Thèse de Doctorat

Pauline FOLZ

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
Docteur de Nantes Métropole
Label européen

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications
Spécialité : Informatique
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)

Soutenue le 12 octobre 2017

Collaboration dans une fédération de consommateurs de données liées

JURY

Président :	**M. Marc GELGON**, Professeur, Université de Nantes
Rapporteurs :	**M^me^ Catherine FARON-ZUCKER**, Maître de conférences / HDR, Université de Sophia Antipolis **M^me^ Esther PACCITI**, Professeur, Université de Montpellier
Examinateur :	**M. Hubert NAACKE**, Maître de Conférences, Université Paris 6
Invité :	**M. Hervé JAIGU**, Chargé de mission Innovation, Nantes Métropole
Directeur de thèse :	**M. Pascal MOLLI**, Professeur, Université de Nantes
Co-encadrante de thèse :	**M^me^ Hala SKAF-MOLLI**, Maître de Conférences, Université de Nantes

Remerciements

Je remercie Pascal Molli et Hala Skaf pour m'avoir donné l'opportunité de faire cette thèse et de m'avoir accompagné. Ainsi que les autres membres de l'équipe GDD pour leur accueil.

Je remercie Nantes Métropole, pour avoir financer ma thèse. La Direction Recherche, Innovation & Enseignement Supérieur pour m'avoir accueillie pendant ces trois ans. Et plus particulièrement Hervé Jaigu, pour ses conseils et sa bienveillance.

Je remercie mes collègues doctorants pour les moments partagés. Je ne me risque pas à citer de noms de peur d'en oublier, mais vous vous reconnaîtrez. Je fais une exception pour mes collègues de bureaux, Gabriela pour ses conseils et sa collaboration, et Brice pour son soutien.

Enfin et surtout, je tiens à remercier mes sœurs : Emma, Chloé et Anaïs, ainsi que ma mère, pour leur soutien tout au long de ma thèse.

Table des matières

1

Introduction

Sommaire

Le Web des données étend le Web en associant à une adresse Web unique (un URI) un document RDF (*Resource Description Framework*) [37]. Par exemple, l'URI `http://fr.dbpedia.org/page/La_Naissance_de_Vénus_(Botticelli)` renvoie un document RDF décrivant le tableau «La naissance de Vénus» selon DBpedia. Ce document RDF contient un ensemble de faits décrivant ce tableau. Un fait est représenté sous forme d'un triplet : ⟨ `sujet, prédicat, objet` ⟩.

```
Sujet : http://fr.dbpedia.org/page/La_Naissance_de_Vénus_(Botticelli)
Prédicat : http://dbpedia.org/ontology/author
Objet : http://fr.dbpedia.org/page/Sandro_Botticelli
```

Le fait ci-dessus établit donc que le tableau «La Naissance de Venus» a été peint par Sandro Botticelli. En déréférençant `http://fr.dbpedia.org/page/Sandro_Botticelli`, on obtient un document RDF décrivant le peintre Sandro Botticelli selon DBpedia. En déréférençant `http://dbpedia.org/ontology/author`, on obtient la définition du prédicat «author» elle-même exprimée en RDF.

À l'image du Web classique, le Web des données est donc navigable. Il peut être exploré aussi bien par un humain que par une machine. Le Web des données est un multi-graphe orienté et étiqueté, distribué à travers un grand nombre de serveurs Web indépendants et

autonomes. Chaque triplet correspond à un arc orienté où le label est le prédicat, le sujet est le nœud source et l'objet le nœud cible.

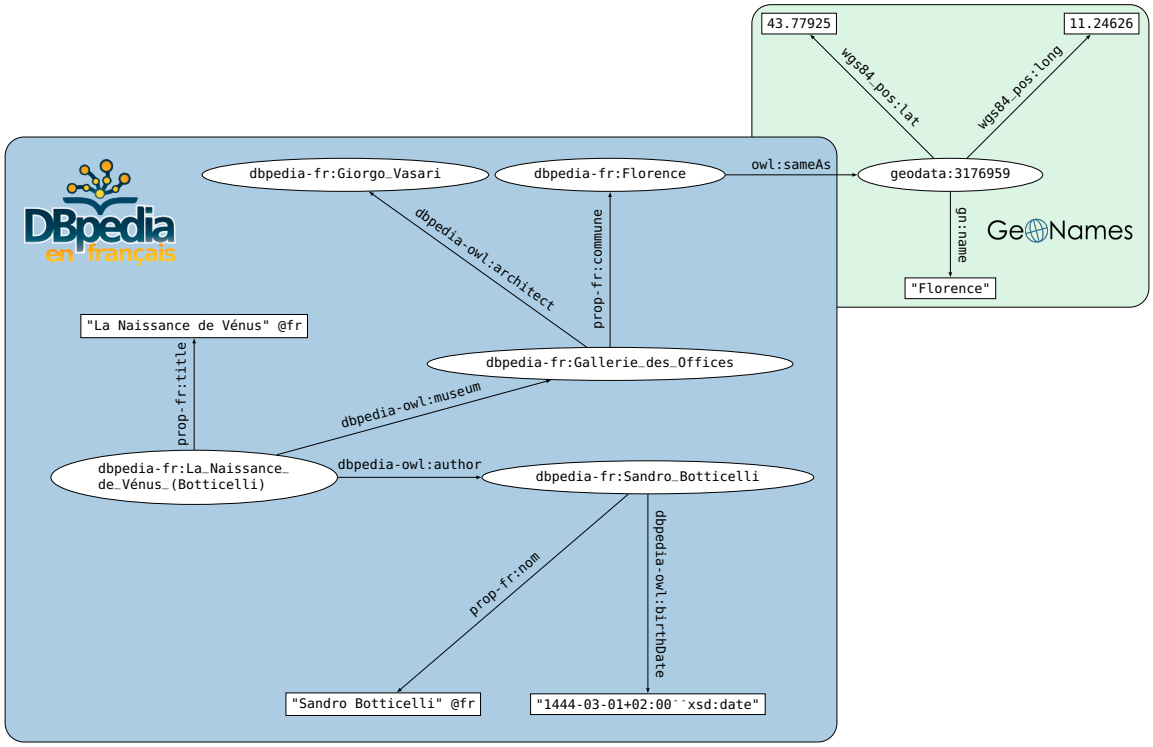


FIGURE 1.1 – Extrait de graphes RDF provenant de DBpediaFr (bleu) et de Geonames (vert).

La figure 1.1 montre comment les données de DBpediaFr à propos du tableau la «Naissance de Venus» sont liées avec des données disponibles sur le site Geonames[1]. Les données en bleu sont extraites de DBpediaFr et les données en vert sont extraites de Geonames. DBpedia nous apprend que «La Naissance de Venus» est exposée dans la galerie des offices à Florence. Elle nous apprend également que «Florence» est aussi définie sur Geonames avec comme identifiant `geodata:317659`. Ainsi, en naviguant de DBpedia à Geonames, il est possible de découvrir les coordonnées GPS de Florence.

```
PREFIX dpedia-owl: <http://dbpedia.org/ontology/museum>
PREFIX dbpedia-fr: <http://fr.dbpedia.org/resource/>
PREFIX prop-fr: <http://fr.dbpedia.org/property/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX geodata: <http://sws.geonames.org/>

dbpedia-fr:La_Naissance_de_Vénus_(Botticelli) prop-fr:titre "La Naissance
    de Vénus @fr" .
dbpedia-fr:La_Naissance_de_Vénus_(Botticelli) dbpedia-owl:museum dbpedia-
   fr:Galerie_des_Offices .
dbpedia-fr:La_Naissance_de_Vénus_(Botticelli) dbpedia-owl:author dbpedia-
   fr:Sandro_Botticelli.
dbpedia-fr:Galerie_des_Offices dbpedia-owl:architect dbpedia-fr:
   Giorgo_Vasari .
dbpedia-fr:Galerie_des_Offices prop-fr:commune dbpedia-fr:Florence .
dbpedia-fr:Sandro_Botticelli prop-fr:nom "Sandro Botticelli @fr" .
dbpedia-fr:Sandro_Botticelli dbpedia-owl:birthDate "1444_03_01+02:00 (xsd
   :date)" .
```

[1] http://www.geonames.org

```
dbpedia-fr:Florence owl:sameAs geodata:317659

PREFIX geodata: <http://sws.geonames.org/>
PREFIX gn: <http://www.geonames.org/ontology#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>

geodata:317659 gn:name "Florence" .
geodata:317659 wgs84_pos:lat 43.77925 .
geodata:317659 wgs84_pos:long 11.24626
```

Listing 1.1 – Représentation en N-triples des données représentées dans la figure 1.1.

L'ensemble de ce graphe au format N-triples est décrit dans le listing 1.1. L'association entre Florence et les données de GeoNames est réalisée sur le site DBpedia avec le fait : `dbpedia-fr:Florence owl:sameAs geodata:317659`.

En suivant les principes des données liées [10], les producteurs de données ont mis à disposition des milliards de triples sur le Web, nombre en constante augmentation [51]. Les principes ont été énoncés par Tim Berners-Lee de la manière suivante :

- Utiliser des URIs pour nommer les «choses».
- Utiliser des URIs HTTP pour que les personnes puissent les consulter.
- Les URIs fournissent des informations utiles en suivant les standards du Web (RDF, SPARQL).
- Inclure des liens vers d'autres URIs, pour découvrir d'autres «choses».

La figure 1.2 présente le diagramme du *Linked Open Data cloud* (LOD) qui référence les sites Web les plus importants respectant ces principes. Pour être intégré au diagramme du LOD, présenté en figure 1.2, il faut un jeu de données d'au moins 1000 tuples et 50 liens vers un jeu de données déjà présent dans le LOD ou depuis le LOD vers le jeu de données en question. La figure 1.2 est datée du 26 janvier 2017 et comprend 1146 jeux de données de multiples domaines.

En respectant les principes des données liées, le Web des données forme un espace de données global unique [10] géré par des sites Web autonomes. Il peut être interrogé avec un langage de requête.

1.1 Interroger le Web des données

Le langage SPARQL (*Simple Protocol And RDF Query Language*) [48] permet d'interroger des données RDF.

```
PREFIX dcterms: <http://purl.org/dc/terms/>

SELECT ?titre
WHERE {
 ?oeuvre dcterms:subject <http://fr.dbpedia.org/resource/Catégorie:
    Œuvre_conservée_à_la_Galerie_des_Offices> .
  ?oeuvre  prop-fr:titre ?titre .
  ?oeuvre  dbpedia-owl:author dbpedia-fr:Sandro_Botticelli
}
```

Listing 1.2 – Requête SPARQL SELECT : Œuvres de Sandro Botticelli conservées à la Galerie des Offices.

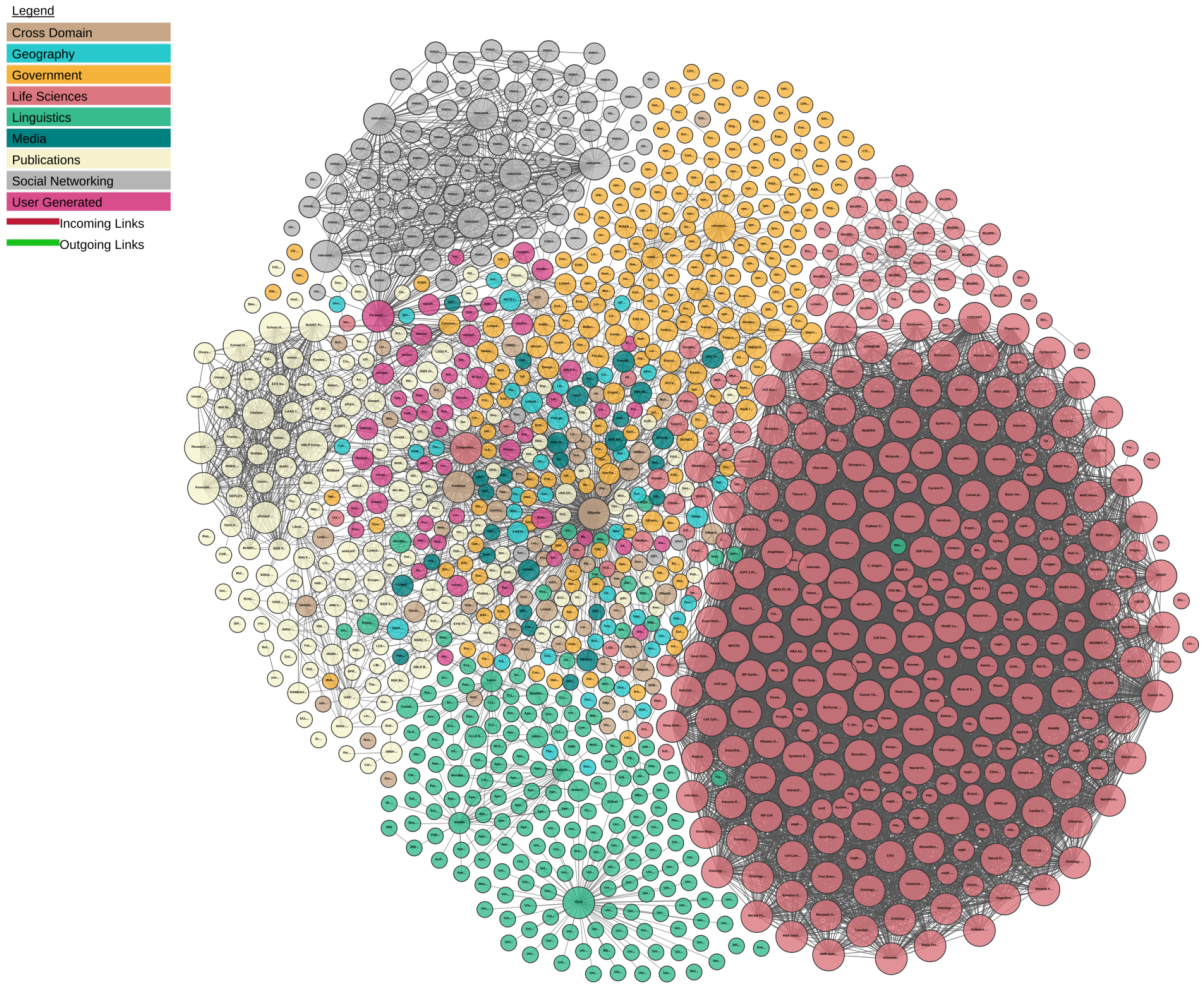


FIGURE 1.2 – Diagramme du *Linked Open Data cloud* du 26 janvier 2017. Il comprend 1146 jeux de données.

La requête SPARQL du listing 1.2 retourne les œuvres de Sandro Botticelli qui sont conservées à la Galerie des Offices. Si on exécute cette requête sur le serveur SPARQL disponible à l'adresse suivante : `http://fr.dbpedia.org/sparql`, cette requête retourne les résultats suivants :

```
"Pallas et le Centaure"@fr
"La Vierge à la grenade"@fr
"Portrait d'homme avec médaille de Cosme l'ancien"@fr
"La Naissance de Vénus"@fr
"La Calomnie d'Apelle"@fr
"Le Printemps"@fr
"L'Adoration des mages"@fr
"L'Annonciation du Cestello"@fr
"La Madone du Magnificat"@fr
"La Découverte du cadavre d'Holopherne"@fr
"Le Retour de Judith à Béthulie"@fr
"Botticelli 1444/45–1510"@fr
"Botticelli, in I protagonisti dell'arte italiana"@fr
"Botticelli. De Laurent le Magnifique à Savonarole"@fr
"Saint Augustin dans son cabinet de travail"@fr
"La Force"@fr
```

```
"La Vierge de la loggia"@fr
"La Vierge et l'Enfant dans une gloire de séraphins"@fr
"La Vierge à la roseraie"@fr
"Retable de Saint–Ambroise"@fr
"Le retable de San Marco"@fr
"ou Couronnement de la Vierge"@fr
"L'Annonciation de San Martino alla Scala"@fr
```

Pour pouvoir exécuter cette requête, l'ensemble des faits RDF disponibles sur le site fr.dbpedia a été, au préalable, inséré dans une base de données RDF (Virtuoso dans ce cas) par les hébergeurs du site fr.dbepdia.org. Les résultats sont donc complets si l'on ne considère que les données de fr.dbpedia.org. Ils ne le sont peut-être pas si l'on doit considérer l'ensemble du Web des données.

En fait, l'évaluation de requêtes SPARQL sur le Web des données pose plusieurs problèmes complexes :

Localisation des données Le premier est de définir un périmètre sur lequel exécuter les requêtes. Le périmètre est l'ensemble des sites Web à considérer pour évaluer une requête. Faut-il prendre le Web des données dans son ensemble ou juste considérer un sous-ensemble des données ; par exemple : données bio-médicales, données gouvernementales, etc.

Performances et passage à l'échelle Sur le Web des données, le volume des données ainsi que le nombre de requêtes par seconde peuvent être très élevé. De plus, SPARQL est un langage de requête très expressif dont l'évaluation peut être coûteuse. Enfin, les données sont naturellement distribuées et partitionnées par site Web. Selon les infrastructures considérées pour évaluer les requêtes, il va exister un compromis entre performances et passage à l'échelle/disponibilité des données. Ce compromis existe également sur le coût financier de l'évaluation des requêtes. L'évaluation est-elle exclusivement à la charge des producteurs de données ? Ou des consommateurs de données ? Et/ou d'éventuels médiateurs entre producteurs de données et consommateurs de données ?

Hétérogénéité sémantique Si le Web des données est homogène dans les formats utilisés pour représenter les données, il est hétérogène d'un point de vue sémantique. En effet, le Web des données ne définit pas un schéma strict permettant d'écrire une requête. Rien ne dit qu'une personne sera décrite en RDF de la même manière par un site et par un autre. Dans ce contexte, comment faire pour écrire une requête ?

Confiance Selon les sources sélectionnées pour effectuer la requête, un utilisateur peut avoir plus ou moins confiance envers telle ou telle source. Si l'utilisateur ne peut pas choisir les sources sur lesquelles il effectue ses requêtes, peut-il alors faire confiance aux résultats retournés ?

Usage Selon que les données sont répliquées ou non, un producteur de données voit ou pas les requêtes effectuées sur ses données. Si l'utilisateur interroge les données originales, c'est-à-dire sur le serveur de celui qui a autorité sur les données, le producteur de données peut savoir comment sont utilisées ses données.

