite de ce travail.

CHAPITRE 6

CONCLUSION GÉNÉRALE

6.1 Conclusion

Le volume de données disponible sur le Web a connu une explosion phénoménale, générant une masse d'informations sans précédent. La gestion efficace de ces données est ainsi devenue une nécessité incontournable. De nombreux systèmes de gestion de données ont été proposés et développés, la plupart mettant l'accent sur des composantes telles que l'optimisation des requêtes, la compression des données ou encore le partitionnement. En revanche, la question du transfert de données reste relativement peu documentée dans la littérature, et rares sont les approches qui implémentent leur propre framework de transfert. La majorité des systèmes s'appuient en effet sur des technologies externes comme Apache Spark ou MPI.

L'équipe PQDAG, hébergée au laboratoire LIAS de l'ISAE-ENSMA, adopte une approche différente : elle vise un contrôle total sur les composants qu'elle développe, en privilégiant des solutions construites from scratch. En collaboration avec le laboratoire LRIT de l'Université de Tlemcen, l'équipe développe PQDAG, un système de gestion de données RDF distribué à grande échelle. Toutefois, la composante de transfert initialement intégrée dans PQDAG présentait certaines limitations, ouvrant ainsi des perspectives claires d'amélioration.

C'est dans ce contexte que s'inscrit notre projet de fin d'études, qui avait pour objectif de concevoir et d'améliorer les mécanismes de transfert de données au sein de PQDAG. Concrètement, nous avons implémenté : (1) une approche de diffusion appelée Cornet, (2) deux stratégies de shuffle : l'une prioritaire et lautre basée sur une matrice sans conflit, (3) un protocole de sérialisation/désérialisation efficace.

Tout au long du projet, nous avons mis en œuvre les compétences acquises au cours de notre formation : gestion de la bande passante, utilisation de sockets pour la communication inter-machines, optimisation des échanges de données, etc. Nous avons également déployé nos solutions sur une plateforme OpenStack, en créant et configurant nous-mêmes les machines virtuelles nécessaires. Cette expérience nous a permis de mieux comprendre les enjeux concrets liés à la distribution de données à grande échelle.

6.2 Perspectives

Bien entendu, notre travail reste perfectible et ouvre plusieurs perspectives intéressantes. Nous espérons que l'équipe PQDAG pourra intégrer le framework que nous avons développé, EGFDT, au sein du système PQDAG, et évaluer l'efficacité réelle des solutions implémentées. Il convient de noter que nous avons déjà fourni une validation expérimentale encourageante dans ce contexte.

Une seconde perspective consisterait à explorer l'intégration de techniques d'intelligence artificielle afin d'aider automatiquement PQDAG à choisir la stratégie de transfert la plus adaptée (shuffle ou broadcast) en fonction des caractéristiques de l'environnement ou de la charge système, afin de garantir une flexibilité et une efficacité accrues.

Bibliographie

Bibliographie

[1] Razen Al-Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB Journal*, 25(3) :355--380, 2016.

[2] Razen Al-Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB Journal*, 25(3) :355--380, 2016.

[3] Xi Chen, Huajun Chen, Ningyu Zhang, and Songyang Zhang. Sparkrdf : elastic discreted rdf graph processing engine with distributed memory. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 1, pages 292--300. IEEE, 2015.

[4] Sai Krishnan Chirravuri. Rdf3x-mpi : A partitioned rdf engine for data-parallel sparql querying. 2014.

[5] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM computer communication review*, 41(4) :98--109, 2011.

[6] Orri Erling and Ivan Mikhailov. Virtuoso : Rdf support in a native rdbms. In *Semantic Web Information Management*. Springer, 2009.

[7] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 120--131. ACM, 1993.

[8] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. Triad : a distributed shared-nothing RDF engine based on asynchronous message passing. In *ACM SIGMOD*, pages 289--300, 2014.

[9] Aidan Hogan, Eva Blomqvist, Mathieu Cochez, and et al. Knowledge graphs. *ACM Computing Surveys*, 54(4) :1--37, 2021.

[10] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. volume 20, pages 359--392, 1999.

[11] Abdallah Khelil, Amin Mesmoudi, Jorge Galicia, Ladjel Bellatreche, Mohand-Saïd Hacid, and Emmanuel Coquery. Combining graph exploration and fragmentation for scalable rdf query processing. *Inf. Syst. Frontiers*, 23(1) :165--183, 2021.

[12] Thomas Neumann and Guido Moerkotte. Characteristic sets : Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, pages 984--994, 2011.

[13] Thomas Neumann and Gerhard Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19:91--113, 2010.

[14] Peng Peng, M Tamer Özsu, Lei Zou, Cen Yan, and Chengjun Liu. Mpc : minimum property-cut rdf graph partitioning. In *IEEE ICDE*, pages 192--204, 2022.