Fraîcheur et cohérence des données Il est important que lorsque l'utilisateur effectue ses requêtes les données soient à jour et cohérentes avec les données originales, s'il s'agit de copies.

Dans une certaine mesure ces problèmes sont liés. On peut en effet considérer que plus le périmètre d'une requête est grand, plus l'hétérogénéité sémantique est grande et plus les problèmes de passage à l'échelle sont importants.

Il existe plusieurs approches pour pouvoir interroger en SPARQL le Web des données : entrepôt de données, indexes centralisés, médiateurs, gestion de données sur réseaux pair-à-pair. Ces différentes approches offrent différents compromis entre périmètre de la requête, performances, passage à l'échelle, hétérogénéité sémantique, confiance, usage, fraîcheur et cohérence des données.

1.2 Approche

Dans cette thèse, nous nous concentrons sur l'approche «médiateur». Les médiateurs [65] sont définis dans le cadre des bases de données fédérées [54] et plus particulièrement des moteurs de requêtes fédérées [1, 53]. Dans notre contexte, un médiateur permet d'exécuter des requêtes SPARQL sur un ensemble de serveurs hébergeant des données RDF et offrant tous la même interface : SPARQL ou un sous-ensemble de SPARQL. La requête n'est donc pas exécutée sur l'ensemble du Web des données mais sur un sous-ensemble déclaré par l'utilisateur final et envers lequel il a confiance. Ces moteurs s'exécutent sur les machines des utilisateurs finaux ou d'intermédiaires à travers des portails.

Dans cette approche «médiateur», on considère que l'hétérogénéité sémantique entre les différentes sources est suffisamment faible pour ne pas recourir à des méthodes d'alignement et de réécriture de requêtes [20].

Aujourd'hui, un médiateur est devenu si compact qu'il peut fonctionner au sein d'un navigateur Web [63]. Il est donc tout à fait possible d'avoir, à un moment donné, des milliers de médiateurs exécutant des requêtes SPARQL sur les mêmes sources de données. Si les sources de données se retrouvent surchargées, alors les données ne seront plus disponibles. Il existe un compromis entre performances des requêtes et disponibilité des données [63]. Suivant les opérations exécutées au sein des médiateurs et celles déléguées aux sources de données, on obtient des solutions qui ménagent les serveurs mais dégradent les performances ou des solutions qui améliorent les performances mais limitent la disponibilité des données. Cependant, ce compromis entre performance et disponibilité est établi avec l'hypothèse qu'un médiateur ne communique pas avec un autre. Dans cette thèse, nous nous posons la question suivante :

> Si les médiateurs collaborent en partageant leurs ressources en : espace disque, capacité de calcul et bande passante ; est-il possible d'obtenir un meilleur compromis entre performances et disponibilité des données ?

Ce problème est important car il est crucial d'avoir à la fois performances des requêtes et disponibilité pour pouvoir écrire des applications sur le Web des données. Ce problème est difficile car toute collaboration nécessite un coût supplémentaire pour connecter et coordonner les différents participants. Il faut donc s'assurer que collaborer pour accéder aux données soit plus efficace en terme de performances ou de disponibilité des données que de ne pas collaborer.

De nombreux travaux portant sur la gestion de données sur réseaux pair-à-pair ont montré qu'il est possible de gérer de manière efficace des données au sein d'un réseau pair-à-

pair [14, 30, 42]. Cependant, dans notre contexte, un utilisateur a toujours le choix d'exécuter sa requête sans passer par ses voisins. Comment lui assurer qu'il sera toujours gagnant s'il collabore ?

1.3 Contributions

Les deux contributions de cette thèse reposent sur une approche médiateur basée sur des serveurs *Triple Pattern Fragment* (TPF) [63] où les médiateurs de données sont connectés entre eux et collaborent.

Cyclades connecte des médiateurs TPF de façon à partager leur cache local. En effet, les performances de TPF sont étroitement liées aux caches : un cache Web situé sur le serveur TPF et un cache local au client. Malheureusement, comme les clients TPF ne collaborent pas, le cache local n'est pas partagé. Nous proposons CyCLaDEs [24], un cache décentralisé construit grâce à un réseau superposé basé sur la similarité des fragments TPF. Pour chaque client TPF, CyCLaDEs construit un voisinage de clients TPF hébergeant des fragments similaires dans leur cache local. Pendant l'exécution de la requête, le cache du voisinage est vérifié avant de demander au serveur TPF. Les résultats expérimentaux montrent que CyCLaDEs est capable de prendre en charge une partie significative des appels HTTPs et fournit un cache plus spécifique du côté des clients.

Ladda connecte les médiateurs TPF de façon à partager leurs capacités de calcul. TPF transfère partiellement le coût d'exécution des requêtes SPARQL des serveurs aux clients, ce qui augmente significativement le nombre de ressources capable d'exécuter des requêtes SPARQL. Cependant, ces nouvelles ressources sont sous-exploitées, parce que les clients TPF ne partagent pas leurs capacités d'exécution. Nous proposons Ladda, un système qui construit une fédération de clients TPF où chaque client est connecté à un nombre fixe de voisins. Un client TPF parallélise à la volée l'exécution de ses requêtes entrantes, en déléguant certaines d'entre elles à ses voisins. Ladda implémente un équilibreur de charge dynamique qui permet le parallélisme inter-requêtes sur les consommateurs de données. Nous avons évalué Ladda de façon expérimentale avec des journaux réels de DBpedia [5]. Les résultats suggèrent que la délégation réduit significativement le temps global d'exécution de la fédération et améliore le débit d'exécution des requêtes dans la fédération.

1.4 Plan du manuscrit

Le chapitre 2 (p. 17) présente l'état de l'art de cette thèse. Il détaille les différentes manières d'interroger le Web des données : requête de «parcours de liens», indexes centralisés, gestion de données sur réseau pair-à-pair, moteur de requêtes fédérées et l'approche *Triple Pattern Fragment* (TPF). Ainsi, que le positionnement de notre approche et sa justification.

Le chapitre 3 (p. 37) décrit dans un premier temps les techniques de mise en cache dans le Web sémantique, du côté des producteurs de données [38, 45, 66] et du côté des clients [31, 57]. Puis les caches collaboratifs créés dans le domaine des systèmes répartis [12, 18] et dans

les réseaux de diffusion de contenu [23, 25, 34]. Ce chapitre détaille CyCLaDEs [24], un cache collaboratif pour les fragments de triplets. Ce cache collaboratif est constitué de clients TPF et est construit du côté des consommateurs de données. Le reste du chapitre présente des expériences afin de valider l'approche.

Le chapitre 4 (p. 55) présente Ladda, une approche qui permet la parallélisation inter-requêtes grâce à la délégation de requêtes dans une fédération de consommateurs de données liées. Dans cette approche, un consommateur de données peut déléguer ses requêtes à un voisin *libre*. Ce chapitre détaille les différentes expériences menées afin de valider cette approche. Avant la conclusion est présenté l'état de l'art, il aborde la théorie de l'ordonnancement dans les systèmes répartis [3, 16, 36, 49], ainsi que des solutions appartenant à la même catégorie que la solution proposée [13, 35, 39].

Le chapitre 5 (p. 77) clôt ce manuscrit. Il résume les contributions de cette thèse et propose différentes perspectives liées à la collaboration de consommateurs de données dans une fédération.

Les annexes A (p. 81) présentent des résultats d'expérimentations complémentaires pour Ladda.

1.5 Publications

Ces travaux ont menés aux publications suivantes :

1. **CyCLaDEs: A Decentralized Cache for Linked Data Fragments**, Pauline Folz, Hala Skaf-Molli, and Pascal Molli. Research paper – Honorable mention research track. Proceedings of the 13th Extended Semantic Web conference (ESWC'16), 2016.

   Abstract: The Linked Data Fragment (LDF) approach promotes a new trade-off between performance and data availability for querying Linked Data. If data providers' HTTP caches plays a crucial role in LDF performances, LDF clients are also caching data during SPARQL query processing. Unfortunately, as these clients do not collaborate, they cannot take advantage of this large decentralized cache hosted by clients. In this paper, we propose CyCLaDEs an overlay network based on LDF fragments similarity. For each LDF client, CyCLaDEs builds a neighborhood of LDF clients hosting related fragments in their cache. During query processing, neighborhood cache is checked before requesting LDF server. Experimental results show that CyCLaDEs is able to handle a significant amount of LDF query processing and provide a more specialized cache on client-side.

2. **Ladda: Query Delegation in a Federation of Linked Data Consumers**. Pauline Folz, Gabriela Montoya, Hala Skaf-Molli, Pascal Molli, Ruben Verborgh, and Miel Vander Sande. Technical report, 2016.

Abstract: The Triple Pattern Fragments interface (TPF) proposes a web-scale approach for publishing and consuming Linked Data. TPF partially transfers SPARQL query processing from servers to clients, increasing significantly the number of resources able to process SPARQL queries. However, the potential of these resources remains under-exploited, because TPF clients do not share their processing capabilities. In this paper, we propose Ladda, a framework that builds a federation of TPF clients, in which any client is connected to a fixed number of neighbors. A TPF client parallelizes on-the-fly the execution of incoming queries by delegating some of these queries to neighbors. Ladda defines a dynamic load-balancer that enables inter-query parallelism on data consumers side. We experimentally evaluated Ladda with real logs of DBpedia. Results suggest that delegation significantly reduces the overall makespan and improves the throughput of the federation.

J'ai également contribué aux travaux suivants, qui ne sont pas présentés dans cette thèse :

1. **Ladda: SPARQL Queries in the Fog of Browsers**. Arnaud Grall, Pauline Folz, Gabriela Montoya, Hala Skaf-Molli, Pascal Molli, Miel Vander Sande, and Ruben Verborgh. Demo paper – Proceedings of the 14th Extended Semantic Web Conference (ESWC'17), 2017.

   Abstract: Clients of Triple Pattern Fragments (TPF) interfaces demonstrate how a SPARQL query engine can run within a browser and re-balance the load from the server to the clients. Imagine connecting these browsers using a browser-to-browser connection, sharing bandwidth and CPU. This builds a fog of browsers where end-user devices collaborate to process SPARQL queries over TPF servers. In this demo, we present Ladda: a framework for query execution in a fog of browsers. Thanks to client-side inter-query parallelism, Ladda reduces the makespan of the workload and improves the overall throughput of the system.

2. **Parallel Data Loading during Querying Deep Web and Linked Open Data with SPARQL**. Pauline Folz, Gabriela Montoya, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. Workshop paper – Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2015) at ISWC, 2015.

   Abstract: Web integration systems are able to provide transparent and uniform access to heterogeneous Web data sources by integrating views of Linked Data, Web Service results, or data extracted from the Deep Web. However, given the potential large number of views, query engines of Web integration systems have to implement execution techniques able to scale up to real-world scenarios and efficiently execute queries. We tackle the problem of SPARQL query processing against RDF views, and propose a non-blocking query execution strategy that incrementally accesses and merges the views relevant to a SPARQL query in a parallel fashion. The proposed strategy is implemented on top of Jena 2.7.4, and empirically compared with SemLAV, a sequential SPARQL query engine on RDF views. Results suggest that our approach outperforms SemLAV in terms of the number of answers produced per unit of time.

3. **SemLAV : Interroger le Web profond et le Web des données avec SPARQL**. Pauline Folz, Gabriela Montoya, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. Demo paper – 30ème conférence Base de Données Avancées (BDA'14), 2014.

Abstract : SemLAV permet d'exécuter des requêtes SPARQL à travers des sources provenant du Web profond et du Web des données. SemLAV implémente l'architecture médiateur basée sur la définition des vues par rapport aux sources de données distantes. Les requêtes SPARQL sont exprimées en utilisant le vocabulaire des médiateurs et SemLAV sélectionne les sources pertinentes et les classent. La stratégie de classement est conçue pour délivrer des résultats rapidement en se basant seulement sur la définition des vues, c-à-d, ni statistiques, ni sondage sur les sources ne sont nécessaires. Dans cette démonstration, nous montrons l'efficacité de SemLAV avec de vraies données issues de réseaux sociaux et du Web des données. En effet, la matérialisation d'un sous ensemble des vues préalablement sélectionnées et classées est suffisant pour produire une partie significative des résultats attendus.

4. **SemLAV: Querying Deep Web and Linked Open Data with SPARQL** Pauline Folz, Gabriela Montoya, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. Demo paper – Proceedings of the 11th Extended Semantic Web Conference (ESWC'14), 2014.

Abstract: SemLAV allows to execute SPARQL queries against the Deep Web and Linked Open Data data sources. It implements the mediator-wrapper architecture based on view definitions over remote data sources. SPARQL queries are expressed using a mediator schema vocabulary, and SemLAV selects relevant data sources and rank them. The ranking strategy is designed to deliver results quickly based only on view definitions, *i.e.*, no statistics, nor probing on sources are required. In this demonstration, we validate the effectiveness of SemLAV approach with real data sources from social networks and Linked Open Data. We show in different setups that materializing only a subset of ranked relevant views is enough to deliver significant part of expected results.

2

État de l'art

Sommaire

Le language SPARQL permet d'évaluer des requêtes sur des données RDF. Évaluer une requête SPARQL sur le Web des données pose plusieurs problèmes complexes :

Expressivité des requêtes Premièrement, faut-il utiliser toute l'expressivité du langage SPARQL ou se restreindre à certains fragments de SPARQL, plus faciles à évaluer ?

Localisation des données Pour évaluer une requête SPARQL sur le Web des données, un premier problème est de définir un périmètre sur lequel exécuter les requêtes. Le périmètre est l'ensemble des sites Web à considérer pour évaluer une requête. Faut-il prendre le Web des données dans son ensemble ? Ou juste considérer un sous-ensemble des données de confiance pour un domaine donné, par exemple : données bio-médicales, données gouvernementales, etc.

Performances et passage à l'échelle SPARQL est un langage de requêtes très expressif. L'évaluation d'une requête SPARQL est PSPACE-complete [46] ; ce qui requiert beaucoup de temps de calcul et de mémoire. Dans le Web des données, le volume des données peut être très important, atteindre des milliards de triplets et le nombre de

requêtes par seconde peut être très élevé. De plus, les données sont naturellement distribuées et partitionnées par site Web. Ce partitionnement peut être très défavorable selon les requêtes. Selon les infrastructures considérées pour évaluer les requêtes, il existe un compromis entre performances et passage à l'échelle/disponibilité des données. Ce compromis existe également sur le coût financier de l'évaluation des requêtes : l'évaluation est-elle exclusivement à la charge des producteurs de données ? Ou des consommateurs de données ? Et/ou d'éventuels médiateurs entre producteurs de données et consommateurs de données ?

Hétérogénéité sémantique Si le Web des données est homogène dans les formats utilisés pour représenter les données, il est hétérogène d'un point de vue sémantique. En effet, le Web des données ne définit pas un schéma strict permettant d'écrire une requête. Rien ne dit qu'une personne sera décrite en RDF de la même manière sur un site et sur un autre. Dans ce contexte, comment faire pour écrire une requête ?

Confiance Selon le périmètre des données, un utilisateur peut voir sa requête exécutée sur des sources envers lesquelles il n'a pas confiance. Ce problème se retrouve également lorsque les données sont répliquées et mises à disposition par un autre producteur que celui qui a autorité sur les données.

Usage Si le producteur de données peut observer que ses données sont accédées pendant l'exécution d'une requête, alors il peut évaluer l'usage de ses données. Si ses données sont recopiées par des intermédiaires, alors il ne peut plus observer l'utilisation de ses données.

Fraîcheur et cohérence des données Dans le cadre de la réplication de données, celles-ci peuvent ne plus être à jour ou cohérentes avec le producteur de données qui a l'autorité dessus.

Il existe plusieurs approches pour évaluer des requêtes SPARQL sur le Web des données : serveurs SPARQL, indexation centralisée du Web des données, traversée le Web des données, fédération de données ou encore les réseaux pair-à-pair. Nous détaillons maintenant chaque approche, avant de positionner notre approche.

2.1 Index centralisé

À l'image du Web classique, il est possible de parcourir (*crawler*) le Web de données et de construire un index permettant de retrouver des ressources selon des mots clés. De nombreux sites proposent[1] de tels indexes comme Sindice [60], Swoogle [2], Watson [19] ou Laundromat [7].

Ces sites ne permettent pas d'exécuter des requêtes SPARQL, mais peuvent être utilisés pour découvrir les resources pertinentes pour une requête ou être directement utilisés par une application utilisant le Web des données comme Sig.ma [59].

Expressivité des requêtes Dans ces systèmes les requêtes ne sont pas exprimées en SPARQL, mais sont des recherches de mots clés.

[1] Ou ont proposé.
[2] http ://swoogle.umbc.edu/2006/

Localisation des données Les données sont chez les producteurs de données, mais l'index sur ces données est centralisé et consultable via des sites Web. Son périmètre est défini par les sources qui sont indexées.

Performances et passage à l'échelle Une fois l'index établi la recherche sur celui-ci est rapide. Cependant, ces approches engendrent des coûts importants en terme de passage à l'échelle, du point de vue du stockage et des ressources nécessaires pour générer les indexes.

Hétérogénéité sémantique Comme il s'agit de recherche de mots clés, les réponses renvoyées sont des correspondances exactes. C'est l'utilisateur qui sait que deux prédicats sont des synonymes.

Confiance et usage Les indexes sont générés par un intermédiaire, mais l'utilisateur peut ensuite sélectionner les sources qui lui semblent de confiance pour exécuter sa requête finale. Les producteurs de données n'ont pas connaissance de cette première recherche pour la découverte de sources de données.

Fraîcheur des données et cohérence des données Les données ne sont pas forcément à jour. À chaque fois qu'une source de données est modifiée il faut recréer l'index pour que celui-ci soit à jour.

2.2 Parcours des liens

SQUIN [32, 33] propose une approche par atteignabilité pour exécuter une requête SPARQL. À partir d'une adresse de départ et d'une requête, on traverse le graphe RDF en construisant les résultats de la requête. Par exemple, pour évaluer la requête SPARQL présentée dans le listing 2.1 [3] ; SQUIN commence avec un répertoire local vide. Dans un premier temps, il récupère les données correspondant aux deux URIs : `producer1` et `vendor1`, puis les rajoute au répertoire local. S'ensuit un processus itératif qui est basé sur les triplets récupérés. Pour chaque triplet correspondant à un triplet de la requête, celui-ci est utilisé pour construire une solution intermédiaire ou pour déréférencer d'autres URIs. À la fin du processus les solutions intermédiaires qui correspondent à tous les triplets de la requête sont les résultats de la requête.

```
SELECT *
WHERE {
  ?p producedBy producer1 .
  ?p name ?pn .
  ?o offeredProduct ?p .
  ?o offeredBy vendor1 .
}
```

Listing 2.1 – Requête SPARQL fictive.

Cette approche est très élégante. Elle garantit la fraîcheur des données et passe à l'échelle dans le sens où le coût d'exécution de la requête est à la charge de celui qui exécute la requête. Cependant, les temps d'exécution des requêtes peuvent devenir rapidement très longs.

Expressivité des requêtes Les requêtes de «parcours des liens» permettent d'utiliser toute l'expressivité du langage SPARQL.

[3] Extraite de [32].

Localisation des données Les requêtes sont évaluées sur l'ensemble des données atteignables en fonction de l'URL de départ. Les résultats peuvent ne pas être complets.

Performances et passage à l'échelle L'exécution d'une requête est peu performante. En effet, les résultats sont produits incrémentalement en parcourant les graphes RDF. Par contre, les données restent très disponibles car l'accès aux données ne nécessite qu'un serveur Web standard.

Hétérogénéité sémantique L'hétérogénéité sémantique est à la charge de l'utilisateur lorsqu'il écrit sa requête.

Confiance et usage Les requêtes sont exécutées sur l'intégralité du Web des données, même sur les sources de données envers lesquelles l'utilisateur n'a pas confiance. Les sites Web fournissant les données peuvent observer quelles données sont accédées et par qui.

Fraîcheur des données et cohérence des données Les données sont à jour et cohérentes car elles sont directement récupérées chez les producteurs de données.

2.3 Serveurs SPARQL avec autorité

Certains sites hébergent un serveur permettant d'évaluer des requêtes SPARQL sur les données locales à ce site, c'est-à-dire les données sur lesquelles le producteur de données a l'autorité. Par exemple, DBpedia permet d'interroger sa base de données RDF à l'adresse suivante : `http://dbpedia.org/sparql`.

Expressivité des requêtes Les sites comme celui de DBpedia permettent d'exécuter des requêtes SPARQL, mais certaines limites sont imposées afin de pas dégrader les performances du serveur SPARQL, comme le nombre de triplets retournés par requête.

Localisation des données Le périmètre des données RDF est restreint aux données RDF gérées par les hébergeurs en question.

Performances et passage à l'échelle Si le serveur n'est pas chargé, l'exécution d'une requête SPARQL est efficace. Cependant, dans le cas général, le volume de requêtes SPARQL à traiter à un moment donné peut poser des problèmes de performances et de passage à l'échelle. Selon une étude de 2013 [4], la majorité des serveurs SPARQL ont une disponibilité inférieure à 95%. Pour pallier ce problème, les producteurs de données mettent en place des quotas très restrictifs pour garantir une qualité de service minimum. Par exemple, DBpedia limite [4] le temps d'execution des requêtes à 120 secondes et 10 000 résultats par requête.

Hétérogénéité sémantique Les problèmes d'hétérogénéité sémantique peuvent être maîtrisés au niveau local, un seul producteur maîtrise les données à publier.

Confiance et usage Le producteur de données publie sur son serveur des données sur lesquelles il a l'autorité. Les requêtes étant exécutées directement sur son serveur SPARQL, le producteur de données connaît les requêtes qui sont exécutées sur ses données.

Fraîcheur des données et cohérence des données On peut considérer que les données sont à jour et cohérentes lorsqu'elles sont hébergées par le même producteur de données. En effet, une mise à jour sur les données Web peut être directement ré-appliquée sur le serveur SPARQL.

[4] https ://medium.com/openlink-software-blog/dbpedia-usage-report-a78b3802a1d3

2.4 Serveurs SPARQL sans autorité

Certains sites publient leurs données sous forme d'archives. Par exemple, DBpedia permet de récupérer l'ensemble des données RDF sous forme d'archive [5]. Une manière simple d'interroger le Web des données consiste à insérer sur un même serveur plusieurs archives. Ce serveur peut être local ou public comme le portail LODCache [6].

Expressivité des requêtes Comme pour les serveurs SPARQL avec autorité, ce type de serveur propose généralement une interface Web pour exécuter les requêtes autorisant les utilisateurs à utiliser toute l'expressivité de SPARQL. Cependant, certaines limites sont mises en place pour garantir une qualité de service à l'utilisateur.

Localisation des données Le périmètre des requêtes SPARQL est donc l'ensemble des archives disponibles dans le serveur SPARQL.

Performances et passage à l'échelle Le système peut être performant mais ne pas passer à l'échelle avec un grand nombre de requêtes concurrentes.

Hétérogénéité sémantique L'hétérogénéité sémantique est présente mais non gérée.

Confiance et usage Dans ce cas le serveur renvoie des triplets sur lesquels il n'a pas d'autorité. Le serveur d'origine, s'il existe, ne peut pas observer les requêtes des utilisateurs.

Fraîcheur des données et cohérence des données Les données RDF sont recopiées avec les problèmes intrinsèques de cohérence et de fraîcheur des données.

2.5 Requêtes sur réseaux pair-à-pair

Les réseaux pair-à-pair peuvent facilement se superposer à un réseau existant et fournir des services supplémentaires au réseau d'origine. L'idée principale est de superposer au Web des données un réseau pair-à-pair capable de fournir les services d'un gestionnaire de données. Il s'agit d'adresser les problèmes suivants [44] :

Localisation des données : les pairs doivent être capables de référencer et de localiser les données stockées chez les autres pairs.

Traitement de requêtes : pour une requête donnée, le système doit être capable de découvrir les pairs qui ont des données pertinentes pour la requête et qui pourraient exécuter cette dernière.

Intégration des données : quand des sources de données partagées ont des schémas de données différents, les pairs doivent être capables d'accéder à n'importe quelle source, idéalement en utilisant la même représentation des données que celle du pair qui consulte ces données.

Cohérence des données : si les données sont répliquées ou mises en cache dans le système, il est important de maintenir la cohérence des données entre la version originale et les répliques.

[5] http://wiki.dbpedia.org/Datasets
[6] http://datahub.io/dataset/openlink-lod-cache

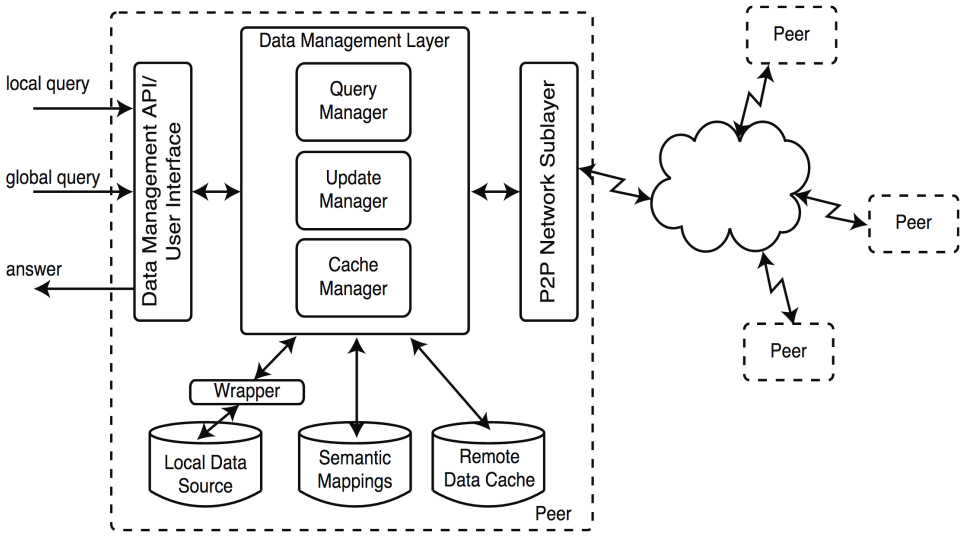


FIGURE 2.1 – Architecture d'un pair - Source [44].

La figure 2.1 présente la structure générale d'un nœud dans un gestionnaire de données sur un réseau pair-à-pair. Chaque nœud est capable de recevoir une requête en entrée soit en provenance d'un utilisateur, soit en provenance d'un autre nœud. L'hétérogénéité syntaxique et sémantique est gérée à travers des *wrappers* et des définitions d'alignements. Un gestionnaire de données sur réseau pair-à-pair requiert différentes propriétés [44] :

Autonomie : les pairs du réseaux sont autonomes et peuvent rejoindre et quitter le système quand ils le veulent et sans restrictions. Ils doivent pouvoir contrôler les données qu'ils stockent et quel(s) autre(s) pair(s) peut stocker leurs données.

Expressivité des requêtes : le language de requête doit permettre à l'utilisateur de décrire les données au degré de détail approprié.

Efficacité : l'utilisation intelligente des ressources du réseau pair-à-pair doit mener à un meilleur débit de requêtes et un coût de fonctionnement moindre.

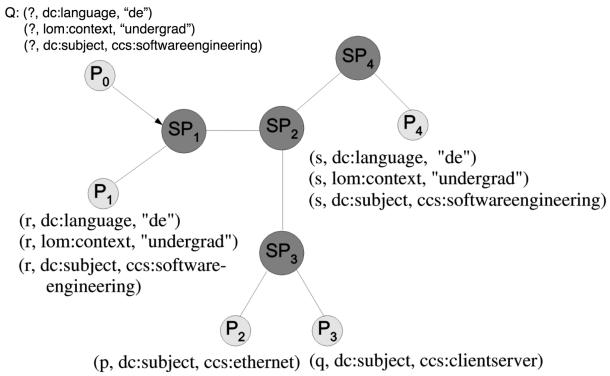
Qualité de service : concerne la perception de l'utilisateur à propos de l'efficacité du système, comme la complétude des résultats aux requêtes, la cohérence des données, le temps de résultats des requêtes, etc.

Tolérances aux pannes : l'efficacité et la qualité du service doivent être maintenues même s'il y a des pairs défaillants.

Sécurité : Les réseaux pair-à-pair étant ouverts, ils soulèvent des problèmes liés à la sécurité, au niveau du contrôle d'accès et de la confiance que s'accordent les pairs du réseau entre eux.

Un gestionnaire de données sur réseau pair-à-pair demande une collaboration bien supérieure à celle demandée pour participer au Web des données. En effet, un producteur de données du Web des données doit juste suivre les principes des données liées. Un participant dans un gestionnaire de données sur réseau pair-à-pair partage des ressources pour le stockage des données et l'exécution des requêtes. Il est tout à fait possible pour un producteur de données de déclarer ses données sur un gestionnaire de données pair-à-pair et de ne pas stocker lui-même ses données.

Les sections suivantes présentent quelques systèmes représentatifs des gestionnaires de données RDF sur réseaux pair-à-pair. Nous nous focalisons sur l'exécution des requêtes dans ces systèmes.



(a) Propagation d'une requête sur les super-pairs.

Granularity	**Index of SP_2**		
Schema	dc		SP_1, SP_3, SP_4
	lom		SP_1, SP_4
Property	dc:subject		SP_1, SP_3, SP_4
	dc:language		SP_1, SP_4
	lom:context		SP_1, SP_4
Property	dc:subject	ccs:networks	SP_3
Value Range	dc:subject	ccs:software-eng.	SP_1, SP_4
Property	lom:context	“undergrad”	SP_1, SP_4
Value	dc:language	“de”	SP_1, SP_4

(b) Indice SP/SP de SP_2 avec différentes granularités.

FIGURE 2.2 – Exécution d'une requête avec Edutella - Source [43].

2.5.1 Edutella

Dans sa première version, Edutella [41] est un gestionnaire de données RDF sur un réseau pair-à-pair non structuré basé sur JXTA [26]. Chaque pair expose les requêtes auxquelles il peut répondre : requêtes conjonctives et requêtes d'intervalles. Quand un pair soumet une requête, cette requête est propagée à travers le réseau (*flooding*) et les pairs concernés retournent les résultats à l'émetteur. L'exécution d'une requête génère beaucoup de trafic dans le réseau pair-à-pair. Pour pallier à ce problème, Edutella a évolué vers un réseau pair-à-pair avec des super-pairs [43]. Un pair est connecté à un seul super-pair. Les super-pairs sont connectés entre eux avec une topologie hypercube basée sur le protocole HyperCuP [50]. La figure 2.2a montre un réseau Edutella composé de quatre super-pairs. Les super-pairs ont deux types d'index :

1. Un index des contenus des pairs (SP/P), similaire à la première version d'Edutella.
2. Un index des contenus des super-pairs (SP/SP).

La figure 2.2b montre l'index SP/SP de SP_2. Les indexes sont utilisés pour propager les requêtes dans le réseau. Dans la figure 2.2b P_0 reçoit une requête : «Trouver les cours de génie logiciel en allemand pour les étudiants de master». Dans cet exemple, P_1 et P_4 peuvent répondre à la requête. P_0 envoie la requête à son super-pair SP_1. SP_1 envoie la requête à P_1 et SP_2. SP_2 envoie la requête à SP_4 et SP_4 envoie la requête à P_4. Dans ce scénario, les indexes ont permis de ne pas renvoyer la requête à SP_3.

Expressivité des requêtes Dans Edutella les requêtes sont exprimées avec le langage RDF-QEL qui est basé sur Datalog.

Localisation des données Le réseau des super-pairs possède un résumé du contenu de chaque pair dont il a la charge. Les supers-pairs forment un index décentralisé sur l'ensemble des pairs du réseau. Le périmètre des données est donc l'ensemble des pairs participant au réseau Edutella.

Performances et passage à l'échelle L'exécution de requêtes SPARQL peut être coûteuse, car la requête parcourt tous les super-pairs du réseau afin de trouver des pairs pertinents pour évaluer la requête. Le passage à l'échelle est difficile dans le cadre de nombreuses requêtes concurrentes.

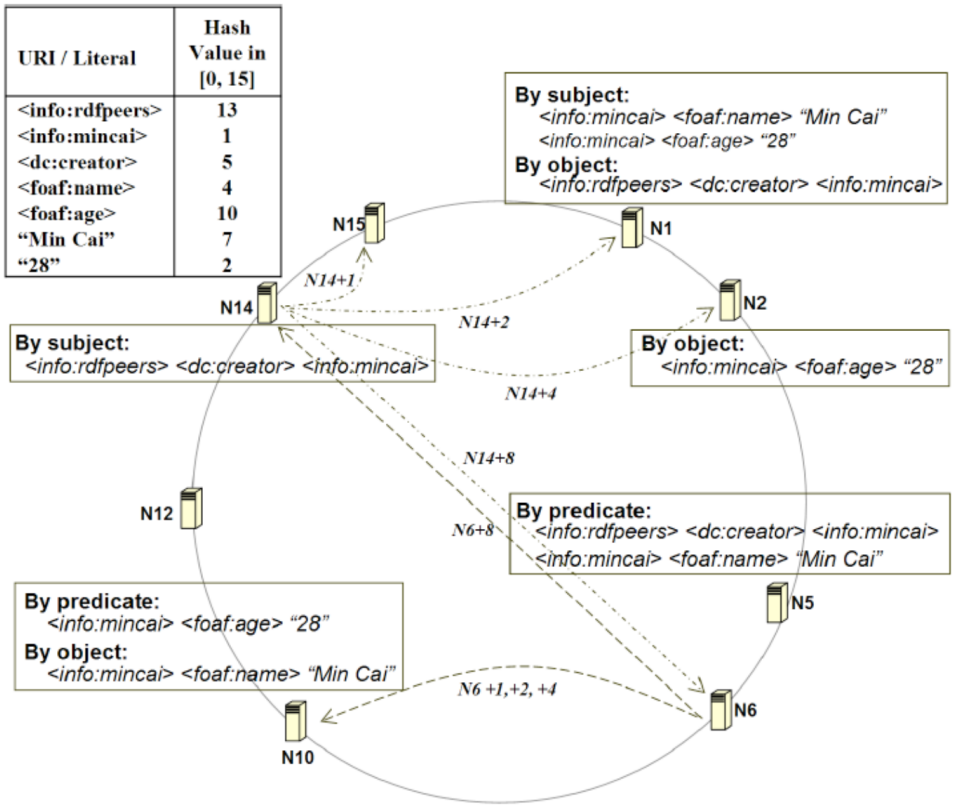


FIGURE 2.3 – Stockage des triples dans RDFPeers - Source [14].

Hétérogénéité sémantique Edutella [43] fournit un service d'alignement de vocabulaires afin de gérer l'hétérogénéité sémantique des pairs.

Confiance et usage Les requêtes sont directement exécutées sur les pairs hébergeant les données sur lesquelles ils ont autorité. Ils sont donc à même de connaître les requêtes.

Fraîcheur des données et cohérence des données Les données sont fraîches et cohérentes car les requêtes sont exécutées directement chez les pairs responsables de ces données.

2.5.2 RDFPeers

RDFPeers [14] est un gestionnaire de données RDF sur un réseau pair-à-pair structuré basé sur MAAN [15] (*Multi-attribute Addressable Network* (MAAN)), lui-même basé sur Chord [56] qui une table de hachage distribuée. Le principe est d'attribuer des identificateurs uniques et strictement ordonnés aux pairs et aux données au sein d'un même espace d'adressage. Un pair est alors responsable des identifiants situés entre son identifiant et celui du premier pair suivant dans l'ordre des identifiants. MAAN étend Chord en utilisant comme fonction de hachage une fonction capable de préserver l'ordre. Cela lui permet d'exécuter des requêtes d'intervalles.