[15] Apache Jena Project. Apache jena tdb. https://jena.apache.org/documentation/tdb/, 2024. Accessed : 2024-12-15.

[16] Kurt Rohloff, Richard Schantz, and et al. Triad : A distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[17] Andreas Schatzle, Martin Przyjaciel-Zablocki, and Georg Lausen. S2rdf : Rdf querying with sparql on spark. volume 9, pages 804--815, 2016.

[18] Bryan Thompson, Andrey Person, Dave Petty, Jesus Barrasa, and et al. Blazegraph : Dgraph database for the semantic web. In *ISWC Developers Workshop*, 2014.

[19] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103--111, 1990.

[20] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103--111, 1990.

Table des figures

Liste des tableaux

Table des abréviations

RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
LOD	Linked Open Data
BSP	Bulk Synchronous Parallel
MPI	Message Passing Interface
MPP	Massively Parallel Processing
RDD	Resilient Distributed Dataset
DLT	Distributed Lookup Table
PQDAG	Parallel Query Data As Graph
QDAG	Query Data As Graph
EGFDT	Enhanced Generic Framework For Data Transfer
LIAS	Laboratoire d'Informatique et d'Automatique pour les Systèmes
ENSMA	École Nationale Supérieure de Mécanique et d'Aérotechnique
LRIT	Laboratoire de Recherche en Informatique et Télécommunication
HDFS	Hadoop Distributed File System

Résumé / Abstract / ملخص

Résumé : L'explosion des données est devenue le centre des préoccupations de plusieurs secteurs, économiques, scientifiques et sociétaux. Dans ce contexte, de nombreux défis sont lancés dans le but de gérer la collection, le traitement et l'exploration de ces données. Les travaux de ce projet de fin d'études s'intègrent dans le cadre du système de gestion de données RDF distribué PQDAG, développé au sein du laboratoire LIAS (ISAE-ENSMA). Notre objectif consistait à analyser les limites du mécanisme de transfert initial de PQDAG, puis à concevoir une solution plus modulaire et performante. Ainsi, nous avons proposé un nouveau framework baptisé **EGFDT** (Enhanced Generic Framework for Data Transfer), permettant de gérer les transferts de données de manière efficace et extensible. Ce framework intègre plusieurs stratégies de transfert efficaces, notamment le *broadcast* et le *shuffle*, qui sont omniprésentes dans le système PQDAG. Toutefois, sa conception générique lui permet de s'intégrer facilement à d'autres systèmes distribués présentant une architecture similaire à celle de PQDAG. L'efficacité de notre solution a été validée expérimentalement à travers une série de tests réalisés sur une infrastructure OpenStack.

Mots-clés : Système de gestion de données, Transfert de données, EGFDT, Scalabilité.

Abstract : The explosion of data has become the focus of several sectors, including economic, scientific and societal domains. In this context, many challenges have emerged to manage the collection, processing and exploration of such data. This graduation project is part of the distributed RDF system PQDAG, developed within the LIAS laboratory (ISAE-ENSMA). Our main objective was to analyze the limitations of PQDAGs initial data transfer mechanism and propose a more modular and efficient solution. We designed a new framework called **EGFDT** (Enhanced Generic Framework for Data Transfer), capable of managing data transfers in a scalable and extensible way. The framework supports optimized strategies (broadcast, shuffle) and can be easily integrated into distributed systems like PQDAG. With the support of PhD students from the IDD team at LIAS, we experimentally validated our solution using an OpenStack-based infrastructure.

Keywords : Data Management System, Data transfer, EGFDT, Scalability.

ملخص: أصبحت انفجار البيانات محط اهتمام العديد من القطاعات الاقتصادية والعلمية والمجتمعية. في هذا السياق، ظهرت تحديات عديدة تهدف إلى إدارة جمع البيانات ومعالجتها واستكشافها. يندرج هذا العمل ضمن نظام PQDAG الموزع، المطوّر داخل مختبر LIAS (ISAE-ENSMA). كان هدفنا الرئيسي هو تحليل حدود آلية النقل الأصلية لـ PQDAG ثم اقتراح حل أكثر مرونة وكفاءة. لذلك، قمنا بتصميم إطار جديد يُدعى **EGFDT**، قادر على إدارة نقل البيانات بطريقة فعالة وقابلة للتوسيع. يعتمد الإطار على استراتيجيات محسّنة (البث، التبديل) ويمكن دمجه بسهولة في أنظمة موزعة مثل PQDAG. بدعم من طلاب الدكتوراه في فريق IDD بـ LIAS، قمنا بالتحقق من فعالية الحل من خلال اختبارات على بنية تحتية تعتمد على OpenStack.

الكلمات المفتاحية: نظام إدارة البيانات، نقل البيانات،EGFDT، قابلية التوسع.