RDFPeers stocke des triplets. Chaque triplet est stocké trois fois, en utilisant une fonction de hachage sur le sujet, le prédicat et l'objet. La figure 2.3 montre comment les trois triplets suivants sont stockés dans RDFPeers :

```
<info:rdfpeers> <dc:creator> <info:mincai> .
<info:mincai> <foaf:name> "Min Cai" .
<info:mincai> <foaf:age> "28"^^<xmls:integer>
```

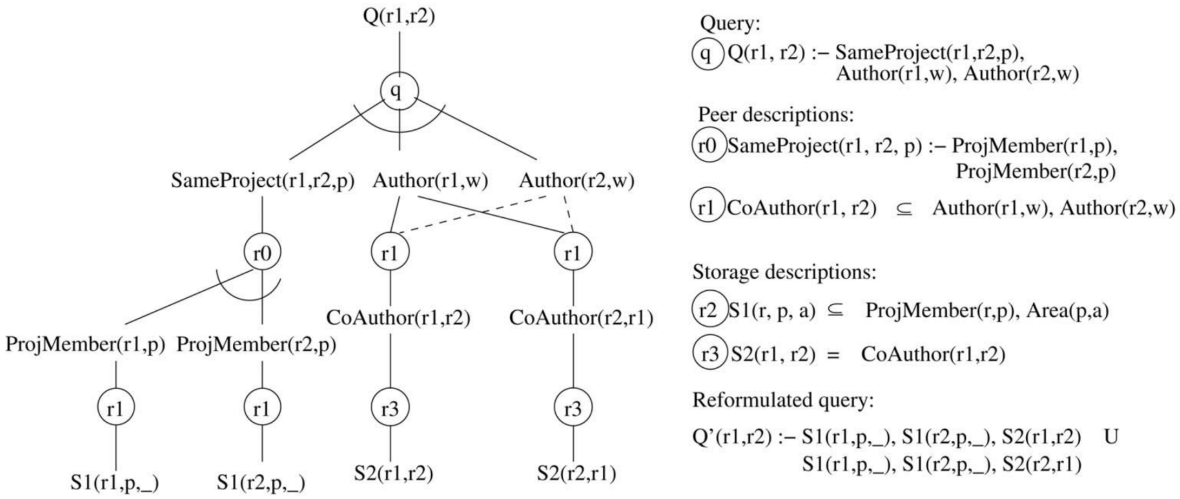


FIGURE 2.4 – Exécution d'une requête avec Piazza - Source [29].

MAAN permet de trouver n'importe quel patron de triplet en $log(N)$ sauts sur le réseau où N est le nombre de pairs. Sauf le patron de triplet `( ?s, ?p, ?o)` qui est en $O(N)$. Les requêtes conjonctives sont évaluées en $O(k(logN) + N * s_i)$, où k est le nombre de sous-requêtes et s_i la sélectivité de la sous-requête pour le prédicat p_i. Les résultats sont produits de manière incrémentale.

Expressivité des requêtes RDFPeers supporte les requêtes : atomiques, disjonctives, d'intervalles et conjonctives.

Localisation des données Dans RDFPeers, le périmètre est l'ensemble des données insérées dans RDFPeers par les participants.

Performances et passage à l'échelle La complexité d'évaluation des requêtes dégrade sérieusement les performances si les requêtes génèrent beaucoup de résultats intermédiaires. RDFPeers passe à l'échelle sur le stockage des données. Cependant, rien ne garantit que les accès soient uniformément répartis sur les données.

Hétérogénéité sémantique L'hétérogénéité sémantique n'est pas gérée dans RDFPeers.

Confiance et usage Les participants choisissent les données qu'ils veulent stocker. Comme les requêtes sont exécutées sur les nœuds RDFPeers, les producteurs de données n'ont plus connaissance des requêtes effectuées sur leurs données.

Fraîcheur des données et cohérence des données Les données recopiées dans RDFPeers sont à jour si le producteur de données met à jour RDFPeers en même temps que les données originales.

2.5.3 Piazza

Piazza [30] est un gestionnaire de données pair-à-pair qui permet de gérer l'hétérogénéité sémantique des sources de données dans le Web sémantique. Chaque pair du réseau propose des données avec son schéma ou seulement une ontologie. Piazza permet d'effectuer des requêtes sur des sources hétérogènes grâce à des alignements (*mappings*) entre pairs, c'est-à-dire que le pair A fait un alignement de son vocabulaire avec le pair B. Cependant, la réciproque n'est pas automatique.

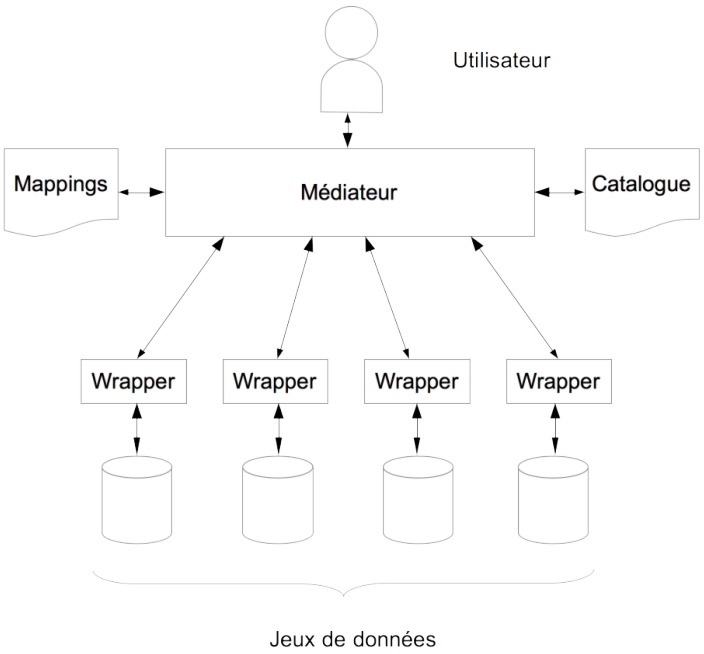


FIGURE 2.5 – Architecture d'un médiateur.

Un utilisateur exprime une requête en fonction du schéma d'un des nœuds du réseau. La figure 2.4 illustre l'exécution d'une requête avec les *mappings* de Piazza. La requête est réécrite en fonction des schémas des pairs pertinents avant d'être évaluée.

Dans un premier temps la requête Q est réécrite (Q') en fonction des données stockées sur le pair P. Puis, l'algorithme considère les voisins de P qui sont liés sémantiquement, c'est-à-dire qui sont reliés à lui par des *mappings*. La requête Q est alors étendue en Q" au schéma des voisins de P. Enfin l'union entre Q' et Q" permet d'éliminer les redondances entre les deux réécritures de requêtes. Ces étapes sont répétées en suivant les *mappings* entre les schémas des nœuds, jusqu'à ce qu'il ne reste plus de chemin utile pour résoudre la requête.

Expressivité des requêtes Piazza supporte le langage SPARQL dans son ensemble.

Localisation des données Le périmètre des données est l'ensemble des pairs sur lesquels la requête a été propagée.

Performances et passage à l'échelle L'exécution d'une requête est longue car la requête doit être propagée pour être réécrite, puis évaluée sur plusieurs pairs. De plus, cette approche ne passe pas à l'échelle. Pour l'instant l'approche ne dépasse pas 80 pairs [2].

Hétérogénéité sémantique L'hétérogénéité sémantique est gérée de pair en pair. Un pair A aligne son vocabulaire par rapport au vocabulaire d'un autre pair B.

Confiance et usage Les données restent chez les producteurs de données. Les producteurs de données observent les requêtes effectuées sur leurs données.

Fraîcheur des données et cohérence des données Les données sont fraîches et cohérentes.

2.6 Approche médiateurs et fédération de données

Un médiateur permet de poser des requêtes sur un ensemble de sources de données prédéfinies. Le schéma de données du médiateur et des sources de données peuvent être différents, ainsi que le format des données.

L'architecture générale d'un médiateur est présentée dans la figure 2.5. Le médiateur maintient un index (*Catalogue*) des sources de données qu'il peut interroger. Les *wrappers* permettent de gérer l'hétérogénéité syntaxique des données. Les alignements (*Mappings*) permettent de gérer l'hétérogénéité sémantique des données. De manière générale, pour une requête donnée, un médiateur commence par sélectionner les sources pertinentes pour la requête parmi son catalogue. Il réécrit ensuite la requête en sous-requêtes exécutables sur les sources de données. Les *wrappers* permettent alors l'exécution des sous-requêtes et la récupération des résultats.

Dans une approche médiateur, les données restent chez les producteurs de données. Il n'y a donc pas de problème de fraîcheur des données. Les sources de données n'ont pas besoin de coopérer. Du point de vue des performances, un médiateur n'a pas accès aux statistiques des données et ne peut donc pas optimiser la requête comme peut le faire un serveur SPARQL. La disponibilité des données est limitée aux capacités de traitement du médiateur.

2.6.1 Médiateurs et bases de données fédérées

Dans le cadre du Web des données, l'utilisation de RDF et de serveurs standardisés permettent de réduire considérablement l'hétérogénéité syntaxique. L'écriture des *wrappers* devient alors assez triviale. Ainsi, le médiateur est assez proche des systèmes de bases de données fédérées comme Garlic [28] ou Disco [58].

Si les données ont bien été publiées en suivant les principes des données liées alors l'utilisation de vocabulaires communs et des relations de type `sameAs` réduit suffisamment l'hétérogénéité sémantique pour permettre l'écriture de requêtes SPARQL [27] sans repasser par des alignements. Par exemple, la requête CDQ5 [52] ci-dessous, peut être exécutée par un médiateur configuré avec deux sources de données : DBpedia et LinkedMDB, sans déclaration d'alignements (*mappings*).

```
SELECT ?film ?director ?genre
WHERE {
  ?film dbpedia-owl:director ?director.
  ?director dbpedia-owl:nationality dbpedia:Italy .
  ?x owl:sameAs ?film .
  ?x linkedMDB:genre ?genre .
}
```

Listing 2.2 – Requête SPARQL : Films de réalisateurs italiens.

La requête CDQ5 retourne les films de réalisateurs italiens sur DBpedia et LinkedMDB. La relation `sameAs` permet de faire le lien entre DBpedia et LinkedMDB. L'écriture de cette requête nécessite une connaissance préalable des données de DBpedia et LinkedMDB. Cette connaissance est obtenue en parcourant le Web des données.

En éliminant les problèmes d'hétérogénéité sémantique et syntaxique, le médiateur devient beaucoup plus léger et peut facilement être déployé sur les machines des utilisateurs finaux, voir directement dans les navigateurs Web.

2.6.2 Moteurs de requêtes fédérées et serveurs SPARQL

Les moteurs de requêtes fédérées [1, 53] sont des médiateurs permettant d'exécuter des requêtes SPARQL sur plusieurs serveurs SPARQL. Par exemple, DBpediaFr et Geonames fournissent tous les deux un serveur SPARQL contenant les faits RDF sur lesquels ils ont l'autorité.

La requête présentée dans le listing 2.3 retourne les œuvres de de Sandro Botticelli ainsi que les coordonnées GPS des villes où elles sont exposées.

```
PREFIX dcterms: <http://purl.org/dc/terms/>

SELECT ?titre ?lat ?long
WHERE {
  ?oeuvre  prop-fr:titre ?titre .
  ?oeuvre  dbpedia-owl:author dbpedia-fr:Sandro_Botticelli .
  ?oeuvre  dbpedia-owl:city ?ville .
  ?ville owl:sameAs ?villeBis .
  ?villeBis wgs84_pos:lat .
  ?villeBis wgs84_pos:long .
}
```

Listing 2.3 – Requête SPARQL SELECT : Coordonnées géographiques des lieux où sont conservées les œuvres de Sandro Botticelli.

Pour exécuter cette requête, il faut configurer un médiateur avec au moins les URIs des serveurs SPARQL de DBpediaFR et Geonames. Le moteur de requêtes fédérées sélectionne les sources pertinentes pour la requête et réécrit la requête présentée dans le listing 2.3 afin de l'exécuter.

Pour évaluer cette requête, le médiateur demande d'abord à DBpediaFR la sous-requête suivante :

```
SELECT *
WHERE {
  ?oeuvre  prop-fr:titre ?titre .
  ?oeuvre  dbpedia-owl:author dbpedia-fr:Sandro_Botticelli .
  ?oeuvre  dbpedia-owl:city ?ville .
  ?ville owl:sameAs ?villeBis .
}
```

Listing 2.4 – Sous-requête SPARQL SELECT : Coordonnées géographiques des lieux où sont conservées les œuvres de Sandro Botticelli.

Ensuite, à chaque résultat intermédiaire, le médiateur fait la jointure avec Geonames. Cette stratégie est efficace si le nombre de résultats intermédiaires est faible. Pour cette requête, certaines jointures sont faites par les serveurs SPARQL et la jointure principale par le médiateur lui-même.

L'utilisateur peut également spécifier sur quelle source de données chaque partie de la requête doit être exécutée grâce au mot clé `SERVICE`. La requête présentée dans le listing 2.5 est une réécriture de la requête présentée dans le listing 2.3 en utilisant le mot clé `SERVICE`.

```
PREFIX dcterms: <http://purl.org/dc/terms/>

SELECT ?titre ?lat ?long
FROM <http://fr.dbpedia.org/sparql>
WHERE {
  ?oeuvre  prop-fr:titre ?titre .
```

```
    ?oeuvre  dbpedia-owl:author dbpedia-fr:Sandro_Botticelli .
    ?oeuvre  dbpedia-owl:city ?ville .
    ?ville owl:sameAs ?villeBis .
    SERVICE <http://www.geonames.org/> {
      ?villeBis wgs84_pos:lat ?lat .
      ?villeBis wgs84_pos:long ?long .
    }
  }
```

Listing 2.5 – Requête SPARQL SELECT : Coordonnées géographiques des lieux où sont conservées les œuvres de Sandro Botticelli avec clause SERVICE.

Expressivité des requêtes Grâce au moteur de requêtes fédérées, l'utilisateur a accès à toute l'expressivité de SPARQL, même si en interne le moteur de requêtes réécrit celles-ci.

Localisation des données Les sources de données sont limitées au catalogue. Pour une requête donnée, c'est un sous-ensemble du catalogue. Les requêtes sont donc incomplètes si l'on considère le Web des données dans son ensemble.

Performances et passage à l'échelle Les performances du moteur de requêtes fédérées sont limités par les performances des serveurs SPARQL. De plus, comme le médiateur n'a pas accès aux statistiques des serveurs SPARQL, l'optimisation des requêtes fédérées est limitée. La disponibilité des données dépend directement de la disponibilité des serveurs SPARQL. Or selon [4], la disponibilité des serveurs SPARQL est inférieure à 95%.

Hétérogénéité sémantique Anapsid et Fedx ne gèrent pas l'hétérogénéité sémantique.

Confiance et usage Dans l'approche médiateur, l'utilisateur peut restreindre le médiateur à un ensemble de sources envers lesquelles il a confiance. Les producteurs de données connaissent l'usage de leurs données en observant les sous-requêtes qu'ils exécutent.

Fraîcheur des données et cohérence des données Si le médiateur est configuré avec des serveurs SPARQL hébergeant des données sur lesquelles ils ont autorité, alors on peut considérer que les données sont cohérentes et fraîches.

2.6.3 Triple Pattern Fragment (TPF)

L'approche Triple Pattern Fragment [63] (TPF) permet d'exécuter des requêtes SPARQL sur un ou plusieurs serveurs TPF. Comparé à un serveur SPARQL, un serveur TPF ne permet d'exécuter que des patrons de triplets. Un serveur TPF ne peut donc pas exécuter de jointures. Toutes les jointures sont exécutées au sein du médiateur TPF.

Le médiateur TPF décompose une requête SPARQL en de multiples requêtes de type patron de triplets, appelée requête de fragment, de façon à remonter dans le médiateur les données suffisantes pour calculer les résultats de la requête. L'exécution des requêtes SPARQL peut être bien moins performante qu'avec des serveurs SPARQL, par contre la disponibilité des serveurs TPF est bien meilleure.

La figure 2.6 résume les différents compromis pour le partage des coûts entre producteurs et consommateurs de données afin d'exécuter une requête SPARQL. Si le producteur de données fournit juste une archive de ses données, cela signifie que le consommateur de données doit charger ces données dans un serveur SPARQL local pour y exécuter sa requête. L'exécution sera certainement performante mais son coût est principalement à la charge du consommateur de données.

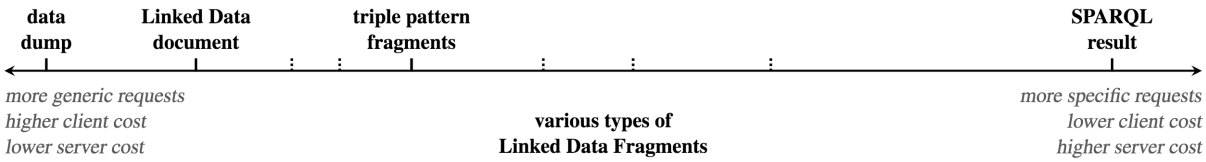


FIGURE 2.6 – Type de fragments de données liées avec leurs spécificités - Source [63].

Si le producteur fournit un serveur SPARQL, alors le consommateur n'a plus qu'à soumettre sa requête et attendre les résultats. Dans ce cas, le coût d'exécution de la requête est presque entièrement à la charge du producteur de données. Les temps d'exécution des requêtes dépendront alors de la charge du serveur à ce moment.

Dans ce cadre, TPF fournit un nouveau compromis. Par rapport à des serveurs SPARQL, TPF offre des performances limitées mais une plus grande disponibilité des serveurs et donc des données.

2.6.3.1 Définitions

Afin d'alléger le coût de mise à disposition des données, le serveur TPF expose des *fragments de triplets* (*triple pattern fragment*) [63] à travers une API REST à la place du protocole SPARQL.

Définition 1 (Fragment de triplet)**.** *Soit G un ensemble fini, sans ressources anonymes (*blank node*), de triplets RDF. Un fragment de triplet de G est un tuple $f = \langle u, s, \Gamma, M, C \rangle$ où [63] :*

- *u est un URI (qui est la source où f peut être récupéré) ;*
- *s est un* sélecteur *;*
- *Γ est un ensemble de triplets RDF, sans ressources anonymes, qui est le résultat de l'application du sélecteur s à G, $\Gamma = s(G)$;*
- *M est un ensemble fini de triplets RDF, incluant les triplets qui représentent les métadonnées de f ;*
- *C est un ensemble fini de contrôles hypermédia.*

Un fragment de triplet est identifié grâce à son *sélecteur* s et sa source u. Afin d'éviter aux consommateurs de données de télécharger de larges paquets de données en une seule fois, les fragments de triplet sont paginés. Chaque page d'un fragment de triplet contient un nombre borné de triplets (ex : 100 triplets par page). Les méta-données M d'un fragment permettent d'estimer le nombre total de triplets qui correspondent à un fragment de triplet, soit le nombre de résultats qui correspondent au sélecteur de fragment et les contrôles hypermédia contiennent les URLs afin de naviguer dans un fragment de triplet (première page, page précédente et page suivante).

Exemple 1. *Un fragment de triplet des architectes sur le jeu de données DBpedia 2016-04 est représenté comme suit : $u =$* `http://fragments.dbpedia.org/2016-04/en`*; $s =$* `?person a dbpedia-owl:Architect`*; Γ = les 100 premiers triplets correspondant aux architectes dans DBpedia 2016-04 ; M = le nombre de triplets correspondant au sélecteur présenté dans le listing 2.6 ; C = la première page et la page suivante du fragment de triplet présenté dans le listing 2.7.*

```
<http://fragments.dbpedia.org/2016-04/en?predicate=http://www.w3.org
    /1999/02/22-rdf-syntax-ns#type&object=http://dbpedia.org/ontology/
    Architect> hydra:totalItems "5823"^^xsd:integer;
```

```
void:triples "5823"^^xsd:integer;
hydra:itemsPerPage "100"^^xsd:integer;
```

Listing 2.6 – Extrait des méta-données du fragment de triplet des architectes.

```
<http://fragments.dbpedia.org/2016-04/en?predicate=http://www.w3.org
    /1999/02/22-rdf-syntax-ns#type&object=http://dbpedia.org/ontology/
    Architect> hydra:first <http://fragments.dbpedia.org/2016-04/en?
    predicate=http://www.w3.org/1999/02/22-rdf-syntax-ns#type&object=http
    ://dbpedia.org/ontology/Architect&page=1>;
hydra:next <http://fragments.dbpedia.org/2016-04/en?predicate=http://www.
    w3.org/1999/02/22-rdf-syntax-ns#type&object=http://dbpedia.org/
    ontology/Architect&page=2>
```

Listing 2.7 – Extrait des contrôles hypermédia du fragment de triplet des architectes.

2.6.3.2 Exécution d'une requête avec le client TPF

Plusieurs jeux de données sont déjà exposés sur des serveurs TPF [7]. Pour exécuter une requête SPARQL sur un serveur TPF, le client TPF transforme la requête SPARQL en une séquence de requêtes TPF de patrons de triplets. Pour limiter le nombre d'appels TPF, le client TPF ordonne les jointures les plus sélectives d'abord. Pour calculer la sélectivité, TPF se base sur la cardinalité des patrons de triplets disponibles dans les méta-données d'un fragment de triplet. Par mesure de simplicité supposons une requête avec un seul BGP (*Basic Graph Pattern*) :

1. Pour chaque triplet dans le BGP (B) récupérer la première page du fragment de triplet correspondant et choisir le fragment de triplet f_ϵ qui est le plus sélectif, soit qui a la plus petite cardinalité.

2. Récupérer les autres pages de f_ϵ. Chaque triplet de résultat du fragment f_ϵ est une solution de *mapping* pour le triplet tp_ϵ, qui est le sélecteur du fragment f_ϵ. Pour chaque solution de *mapping*, créer un sous-BGP, B', des triplets restants du BGP original (B), $\forall tp_i \in B | tp_i \neq tp_\epsilon$, et appliquer la solution de *mapping* de tp_ϵ aux triplets de B'.

3. Répéter l'opération précédente de façon récursive, jusqu'à l'obtention des solutions de *mappings* pour tous les triplets.

Exemple 2. *Considérons la requête 2.8 [8], des architectes nés dans une capitale Européenne, le client TPF découpe la requête en triplets : tp_1, tp_2 et tp_3. Puis procède comme suit :*

1. *Récupérer la première page de chaque fragment de triplet pour avoir sa cardinalité :*
   - tp_1 = *5823 ;*
   - tp_2 = *1 137 061 ;*
   - tp_3 = *59.*

   Le triplet tp_3 est le plus sélectif, soit celui avec le plus petit nombre de triplets correspondant au sélecteur, il est donc le fragment de départ.

[7] http://linkeddatafragments.org/data/
[8] Requête exécutée sur : http://fragments.dbpedia.org/2016-04/en.

2. *Pour chaque triplet contenu dans f_{tp_3}, soit les mappings de la variable* `?city` *; créer un sous-BGP, $B' = \{tp_1, tp_2\}$ et appliquer la solution aux triplets restants. La variable* `?city` *dans tp_2 est remplacée par les solutions de mapping de tp_3 :* `?person dbpedia-owl:birthPlace dbpedia:Amsterdam`.

3. *Pour chaque solutions de mappings de la variable* `?person` *de tp_2, les appliquer à la variable* `?person` *de tp_1 :* `dbpedia:Erick_van_Egeraat a dbpedia-owl:Architect`.

```
SELECT ?person ?city
WHERE {
   ?person a dbpedia-owl:Architect.                          # tp1
   ?person dbpedia-owl:birthPlace ?city.                     # tp2
   ?city dc:subject dbpedia:Category:Capitals_in_Europe.     # tp3
}
LIMIT 5
```

Listing 2.8 – Requête SPARQL SELECT : Architectes nés dans une capitale Européenne.

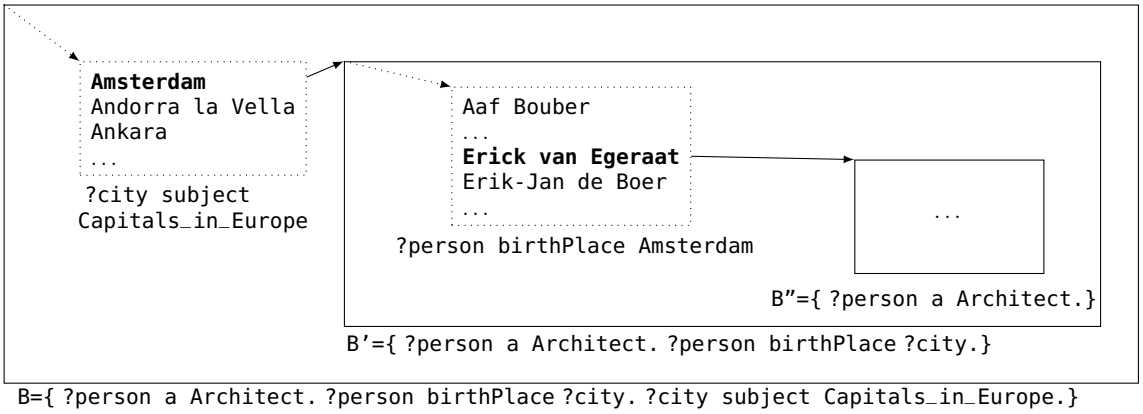


FIGURE 2.7 – Un itérateur décompose la requête en itérateur de triplet, pour chaque résultat de *mapping*, créer un nouvel itérateur pour les triplets restants.

Le client TPF produit des résultats de façon incrémentale grâce à des itérateurs en pipeline [63]. Il existe deux types principaux d'itérateurs : l'itérateur de BGP et l'itérateur de triplets. Les itérateurs de triplets, I_{tp}, sont initialisés avec un itérateur prédécesseur (I_p) et un triplet (tp_i). L'itérateur applique ensuite les *mappings* de I_p au triplet tp_i fourni en entrée. Un itérateur de BGP, (I_{BGP}) est initialisé avec un itérateur prédécesseur (I_p) et un BGP ($B = \{tp_1, \ldots, tp_n\}$). Un itérateur I_{BGP} évalue un BGP de façon récursive en le décomposant en plusieurs itérateurs. Pour chaque triplet dans le BGP, I_{BGP} crée un itérateur pour récupérer la première page de chaque fragment de triplet. Le BGP est alors décomposé avec (i) un itérateur de triplets, pour le triplet avec la cardinalité la plus petite $I_{tp_\epsilon} = (I_p, tp_\epsilon)$, (ii) un itérateur BGP pour les triplets restants $I_{B'} = (I_p, \forall tp_i \in B \wedge \neq tp_\epsilon)$. La figure 2.7 illustre le fonctionnement des itérateurs pour la requête du listing 2.8.

2.6.3.3 Architecture de l'approche TPF

L'approche TPF réduit la pression sur le producteur de données, d'une part parce que les requêtes envoyées au serveur sont simples, elles ne contiennent qu'un patron de triplet et d'autre part parce que la mise en cache joue un rôle important. Dans TPF il existe deux types de cache, le cache Web placé du côté du producteur de données et le cache local placé chez le consommateur de données, comme on peut le voir dans la figure 2.8.

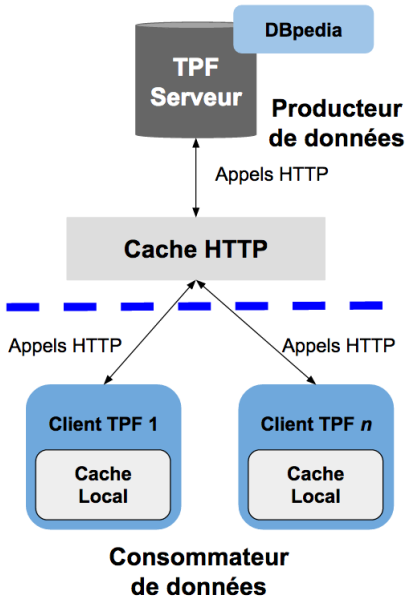


FIGURE 2.8 – Architecture de l'approche TPF du côté du producteur de données et du consommateur de données.

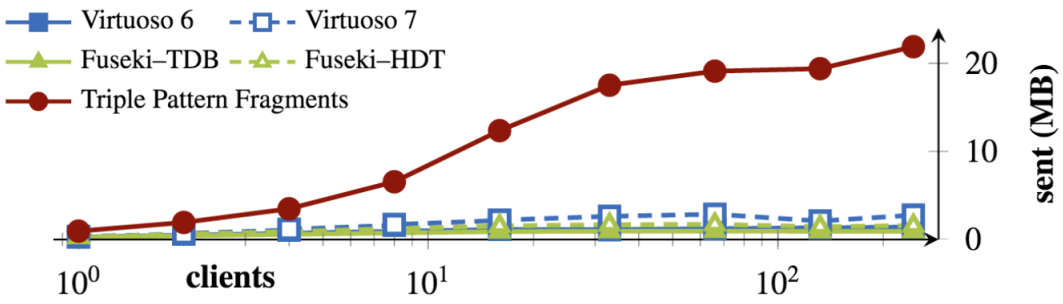


FIGURE 2.9 – Traffic réseau sur le serveur de cache - Source [63].

Lorsqu'un client TPF exécute une requête SPARQL, il la décompose en une multitude de requêtes de fragment. Par exemple, pour la requête sur les architectes 2.8, si on enlève la `LIMIT 5` le client TPF génère 18 669 appels HTTP. Les résultats des requêtes HTTP sont mis en cache dans le cache local du client. Le cache local du client est un cache LRU (*Last Recently Use*) qui contient 100 entrées. Il peut être complètement réécrit d'une requête à l'autre ou au sein d'une même requête, comme la requête des architectes du listing 2.8 sans limite.

Pour chaque requête de fragment le client TPF vérifie dans son cache local. Si la réponse ne s'y trouve pas, la requête est envoyée au producteur de données. Si la réponse est dans le cache Web celle-ci est directement renvoyée au client, sinon le serveur TPF recalcule la réponse, qui est mise en cache dans le serveur de cache Web, puis transmise au client TPF.

Dans [63], les auteurs ont démontré l'importance du cache Web chez le producteur de données, comme l'atteste la figure 2.9. On constate que, comparé aux approches utilisant un serveur SPARQL, avec l'approche TPF le cache Web est plus utile quand le nombre de clients augmente.

Expressivité des requêtes L'approche TPF supporte un sous-ensemble du langage SPARQL. Il permet de faire des requêtes : `SELECT`, `CONSTRUCT`, `ASK` et `DESCRIBE`. TPF n'implémente pas l'opérateur de négation.

Localisation des données Les sources de données sont données par l'utilisateur qui effectue la requête. Le périmètre des données est donc local, TPF n'interroge que les serveurs TPF dont il a l'adresse.

Performances et passage à l'échelle Comme pour les moteurs de requêtes fédérées, un client TPF peut facilement être utilisé par un utilisateur, il peut même être embarqué dans un navigateur Web. Cependant, exécuter des requêtes avec le client TPF génère beaucoup de traffic réseau. Si l'on compare l'exécution d'une seule requête sur un serveur non chargé, celle-ci sera plus rapide sur un serveur SPARQL comparé à un serveur TPF. Mais cette approche permet de passer à l'échelle grâce aux caches mis en place du côté du producteur de données et du consommateur de données.

Hétérogénéité sémantique L'hétérogénéité sémantique n'est pas considérée dans cette approche.

Confiance et usage Les utilisateurs peuvent choisir les sources de données en lesquelles ils ont confiance. Les producteurs de données peuvent observer l'usage de leurs données en analysant les journaux de leurs serveurs TPF.

Fraîcheur des données et cohérence des données La fraîcheur et la cohérence des données sont à la charge des producteurs de données. S'il servent les données sur lesquelles ils ont l'autorité, alors le serveur TPF peut être vu comme un cache.

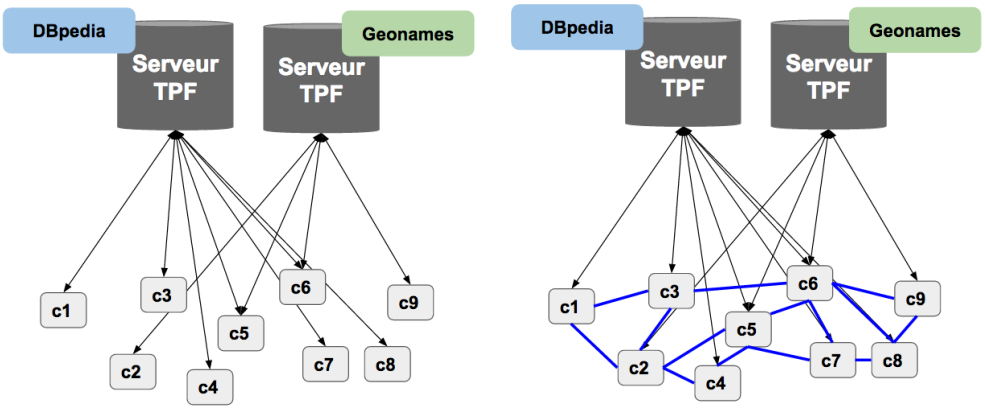
2.7 Synthèse et positionnement de l'approche

Comme nous venons de le présenter, l'exécution de requêtes sur le Web de données reste un problème complexe. Les différentes approches présentent des compromis entre le périmètre des requêtes, les performances, l'hétérogénéité sémantique, la disponibilité des données, la fraîcheur, l'usage et la confiance.

Dans cette thèse, nous nous concentrons sur l'approche «médiateur» et ce, pour plusieurs raisons :

1. Un médiateur permet d'utiliser toute l'expressivité du langage SPARQL.
2. L'utilisateur peut choisir les sources envers lesquelles il a confiance. Par conséquent, la complétude des résultats est limitée aux sources déclarées. Comme les sources de données sont identifiées, il n'est pas nécessaire de chercher quelles sont les sources de données pertinentes sur le Web des données pour une requête comme dans Edutella. Il est juste nécessaire de sélectionner les sources pertinentes au sein des sources déclarées.
3. Les sources de données sont capables d'observer les requêtes entrantes et sont donc capables d'apprécier l'usage fait de leurs données.
4. L'approche médiateur ne demande pas de collaboration entre les sources de données. Elle est capable de s'exécuter sur le Web des données tel qu'il est actuellement.
5. Les principes de publication de données liées ont réduit suffisamment l'hétérogénéité sémantique entre les sources de données pour rendre possible l'écriture de requête SPARQL sur plusieurs sources. En effet, l'utilisation de vocabulaires communs et de relations `sameAs` permet l'écriture de requêtes SPARQL sans recourir aux techniques d'alignement et de réécriture de requêtes [20].

6. Ces mêmes principes permettent l'écriture de médiateurs très compacts capables de s'exécuter dans les navigateurs Web comme l'a montré l'approche TPF.



(a) Sans fédération. (b) Avec fédération.

FIGURE 2.10 – Médiateurs TPF exécutant des requêtes sur un serveur TPF.

En suivant l'approche TPF, il est donc tout à fait possible d'avoir, à un moment donné, des milliers de médiateurs exécutant des requêtes SPARQL sur les mêmes sources de données. Chaque médiateur dispose de sa propre bande passante, d'un cache local et peut exécuter n'importe quelle requête SPARQL. La figure figure 2.10a, montre neuf médiateurs TPF effectuant des requêtes SPARQL sur deux serveurs TPF. Dans l'approche TPF, les serveurs TPF n'exécutent pas de jointures. Une requête SPARQL exécutée au sein d'un médiateur TPF va donc faire de nombreux appels aux serveurs TPF et transmettre de grandes quantités de données sur les médiateurs pour qu'ils puissent calculer les jointures d'une requête. Par exemple, la requête des architectes 2.8 (p. 32) génère $\approx$ 18 670 appels HTTP qui vont retourner chacun des pages de 100 triplets représentant au maximum 10 Ko. Cette requête va donc transférer 186 Mo de données sur le réseau. TPF améliore grandement la disponibilité des serveurs TPF, mais dégrade considérablement les performances des requêtes.

Dans cette thèse, nous proposons de faire collaborer les médiateurs TPF en les regroupant au sein d'une fédération de consommateurs de données. La figure 2.10b décrit schématiquement cette approche. On observe que si les médiateurs collaborent, les sources de données gardent leur autonomie complète. Les médiateurs sont donc regroupés au sein d'un ou plusieurs réseaux pair-à-pair, mais pas les sources de données.

La collaboration entre les médiateurs peut prendre plusieurs formes :

1. Ils peuvent partager leur cache local en utilisant des techniques de cache collaboratif.
2. Ils peuvent exécuter des requêtes SPARQL pour le compte d'un autre participant.
3. Ils peuvent exécuter une sous-requête pour le compte d'un autre participant.

Dans notre contexte, un utilisateur a toujours le choix d'exécuter sa requête sans passer par ses voisins. Dans quelles conditions cette collaboration sera-t-elle fructueuse ? Quels gains en performances ou en disponibilité peut-on attendre en cas de collaboration ?

3

CyCLaDEs : un cache collaboratif décentralisé pour les fragments de triplet

Sommaire

Les producteurs de données liées ont publié sur le Web des millions de triplets [9] et ce nombre continue d'augmenter [51]. Une partie de ces données est disponible sur des serveurs SPARQL publics maintenus par les producteurs de données. Un des problèmes majeurs des serveurs publics est la disponibilité des données [4]. L'approche Triple Pattern Fragment (TPF) [63] (présentée dans le chapitre 2 p. 17) adresse ce problème en balançant le coût d'exécution de requêtes entre les producteurs de données et les consommateurs de données. TPF établit un nouveau compromis entre la disponibilité des données et les performances, réduisant ainsi la pression sur les producteurs de données. Un producteur de données peut alors fournir, à faible coût, plusieurs jeux de données sur un serveur TPF. Dans WarDrobe [7] plus de 657 000 jeux de données sont disponibles sur des serveurs TPF.

Le cache Web joue un rôle important dans les performances d'un serveur TPF [62]. Lors de l'exécution d'une requête SPARQL, un client TPF génère beaucoup d'appels au serveur TPF. Comme il s'agit de requêtes de fragment, une part importante de ces appels est interceptée par un serveur de cache Web. Ce cache réduit la pression sur le serveur TPF.

Toutefois, ces caches HTTP restent à la charge des producteurs de données. De plus, si un serveur TPF expose plusieurs jeux de données le cache peut se révéler inutile si la requête n'est pas populaire.

Pendant l'exécution d'une requête, les clients TPF mettent en cache les résultats des requêtes HTTP. Où les résultats des requêtes HTTP sont des fragments de triplet. Cependant, les clients ne collaborent pas entre eux et ne peuvent donc pas tirer avantage de ce cache décentralisé hébergé par les clients. Construire des caches décentralisés du côté client a déjà été étudié dans les approches basées sur les DHT (*Distributed Hash Table*) [23]. Les approches basées sur les DHTs introduisent une forte latence lors de la recherche de contenu et peuvent réduire les performances du système. Behave [25] construit un cache décentralisé sur un réseau pair-à-pair non structuré. Le cache est construit pour les utilisateurs naviguant sur le Web en exploitant les similarités entre les comportements de navigation des utilisateurs. Le navigateur est directement connecté à un nombre fixe de navigateurs avec des profils de navigation similaires. Un profil de navigation est défini comme similaire en comparant la navigation passée des utilisateurs. Ainsi, une nouvelle adresse URL peut être recherchée dans le cache des voisins sans latence de connexion. Ce type d'approche n'a pas été appliquée dans le contexte du Web des données. Exécuter des requêtes SPARQL et naviguer sur le Web sont deux problèmes différents, en terme d'appels HTTP par seconde et de profilage des clients.

Ce chapitre présente CyCLaDEs [24], une approche qui permet de construire un cache décentralisé hébergé par les clients TPF. Les contributions de ce chapitre sont les suivantes :

- CyCLaDEs, un cache décentralisé du côté des clients. Pour chaque client TPF, CyCLaDEs construit un voisinage de clients TPF hébergeant des fragments TPF similaires dans leur cache. Le cache des voisins est vérifié avant de transmettre la requête au serveur TPF.
- Un algorithme pour construire les profils des clients. Le profil caractérise le contenu du cache d'un client TPF.
- Une évaluation de CyCLaDEs avec BSBM (Berlin Benchmark) [11] dans différentes configurations. Les résultats montrent que CyCLaDEs réduit considérablement la charge du serveur TPF.

La section 3.1 (p. 38) présente l'état de l'art, elle détaille les caches dans le Web des données et les caches collaboratifs dans les systèmes distribués. La section 3.2 (p. 41) décrit l'approche générale de CyCLaDEs. La section 3.3 (p. 42) définit le modèle de CyCLaDEs. La section 3.4 (p. 46) détaille l'environnement d'expérimentation et les expériences menées pour valider l'approche. La section 3.5 (p. 53) conclut ce chapitre.

3.1 État de l'art

L'amélioration de l'exécution de requêtes SPARQL grâce à la mise en cache a déjà été étudiée dans le Web des données. Martin et *al.* [38] proposent de mettre en cache les résultats des requêtes SPARQL et de gérer le remplacement du cache. Schmachtenberg [57] propose une mise en cache sémantique de requêtes, basée sur la similarité des requêtes. Hartig [31] propose la mise en cache pour améliorer l'efficacité et la complétude des résultats pour l'exécution de requêtes de «parcours de liens». Toutes ces approches s'appuient sur une localité

temporelle où des données spécifiques sont supposées être réutilisées dans un court laps de temps et où les caches sont fournis par les producteurs de données. CyCLaDEs s'appuie sur des localités de spécialité où les clients avec un profil similaire sont directement connectés et où les ressources de mise en cache sont fournis par les consommateurs de données.

L'approche de Triple Pattern Fragment (TPF) [62, 61] propose de déplacer le processus complexe de traitement des requêtes des serveurs aux clients, dans le but d'améliorer la disponibilité et la capacité de passage à l'échelle des serveurs SPARQL. Une requête SPARQL est décomposée en requête de fragment et le serveur TPF renvoie aux clients les données correspondant aux triplets reçus. Le client est alors capable d'effectuer les jointures basées sur les opérateurs de boucles imbriquées. Les triplets récupérés durant le traitement de la requête sont mis en cache dans le client TPF et dans un serveur de cache Web, placé devant le serveur TPF. Bien que le traitement d'une requête SPARQL augmente le nombre d'appels au serveur HTTP, une grande partie de ces appels est interceptée par le cache Web, ce qui réduit la charge du serveur TPF, comme démontré dans [62].

TPF se base sur des localités temporelles et les producteurs de données doivent fournir les ressources pour la mise en cache. Comparés aux résultats des autres techniques de mise en cache dans le Web des données, les résultats de la mise en cache par l'interface TPF, augmentent leur réutilité pour d'autres requêtes, c'est-à-dire que la probabilité d'avoir un *cache hit* est plus élevée qu'avec la mise en cache des résultats d'une requête SPARQL. Le but de CyCLaDEs est de découvrir et connecter dynamiquement les clients TPF selon leurs affinités. CyCLaDEs émet l'hypothèse que les clients effectuent un nombre limité de requêtes. Par conséquent, un triplet peut être facilement trouvé dans le cache décentralisé. Pour construire ce cache décentralisé, chaque client TPF doit avoir un nombre limité de voisins avec un accès sans ou à très faible latence. Durant le traitement des requêtes, pour chaque triplet d'une sous requête, CyCLaDEs vérifie s'il se trouve dans le cache local du client, puis dans le cache local de ses voisins. En dernier recours, la requête est transmise au serveur TPF. CyCLaDEs améliore TPF en construisant un cache décentralisé sur les clients, sans avoir besoin de ressources supplémentaires. Les caches décentralisés permettent de réduire les appels aux serveurs TPF. Ce constat est particulièrement vrai si le serveur TPF héberge plusieurs jeux de données. Le cache Web peut contenir les requêtes fréquentes, mais pas tous les appels faits au serveur TPF. En d'autres termes, les requêtes qui ne sont pas populaires ne seront probablement pas dans le cache Web et devront être résolues par le serveur TPF. Dans CyCLaDEs, le voisinage dépend de la similarité des profils. Les clients sont groupés en communautés selon leurs requêtes passées. En procédant ainsi les requêtes peu fréquentes peuvent être stockées dans le cache local des voisins.

Plusieurs domaines de recherche ont proposé des caches décentralisés. Dahlin et *al.* [18] proposent un cache coopératif pour améliorer le temps de réponse des systèmes de fichiers. En analysant la charge de travail de systèmes de fichiers distribués (DFS) à large échelle, Blaze [12] a découvert qu'une grande proportion des éléments non trouvés dans le cache étaient des fichiers déjà stockés dans le cache d'autres clients. Blaze propose un cache hiérarchique dynamique pour réduire le traffic sur le DFS induit par les éléments non trouvés dans le cache et diminuer la charge du serveur. Les recherches sur les réseaux de diffusion de contenu (CDN) pair-à-pair, proposent des caches Web décentralisés comme Squirrel [34], FlowerCDN [23] et Behave [25]. Squirrel et FlowerCDN utilisent des tables de hachage distribuées (DHT) pour indexer le contenu de tous les pairs. Bien que ces approches soient pertinentes, la mise en cache de requêtes est chère en terme de latence. Avec n participants, une DHT requiert $log(n)$ accès pour vérifier la présence d'une clé dans la DHT. Comme l'approche TPF peut générer des milliers d'appels, la latence des DHTs devient un goulot

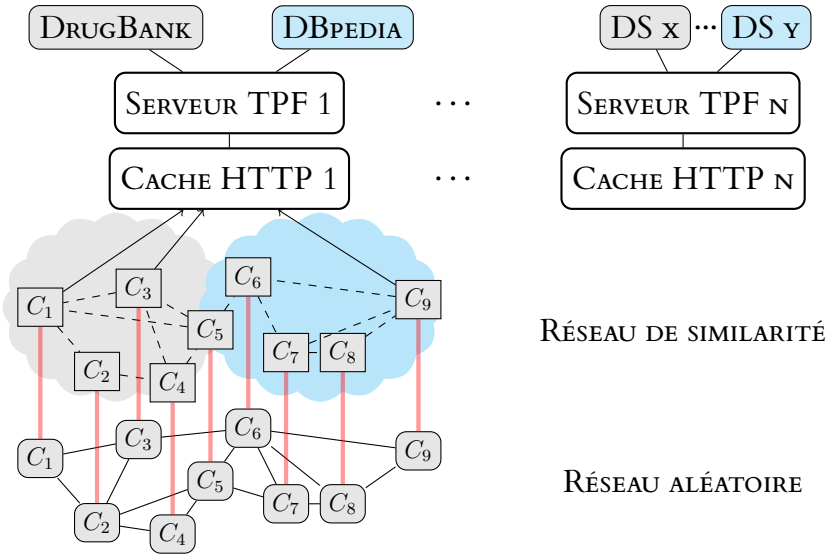


FIGURE 3.1 – Les clients C_1-C_9 représentent les clients TPF exécutant des requêtes sur le serveur TPF 1. Le réseau aléatoire (RPS) connecte les clients sous forme d'un graphe aléatoire. Le réseau de similarité (CON) connecte les mêmes clients (liens pointillés) selon leurs requêtes. Les clients C_1-C_4 exécutent des requêtes sur DrugBank et les clients C_6-C_9 exécutent des requêtes sur DBpedia. Le client C_5 exécute des requêtes sur les deux jeux de données. Le nombre total de serveurs TPF est N.

d'étranglement et interroger une DHT devient considérablement moins performant qu'interroger directement le serveur TPF.

Behave [25] est un cache décentralisé pour la navigation sur le Web. Il est basé sur l'approche Gossple [8]. L'hypothèse de base est la suivante : si deux utilisateurs ont visité la même page dans le passé, ils sont plus susceptibles de présenter des intérêts communs dans le futur. À partir de cette supposition, Behave s'appuie sur des techniques de protocole épidémique pour construire dynamiquement des voisinages à taille fixe basés sur le profil des clients, c'est-à-dire sur les appels HTTP qu'ils ont effectué. Quand un client accède à une nouvelle URL, Behave est capable de vérifier rapidement si cette URL est présente dans le cache des voisins. Comparé à une DHT, le nombre de ressources qui est stocké dans le cache de Behave est plus petit, mais les caches sont accessibles sans ou avec peu de latence. Les ressources disponibles sont personnalisées suivant le comportement des clients, plutôt que selon les localités temporelles.

Dans CyCLaDEs nous souhaitons appliquer l'approche générale de Behave pour les clients TPF. Cependant, comparé à une navigation humaine, un client TPF peut traiter un grand nombre de requêtes par seconde et le cache local du client peut rapidement changer. Nous émettons l'hypothèse que les clients qui ont traité les même requêtes dans le passé seront susceptible de traiter les mêmes requêtes dans le futur. Nous construisons une mesure de similarité en comptant le nombre de prédicats dans les triplets sur une fenêtre glissante. Nous démontrons que cette métrique est efficace pour construire un cache décentralisé pour les clients TPF.

3.2 Motivation et approche de CyCLaDEs

Dans CyCLaDEs, nous faisons l'hypothèse que les clients qui ont exécuté des requêtes similaires dans le passé, exécuteront des requêtes similaires dans le futur [22]. C'est par exemple le cas dans les applications Web qui proposent des formulaires aux utilisateurs, générant des requêtes SPARQL avec des paramètres. Le jeu de données Berlin Benchmark (BSBM) est construit de cette manière [11]. BSBM simule une application Web réaliste où les utilisateurs naviguent à travers des produits et des commentaires. BSBM génère un ensemble de requêtes à partir de 12 modèles de requêtes et 40 prédicats.

CyCLaDEs a pour but de construire un cache collaboratif décentralisé pour le traitement de requêtes de fragment, basé sur la similarité des profils des clients TPF. Pour chaque client, CyCLaDEs sélectionne un nombre fixe de clients les plus similaires appelés *voisins* et établit une connexion directe avec eux. Pendant le traitement d'une requête, pour un client donné et pour chaque requête de fragment, le client vérifie si la réponse est présente dans son cache local, puis dans le cache local de ses voisins et en dernier recours la requête est transmise au serveur TPF. CyCLaDEs introduit une nouvelle étape de vérification lors de l'exécution de requêtes : la vérification dans le voisinage du client. Il est donc important que cette opération soit effectuée rapidement, de manière à ne pas dégrader les performances. Nous pensons que la combinaison du cache collaboratif décentralisé hébergé par les consommateurs de données et le cache temporel hébergé par le producteur de données réduira significativement la charge du serveur TPF.

Afin de construire le voisinage et de prendre en charge l'autonomie des clients, nous avons suivi l'approche générale de Gossple [8]. CyCLaDEs construit deux réseaux superposés côté clients, où l'on suppose qu'il n'y a pas de clients malveillants :

1. un *réseau superposé d'échantillonnage aléatoire des pairs (RPS)* qui maintient les connexions entre les clients. Ce réseau est implémenté par le protocole Cyclon [64]. Chaque client maintient une vue partielle sur l'ensemble du réseau, qui est un sous-ensemble aléatoire de nœuds du réseau. Périodiquement, le client sélectionne le nœud le plus vieux dans sa vue et ils échangent une partie de leurs voisins. Cette vue est utilisée pour initier et maintenir le réseau communautaire.

2. un *réseau superposé communautaire (CON)* construit au-dessus du RPS. Il rassemble les clients selon leur profil. Chaque client maintien une deuxième vue, cette dernière contient les k meilleurs voisins selon la similarité de leur profil avec celui du client. Le maintien des k meilleurs voisins est effectué via le RPS lors des échanges de voisins. Pour minimiser le coût de l'échange de voisins (*shuffling*) : (1) les informations du profil doivent être aussi petites que possible, (2) la mesure de similarité doit être calculée rapidement, afin d'éviter de ralentir le moteur de requêtes.

La figure 3.1 représente des clients TPF. Les clients C_1 à C_4 traitent des requêtes sur DrugBank, les clients C_6 à C_9 traitent des requêtes sur DBpedia et le client C_5 effectue des requêtes sur les deux jeux de données. Le réseau RPS garantit que tous les clients sont connectés au travers d'un graphe aléatoire, tandis que les profils des clients font en sorte que le réseau communautaire converge vers deux communautés. Les clients C_1 à C_4 sont fortement connectés car ils accèdent au même jeu de données, DrugBank. Les clients C_6 à C_9 sont regroupés ensemble parce qu'ils sont intéressés par DBpedia. Concernant le client C_5, il peut faire partie des deux communautés étant donné qu'il effectue des requêtes aussi bien sur DrugBank que sur DBpedia.

Grâce au réseau communautaire, un client est maintenant capable de vérifier la disponibilité d'un fragment dans son voisinage avant d'envoyer sa requête au cache HTTP. En supposant que les clients sont profilés, le cache communautaire devrait être capable d'absorber un nombre important d'appels et de passer à l'échelle, selon le nombre de clients, sans requérir de nouvelles ressources de la part de producteurs de données. Bien évidemment, le cache communautaire est efficace si le voisinage de chaque client est pertinent et si le coût du maintien du réseau est faible.

3.3 Modèle de CyCLaDEs

Dans cette section, nous détaillons les réseaux superposés construits par CyCLaDEs du côté clients.

3.3.1 Réseau d'échantillonnage aléatoire des pairs (RPS)

Le protocole d'échantillonnage aléatoire des pairs [64] permet à chaque client de maintenir une vue aléatoire d'un sous-ensemble du réseau appelé *voisinage*. Une vue est une table à taille fixe qui associe un identifiant client à une adresse IP. La taille de cette vue peut être fixée à $log(N)$, où N est le nombre total de nœuds dans le réseau. Le protocole RPS garantit que le réseau converge rapidement vers un graphe aléatoire sans partitions, c'est-à-dire un graphe connecté.

Pour maintenir la connectivité du réseau superposé, un client sélectionne périodiquement le nœud le plus vieux de sa vue et échange une partie de sa vue avec lui. Cet échange périodique garantit que chaque vue des clients contient toujours un sous-ensemble aléatoire des nœuds du réseau et par conséquent maintient les clients connectés à travers un graphe aléatoire.

Dans CyCLaDEs, pour faciliter l'entrée dans le réseau, le serveur TPF maintient une liste des trois derniers clients connectés. Chaque fois qu'un nouveau client rejoint le réseau il contacte le serveur TPF et reçoit automatiquement une liste des trois derniers clients ayant accédé au serveur et ajoute de façon aléatoire l'un d'entre eux à sa vue. L'échange périodique de voisins rétablit rapidement les propriétés d'un graphe aléatoire du réseau dans lequel est inclus le nouveau client.

3.3.2 Profil des clients TPF

Le réseau superposé communautaire (CON) se base sur le profil des clients TPF. Le profil d'un client doit caractériser le contenu de son cache local. À un instant donné, le contenu du cache est déterminé par les résultats des requêtes du passé récent.

Le cache d'un client TPF est constitué d'une liste de taille fixe de couples *(clé, valeur)* et implémente un cache LRU (*Last Recently Used*). Les entrées utilisées le moins récemment sont supprimées en premier. La *clé* est le sélecteur d'un fragment de triplets (cf. chapitre 2 p. 17) où le prédicat est une constante et la *valeur* est un ensemble de triplets qui correspond à un fragment de triplets [61]. Chaque fragment est divisé en pages et chaque page contient 100 triplets. Le fragment est constitué de façon asynchrone et peut être incomplet.

Par exemple, le fragment $f1$ correspond à 1000 triplets mais à un instant t seulement les 100 premiers triplets ont été récupérés depuis le serveur.

Pour illustrer le fonctionnement du cache, supposons qu'un client TPF exécute la requête SPARQL suivante :

```
SELECT DISTINCT ?book ?author
WHERE {
  ?book rdf:type dbpedia-owl:Book;    # tp1
  dbpedia-owl:author ?author.         # tp2
}
LIMIT 5
```

Listing 3.1 – Requête SPARQL SELECT : Auteurs de livre.

La requête est décomposée en triplets $tp1$ et $tp2$. Le cache local sera rempli de manière asynchrone comme illustré dans le tableau 3.1. Le nombre de triplets correspondant aux livres (31 172) étant plus petit que le nombre de triplets correspondant aux auteurs (39 935), le client TPF commence par récupérer les livres. L'entrée 0 contient $tp1$ sans donnée, l'entrée 1 contient $tp2$ avec quelques données. Le client TPF récupère les livres et commence la boucle imbriquée pour récupérer les auteurs pour un livre donné. Les entrées 2 à 9 contiennent toutes un triplet comme réponse, mais seulement les 5 premiers sont nécessaires pour répondre à la requête étant donnée la limite ($Limit$ 5). Plusieurs stratégies pour calculer le profil d'un client sont alors possibles :

- *Mise en cache des clés* : nous pouvons considérer un cache de vecteurs de clés comme dans Behave [25], en réduisant les dimensions du vecteur grâce à un filtre de bloom. Cependant, le traitement de boucle imbriquée induit une réécriture complète du cache avec la requête suivante. Si une nouvelle requête cherchant des auteurs français est exécutée, alors la boucle imbriquée va itérer sur les auteurs français à la place des livres et va complètement réécrire le cache comme démontré dans le tableau 3.1. Par conséquent, l'état du cache à un instant donné ne reflète pas le cache du passé récent.
- *Requêtes passées* : on peut analyser de façon statique les requêtes du passé et en extraire les prédicats. Malheureusement, cela reflète l'ordonnancement de la jointure lors de l'exécution par le client TPF et ne prend pas en considération la boucle imbriquée.
- *Count-min sketch* : nous pouvons utiliser le count-min sketch [17] pour analyser la fréquence des prédicats traités depuis le début de la session. Cependant, le *count-min sketch* capture l'ensemble des opérations passées et non le passé récent.

Dans CyCLaDEs, nous voulons définir le profil dans l'esprit du *count-min sketch* mais avec une mémoire temporelle réduite, seulement la mémoire du passé récent. Le profil d'un client c est défini comme suit : $Pr(c) = \{(p, f_p, t)\}$, où Pr est une fenêtre de taille fixe sur le flux de triplets traités par le client, p est un prédicat d'un triplet du flux, f_q est la fréquence et t est l'estampillage de la dernière mise à jour de p. Pour éviter de mélanger les mêmes prédicats provenant de sources différentes, la provenance est concaténée avec le prédicat lui-même. Par exemple, le prédicat général `rdfs:label` récupéré depuis DBpedia ne doit pas être confondu avec le même prédicat récupéré de DrugBank. Pour simplifier la notation, à la place d'utiliser le couple *(provenance, prédicat)*, seul le terme *prédicat* sera utilisé.

L'algorithme 1 présente la procédure de profilage de CyCLaDEs. CyCLaDEs intercepte le flux de triplets traité et en extrait les prédicats. Si le prédicat appartient au profil, la fréquence du prédicat est incrémentée et son temps est mis à jour (lignes 4 – 5). Sinon,

	Clés	Triples
0	?book http:.../ontology/author ?author	∅
1	?book http://www.w3.org/1999/02/22-rdf-syntax-ns#type http:.../ontology/Book	http:.../resource/%22...And_Ladies_of_the_Club%22 http://www.w3.org/1999/02/22-rdf-syntax-ns#type http:.../ontology/Book ... http:.../resource/%22K%22_Is_for_Killer http://www.w3.org/1999/02/22-rdf-syntax-ns#type http:.../ontology/Book
2	http:.../resource/%22...And_Ladies_of_the_Club%22 http:.../ontology/author ?author	http:.../resource/%22...And_Ladies_of_the_Club%22 http:.../ontology/author http:.../resource/Helen_Hooven_Santmyer
3	http:.../resource/%22A%22_Is_for_Alibi http:.../ontology/author ?author	http:.../resource/%22A%22_Is_for_Alibi http:.../ontology/author http:.../resource/Sue_Grafton
4	http:.../resource/%22B%22_Is_for_Burglar http:.../ontology/author ?author	http:.../resource/%22B%22_Is_for_Burglar http:.../ontology/author http:.../resource/Sue_Grafton
5	http:.../resource/%22C%22_Is_for_Corpse http:.../ontology/author ?author	http:.../resource/%22C%22_Is_for_Corpse http:.../ontology/author http:.../resource/Sue_Grafton
6	http:.../resource/%22D%22_Is_for_Deadbeat http:.../ontology/author ?author	http:.../resource/%22D%22_Is_for_Deadbeat http:.../ontology/author http:.../resource/Sue_Grafton
7	http:.../resource/%22E%22_Is_for_Evidence http:.../ontology/author ?author	∅
8	http:.../resource/%22F%22_Is_for_Fugitive http:.../ontology/author ?author	∅
9	http:.../resource/%22G%22_Is_for_Gumshoe http:.../ontology/author ?author	∅

TABLE 3.1 – Cache local d'un client TPF après l'exécution de la requête dans le listing 3.1.

Algorithme 1: CalculerProfil(s : Flux de triplets, w : Taille de la fenêtre, t : Estampille)

1 $Pr \rightarrow \varnothing$
2 **while** *flux de données continue* **do**
3 Reçois le prochain triplet du flux $tp = (s\,p\,o)$
4 **if** $(tp.p, f_p, \_) \in Pr$ **then**
5 $Pr.miseAJour(tp.p,\ f_p + 1, t)$
6 **else**
7 $Pr \leftarrow Pr \cup (tp.p, 1, t)$
8 **if** $|Pr| > w$ **then**
9 $Pr \setminus (p_i, f_{p_i}, t_i) : (p_i, f_{p_i}, t_i) \in Pr \wedge \nexists\, (p_j, f_{p_j}, t_j) \in Pr : t_j < t_i\}$

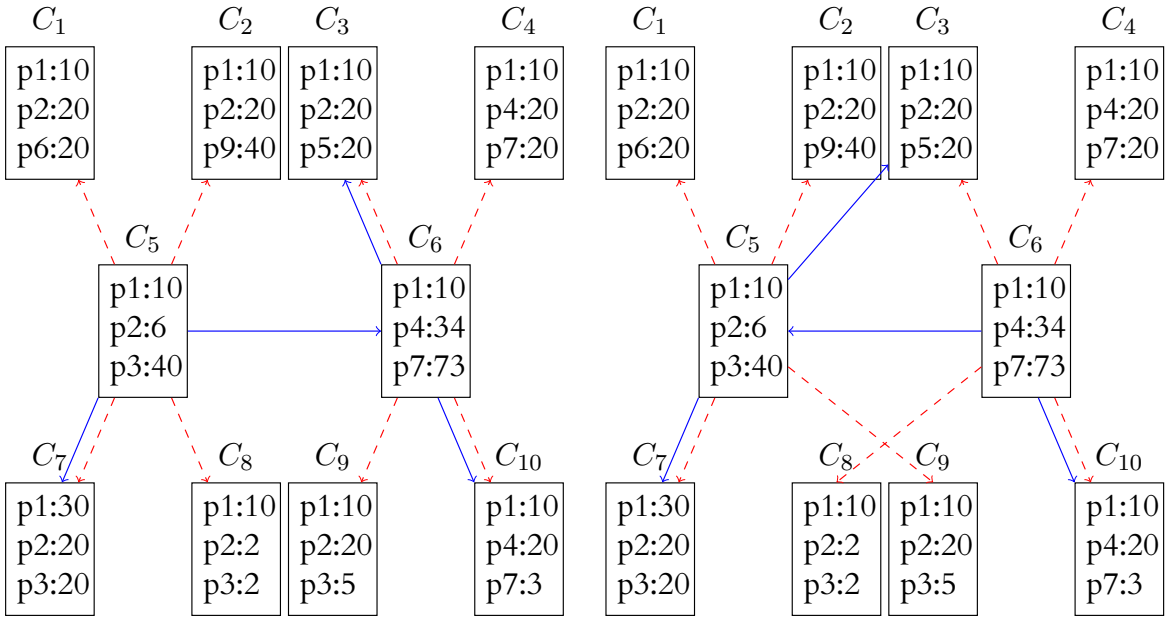
CyCLaDEs insère une nouvelle entrée dans le profil (ligne 7). Si la structure dépasse w entrées, alors CyCLaDEs supprime l'entrée la plus ancienne (lignes 8 – 9). Cet algorithme de profilage est créé pour prendre en compte les boucles imbriquées et oublier les prédicats qui ne sont pas fréquemment utilisés. Pour le client dont le cache est détaillé dans le tableau 3.1, après l'entrée 4 dans le cache, le profil est :

```
{(http://www.w3.org/1999/02/22-rdf-syntax-ns#type,1),
 (http://dbpedia.org/ontology/author, 3)}
```

3.3.3 Réseau communautaire et mesure de similarité

CyCLaDEs se base sur le réseau superposé d'échantillonnage aléatoire des pairs pour maintenir le voisinage et le réseau superposé communautaire pour maintenir les k meilleurs voisins. Concrètement, le réseau communautaire est simplement une seconde vue du réseau hébergée par chaque client. Cette vue est composée d'une liste des k voisins avec le profil le plus similaire. La vue est mise à jour pendant la phase d'échange des voisins : quand un client démarre le processus d'échange de voisins, il sélectionne le nœud le plus vieux et échange avec lui des informations sur son profil, si ce client a des voisins ayant une plus grande similarité dans sa vue, alors la vue locale est mise à jour dans le but de garder les k meilleurs voisins.

Le coefficient de similarité Jaccard généralisé est utilisé pour définir si un profil est



(a) C_5 commence l'échange de vue avec C_6. (b) Etat après l'échange de vue.

FIGURE 3.2 – Réseau partiel de CyCLaDEs centré sur C_5 et C_6. Les lignes solides représentent des clients dans la vue RPS (2 clients). Les lignes en pointillées représentent des clients dans la vue CON (4 clients). Chaque client à un profil de taille 3 défini comme suit : (*prédicat: fréquence*).

meilleur qu'un autre :

$$J(x,y) = \frac{\sum_i min(x_i, y_i)}{\sum_i max(x_i, y_i)}$$

Où x et y sont deux multi-ensembles et les nombres naturels $x_i \geq 0$ et $y_i \geq 0$ sont la multiplication des items i dans chaque multi-ensemble.

Exemple 3. *La figure 3.2 décrit un exemple de réseau de CyCLaDEs. Prenons l'exemple de trois clients : C_5, C_8 et C_9, avec les profils présentés dans le tableau 3.2. Nous voulons savoir quel client est le plus similaire à C_5. Pour cela nous allons calculer la similarité Jaccard pour C_5 et C_8, puis pour C_5 et C_9. Nous allons commencer par calculer la similarité Jaccard entre C_5 et C_8. La première étape est de calculer le numérateur ($\sum_i min(x_i, y_i)$). Pour chaque prédicat : P_1, P_2 et P_3 nous allons prendre la plus petite valeur entre la fréquence de C_5 et celle de C_8. Pour P_1 la valeur est la même pour C_5 et C_8 : 10. Pour P_2 la valeur de C_5 est 6 et pour C_8 : 2, la plus petite valeur de P_2 est donc 2. On effectue la même opération avec P_3.*

$$J(C_5, C_8) = \frac{(10+2+2)}{\sum_i max(x_i, y_i)}$$

Pour calculer le dénominateur ($\sum_i max(x_i, y_i)$) nous allons prendre la valeur maximale pour chaque prédicat :

$$J(C_5, C_8) = \frac{(10+2+2)}{(10+6+40)} = \frac{14}{56} = 0,25$$

On calcule la similarité Jaccard pour comparer C_5 et C_9 :

$$J(C_5, C_9) = \frac{(10+6+5)}{(10+20+40)} = \frac{21}{70} = 0,3$$

	P_1	P_2	P_3
C_5	10	6	40
C_8	10	2	2
C_9	10	20	5

TABLE 3.2 – Exemple de profil pour les clients C_5, C_8 et C_9, avec un profil à trois entrées.

On peut en déduire que C_9 est plus similaire à C_5 que C_8.

La figure 3.2 représente le réseau de CyCLaDEs, elle se focalise sur les clients C_5 et C_6. La taille de vue du réseau RPS est fixée à deux entrées et est représentée par les lignes solides. La taille de vue du réseau CON est fixée à quatre entrées et est représentée par les lignes en pointillés. Chaque client a un profil de trois entrées, c'est-à-dire qu'il contient les trois prédicats les plus utilisés dans le passé récent. La variable p_i représente un prédicat et le nombre associé est sa fréquence d'utilisation dans le passé récent. La figure 3.2a illustre l'état de C_5 et C_6 avant que C_5 ne commence l'échange de voisins avec C_6. Le voisin C_6 est choisi parce que c'est le nœud le plus vieux dans la vue RPS de C_5. La figure 3.2b représente l'état de C_5 et C_6 après l'opération d'échange des voisins. On peut constater que les voisins de la vue RPS ont changé pour les clients C_5 et C_6. C'est le résultat de l'échange de la moitié des voisins comme décrit dans le protocole Cyclon [64]. Pour les vues CON, C_5 intègre C_9 dans sa vue, tandis que C_6 intègre C_8. Durant l'échange de vues, C_5 récupère les profils présents dans la vue CON de C_6, ainsi que le profil de C_6. Ensuite il classe tous les profils selon le coefficient de Jaccard généralisé et garde les quatre meilleurs. Ainsi le client C_9 est plus similaire au client C_5 que le client C_8, parce que $J(C_5, C_9) = 0,3$ et $J(C_5, C_8) = 0,25$. Par conséquent, C_5 supprime C_8 et intègre C_9 à la place. Le client C_6 suit la même procédure et remplace C_9 par C_8.

3.4 Étude expérimentale

L'objectif de l'étude expérimentale est d'évaluer l'efficacité de CyCLaDEs. Nous allons mesurer principalement le *cache hit*.

3.4.1 Environnement d'expérimentation

Dans cette section nous décrivons l'environnement et la configuration d'expérimentation, ainsi que les métriques observées.

Jeux de données et requêtes Nous utilisons le jeu de données Berlin Benchmark (BSBM) [11], comme dans [61]. Nous avons généré trois jeux de données avec 1 million de triplets, 10 millions de triplets et 100 millions de triplets. Les requêtes ont été générées à partir du cas d'utilisation *explore*. Ce cas d'utilisation simule une navigation sur un site Web. Nous avons généré 100 ensembles de requêtes, chaque ensemble étant constitué de 25 requêtes.

Paramètre	Valeurs
Nombre de clients	10 - 50 - 100
Vue RPS	4 - 6 - 7
Vue CON	9 - 15 - 20
Cache Local	100 - 1000 - 10 000
Taille du profil	5 - 10 - 30
Temps de *shuffle*	10s
Jeux de données	BSBM 1M - BSBM 10M - BSBM 100M
Requêtes	25 sur BSBM

TABLE 3.3 – Récapitulatif des paramètres d'expérimentation de CyCLaDEs.

Implémentation Nous étendons le client TPF avec le modèle de CyCLaDEs présenté en section 3.3 (p. 42). Le code source de CyCLaDEs est disponible à l'adresse : https://github.com/pfolz/cyclades [1].

Environnement L'environnement d'expérimentation est composé :

- d'un serveur TPF exposant les jeux de données BSBM ;
- d'un serveur Web de mise en cache, NGINX. Il est configuré avec un cache de 1 Go ;
- et de clients.

Un serveur HPC exécute le serveur TPF, le serveur de cache Web et les clients. Le serveur HPC a la configuration suivante : 40 processeurs, 130 Go de mémoire, tournant sur Debian 7.8.

Chaque client a son propre ensemble de requêtes à exécuter, soit 25 requêtes, qu'il exécute toujours dans le même ordre, afin de préserver la simulation de navigation créée par le cas d'utilisation. Toutes les expériences se sont déroulées en deux étapes. La première étape permet d'initialiser le réseau, le serveur de cache Web et le cache local des clients. La deuxième étape est lancée après une barrière de synchronisation et effectue les mesures des expériences. Dans les deux étapes, chaque client exécute le même ensemble de requêtes dans le même ordre.

Configuration Le tableau 3.3 présente les différentes configurations utilisées dans les expériences. Les valeurs des paramètres ont été variées selon l'objectif de l'expérimentation comme expliqué dans les prochaines sections. Le temps de *shuffle*, phase d'échange de voisins, est fixé à 10 secondes pour toutes les expériences.

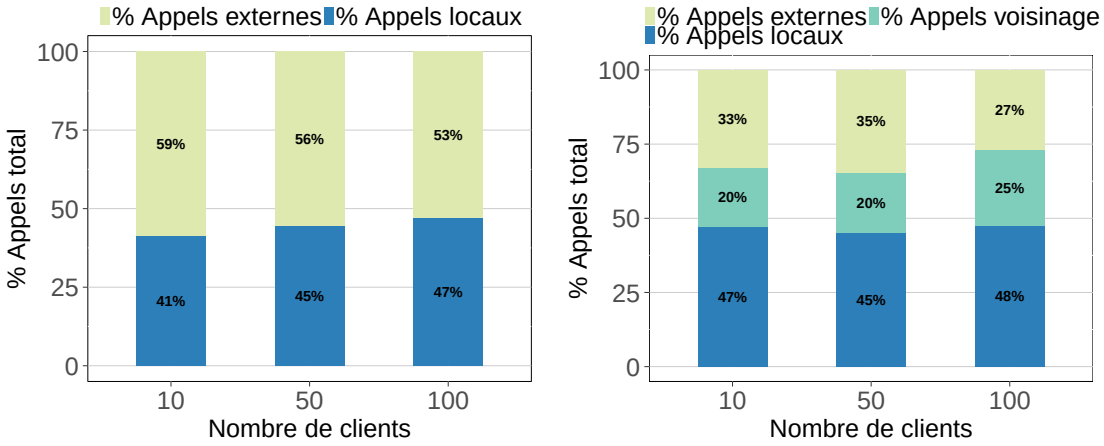
Métriques Les expériences mesurent principalement le *cache hit*, c'est le nombre d'appels résolus dans un cache. Dans le cadre de CyCLaDEs nous faisons la distinction entre les appels résolus dans le cache local, dans le cache décentralisé et chez le producteur de données (serveur de cache Web et serveur TPF).

[1] L'implémentation actuelle ne comprend pas l'introduction dans le réseau, ni le transfer de fragments. Les données sont récupérées depuis le serveur TPF.

3.4.2 Résultats

Cette section présente l'ensemble des résultats des expérimentations.

3.4.2.1 Impact du nombre de clients sur le cache distribué



(a) Client TPF original. (b) Client TPF avec CyCLaDEs.

FIGURE 3.3 – Impact du nombre de clients sur le *cache hit* : (10 clients, $Vue_{RPS} = 4$, $Vue_{CON} = 9$), (50 clients, $Vue_{RPS} = 6$, $Vue_{CON} = 15$) et (100 clients, $Vue_{RPS} = 7$, $Vue_{CON} = 20$).

Objectif : Est-ce que le cache distribué passe à l'échelle ?

Description : La taille du cache local a été fixée à 1000 entrées pour chaque client avec le jeu de données BSBM 1M. La taille de la vue RPS et celle de la vue CON sont fixées à (4,9) pour 10 clients, (6,15) pour 50 clients et (7,20) pour 100 clients.

Résultats : La figure 3.3a présente les résultats d'un client TPF sans CyCLaDEs. Approximativement ≈45% des appels sont traités par le cache local, peu importe le nombre de clients et ≈46% des appels sont traités par le serveur TPF ou le serveur de cache Web.

La figure 3.3b présente les résultats obtenus avec CyCLaDEs. Les performances du cache local sont quasiment pareilles, ≈45% des appels sont traités par le cache local. Cependant, ≈22% du total des appels est résolu par le voisinage, réduisant considérablement le nombre d'appels au serveur TPF.

Explication : Dans la figure 3.3b on peut voir que le nombre d'appels résolus par le voisinage est supérieur de 5% quand il y a 100 clients, par rapport aux expériences avec 10 et 50 clients. La variation du nombre de clients entraîne également la variation du nombre de voisins dans les tables de voisinage. Lors de l'expérience avec 100 clients, un client a plus de voisins similaires que dans les deux autres expériences. Son cache décentralisé est donc plus grand.

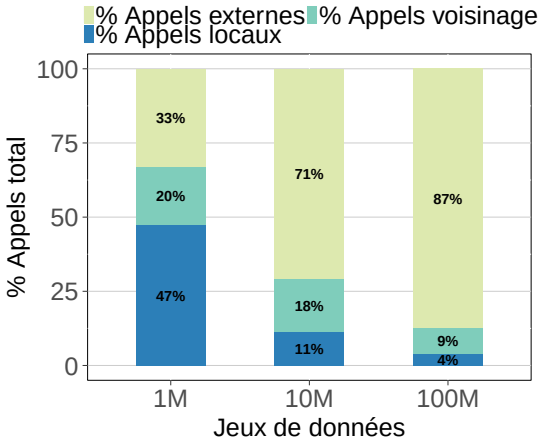


FIGURE 3.4 – Impact de la taille du jeu de données sur le *cache hit*. Pour 10 clients TPF avec $Vue_{RPS} = 4$, $Vue_{CON} = 9$ et $Vue_{Profil} = 10$.

3.4.2.2 Impact de la taille du jeu de données sur le cache décentralisé

La taille du jeu de données peut impacter les performances du cache décentralisé, car un jeu de données plus conséquent augmente la diversité du cache temporel.

Objectif : Est-ce que le nombre de triplets d'un jeu de données peut dégrader les performances du cache décentralisé ?

Description : Pour cette expérimentation, trois jeux de données sont utilisés, BSBM avec 1M de triplets, BSBM avec 10M de triplets et BSBM avec 100M de triplets. Un cache local de 1000 entrées, un profil de taille 10 et 10 clients TPF. Comme dans l'expérience précédente, les vues sont fixées à $Vue_{RPS} = 4$ et $Vue_{CON} = 9$.

Résultats : La figure 3.4 montre le pourcentage d'appels résolus par le cache local, le cache des voisins et le serveur TPF avec les différents jeux de données. Les appels dans le cache local dépendent considérablement de la taille du jeux de données, le *cache hit* est de 47% avec le jeux de données BSBM 1M et descend à 4% avec BSBM 100M. De même, les appels au cache décentralisé sont liés à la taille du jeu de données, avec un *cache hit* autour de 19% pour BSBM 1M et 10M et une descente à 9% pour BSBM 100M.

Explication : La différence du pourcentage de réutilisation du cache local entre BSBM 1M et BSBM 100M est dûe à la taille limitée du cache. Le nombre de triplets dans le jeu de données augmente, mais pas la taille du cache local des clients. De plus, les localités temporelles du cache réduisent l'utilité du cache décentralisé. Cette explication est également valable pour le cache décentralisé dont l'utilité est réduite entre BSBM 10M et BSBM 100M.

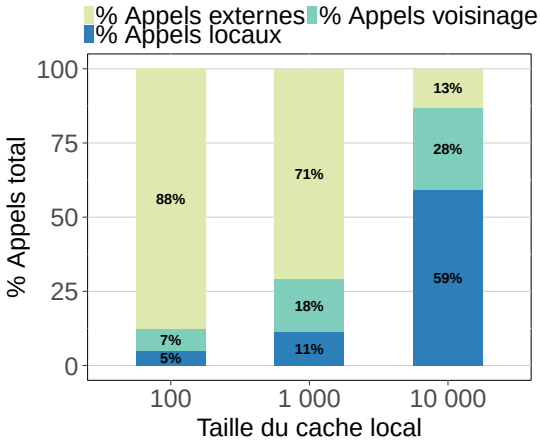


FIGURE 3.5 – Impact de la taille du cache local sur le *cache hit*. Pour 10 clients TPF avec $Vue_{RPS} = 4$, $Vue_{CON} = 9$ et $Vue_{Profil} = 10$.

3.4.2.3 Impact de la taille du cache

Dans cette expérience nous souhaitons voir l'impact de la taille du cache sur le cache décentralisé.

Objectif : Est-ce qu'augmenter le cache local des clients améliore les performances du cache décentralisé ?

Description : Nous utilisons les paramètres suivants : BSBM 10M, 10 clients TPF, $Vue_{RPS} = 4$ et $Vue_{CON} = 9$.

Résultats : La figure 3.5 montre que le nombre d'appels résolus par le cache local décentralisé est proportionnel à la taille du cache. Pour un cache local avec 100 entrées, le *cache hit* est de 5% pour le cache local et 7% pour le cache décentralisé. Pour le cache local avec une taille de 1000 entrées, le *cache hit* du cache décentralisé est de 18%, soit plus grand que le pourcentage d'appels traité par le cache local, qui est de 11%. Le cache décentralisé est donc plus efficace que le cache local. La situation est inversée avec un cache de 10 000 entrées, dans ce cas, le cache local absorbe 59% des appels, le cache décentralisé 28% et seulement 13% des appels vont au serveur.

Explication : Pour l'expérience avec le cache local à 10 000 entrées, 2/3 des appels sont résolus localement, car la plupart des réponses sont déjà présentes dans le cache grâce au premier tour de l'expérience qui permet d'initialiser le cache local des clients. Mais cela profite également aux voisins car le cache décentralisé absorbe 10% d'appels en plus qu'avec un cache local à 1000 entrées. En augmentant la taille du cache local d'un client, le cache décentralisé devient également plus grand et la probabilité de trouver ce que l'on souhaite chez ses voisins est de ce fait plus grande.

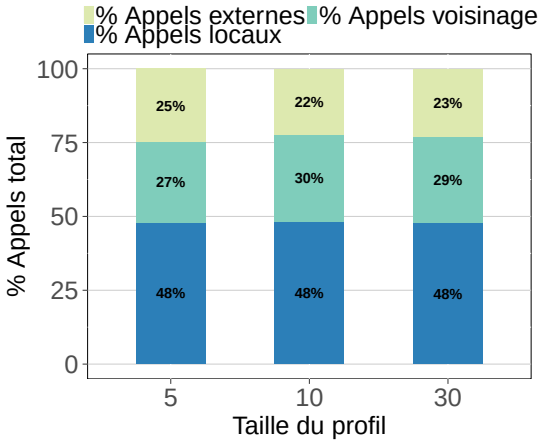


FIGURE 3.6 – Impact de la taille du profil sur le *cache hit* pour deux jeux de données avec 50 clients pour chaque jeu de données. $Vue_{RPS} = 6$, $Vue_{CON} = 15$, $Vue_{profil} = 5, 10$ et 30.

3.4.2.4 Impact de la taille du profil sur le *cache hit*

Dans CyCLaDEs chaque client a un profil qui représente les requêtes qu'il a effectuées dans le passé récent. Ce profil permet à un client de se comparer à ses voisins et de savoir s'ils lui sont similaires.

Objectif : Est-ce que la précision du profil des clients impacte les performances du cache décentralisé ?

Description : Nous effectuons une expérience avec deux jeux de données BSBM 1M différents, hébergés sur le même serveur TPF avec deux URLs différentes. Chaque jeu de données a sa propre communauté de 50 clients les sollicitant pour répondre à leurs requêtes. Comme expliqué dans la section 3.3.2 (p. 42), nous utilisons la provenance pour différencier les prédicats dans le cache local du client. Tous les clients avaient $Vue_{RPS} = 6$, $Vue_{CON} = 15$ et un cache avec 1000 entrées. Nous avons varié la taille du profil à 5, 10 et 30 prédicats.

Résultats : La figure 3.6 démontre que les performances de CyCLaDEs sont similaires peu importe la taille du profil. Environ ≈30% des requêtes sont résolues dans le cache décentralisé.

Explication : La différence entre un profil de taille 5 et un profil de taille 10 ou 30 est d'environ ≈3%. Dans BSBM les requêtes générées utilisent souvent 16 prédicats différents. Par conséquent, avoir 5 entrées dans le profil n'est pas suffisant pour avoir une bonne similarité.

3.4.2.5 Distribution des requêtes

Lorsqu'un client effectue des requêtes SPARQL, celles-ci sont décomposées en une multitude de requêtes de fragment. Pour chaque requête de fragment, si le résultat de la requête de fragment n'est pas présent dans le cache local du client, celui-ci va demander à tous ses voisins dans sa vue de similarité (CON).

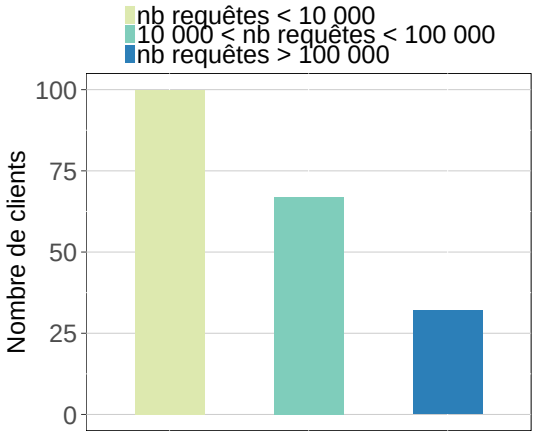
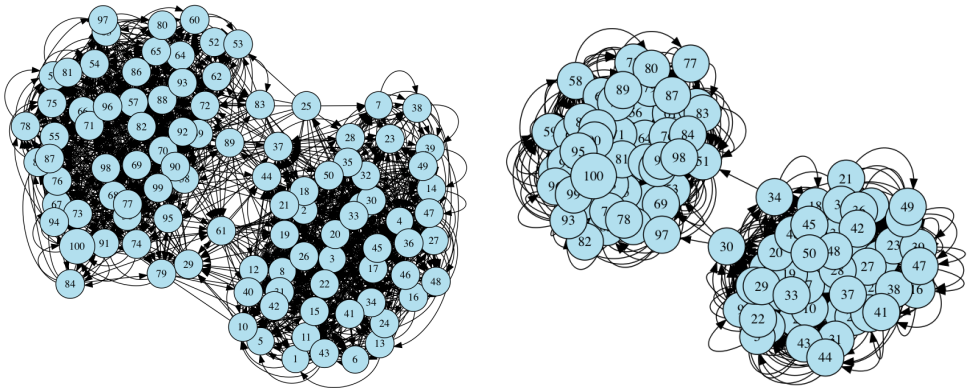


FIGURE 3.7 – Distribution des requêtes sur les clients.



(a) Taille du profil = 5. (b) Taille du profil = 30.

FIGURE 3.8 – Impact de la taille du profil sur la similarité du réseau superposé communautaire.

Objectif : Est-ce qu'il y a des points chauds dans le cache décentralisé ?

Description : Comme dans les expérimentations précédentes, nous effectuons deux nouvelles expérimentations avec deux jeux de données BSBM 1M hébergés sur le serveur TPF avec deux URLs différentes. Nous voulons vérifier s'il y a un point chaud, c'est-à-dire un seul client qui reçoit beaucoup d'appels externes sur son cache local.

Résultats : La figure 3.7 montre la distribution des requêtes sur les clients. Comme nous pouvons le voir, la plupart des clients gèrent environ 10 000 appels à leur cache local et seulement peu de clients gèrent plus de 100 000 entrées dans leur cache local.

Explication : Dans CyCLaDEs le réseau superposé de similarité est basé sur le réseau superposé aléatoire. Par conséquent, la vue de similarité d'un voisin est mise à jour.

3.4.2.6 Impact de la taille du profil sur les communautés du réseau superposé communautaire

Un client établit si un autre client est similaire à lui grâce aux profils. Les clients similaires forment ainsi des communautés.

Objectif : Est-ce que la précision du profil impacte la définition des communautés ?

Description : Comme dans les expériences précédentes, nous mettons en place deux jeux de données BSBM 1M avec 50 clients par jeu de données. Nous varions la taille des profils à 5, 10 et 30 prédicats.

Résultats : Dans la figure 3.8, le graphe connecté représente le réseau superposé communautaire, où un nœud représente un client dans le réseau et les arcs représentent une connexion entre deux clients. Par exemple, l'arc $1 \rightarrow 2$ signifie que le *client 1* a le *client 2* dans sa vue CON. La figure 3.8 montre clairement que CyCLaDEs est capable de construire deux communautés pour les deux valeurs de la taille du profil.

Explication : Comme on peut le voir dans la figure 3.8b, un profil avec une taille plus grande améliore la définition des communautés, c'est-à-dire que seuls les clients avec des profils similaires reçoivent des demandes pour récupérer des fragments.

3.5 Conclusion

Dans ce chapitre nous avons présenté CyCLaDEs, un cache collaboratif décentralisé pour les clients TPF. Ce cache est hébergé par les clients et complète le cache temporel HTTP hébergé par les producteurs de données.

Les résultats expérimentaux démontrent que le cache collaboratif décentralisé est capable de prendre en charge un nombre important de requêtes de fragment générées par le traitement de requêtes TPF, dans le contexte d'une application Web, comme décrit dans la section 3.2 (p. 41). Par conséquent, la pression sur les resources du producteur de données est réduite.

CyCLaDEs propose un algorithme peu coûteux capable de profiler des clients TPF selon leurs requêtes passées et de rassembler les voisins les plus similaires entre eux. Ce profilage est efficace comme le démontre les expériences précédentes. CyCLaDEs démontre comment amener les données aux requêtes grâce à des techniques de mise en cache, une autre approche pourrait être d'amener les requêtes là où sont les données en choisissant parmi les voisins, qui est capable de traiter plus d'un triplet d'une même requête.

4

Ladda : délégation de requêtes dans une fédération de consommateurs de données liées

Sommaire

Dans le chapitre précédent, les clients collaborent en partageant leurs capacités de stockage. Dans ce chapitre, nous nous intéressons au partage du calcul et de la bande passante. En effet, l'approche TPF [63] transforme n'importe quel client TPF en *serveur SPARQL*. Ainsi, n'importe quel client TPF est capable d'exécuter n'importe quelle requête SPARQL. Il est donc possible pour un client ayant de nombreuses requêtes à exécuter d'en déléguer une partie à d'autres clients. Cette approche permet de partager non seulement les capacités de calcul, mais aussi la bande passante des différents clients.

La délégation permet de paralléliser l'exécution des requêtes sans requérir de ressources supplémentaires de la part du producteur de données. L'exécution de requêtes en parallèle peut réduire le temps d'exécution des requêtes, c'est-à-dire le temps entre l'arrivée des requêtes et l'obtention des résultats.

Dans ce chapitre nous proposons Ladda, une approche permettant de déléguer des requêtes dans une fédération de consommateurs de données. Les contributions de ce chapitre

sont les suivantes :

- Un modèle permettant de déléguer des requêtes SPARQL au sein d'une fédération de consommateurs de données.
- Un algorithme dynamique d'équilibrage de charge.
- Une expérimentation basée sur les journaux de DBpedia 3.8 (USEWOD 2013).

La section 4.1 (p. 56) présente le contexte de Ladda. Le modèle de Ladda ainsi que le problème scientifique sont présentés dans la section 4.2 (p. 57). L'approche est présentée en détails dans la section 4.3 (p. 59). La section 4.4 (p. 64) détaille l'environnement d'expérimentation et les expériences menées pour valider l'approche. La section 4.5 (p. 74) présente l'état de l'art. La section 4.6 (p. 75) conclut ce chapitre.

4.1 Contexte et Motivation

Dans Ladda nous faisons l'hypothèse qu'il existe une fédération de consommateurs de données où certains d'entre eux sont libres. L'analyse des journaux de DBpedia 3.8 sur 24 heures [1] permet de vérifier cette hypothèse. La figure 4.1 montre la distribution du nombre de requêtes par nombre de clients. On peut voir que la plupart des clients exécutent moins de 10 requêtes et seulement quelques rares clients exécutent plus de 10 requêtes. Cette analyse suggère l'existence de clients libres, sous réserve que ces derniers restent connectés et ne quittent pas immédiatement le réseau après avoir fini l'exécution de leurs requêtes.

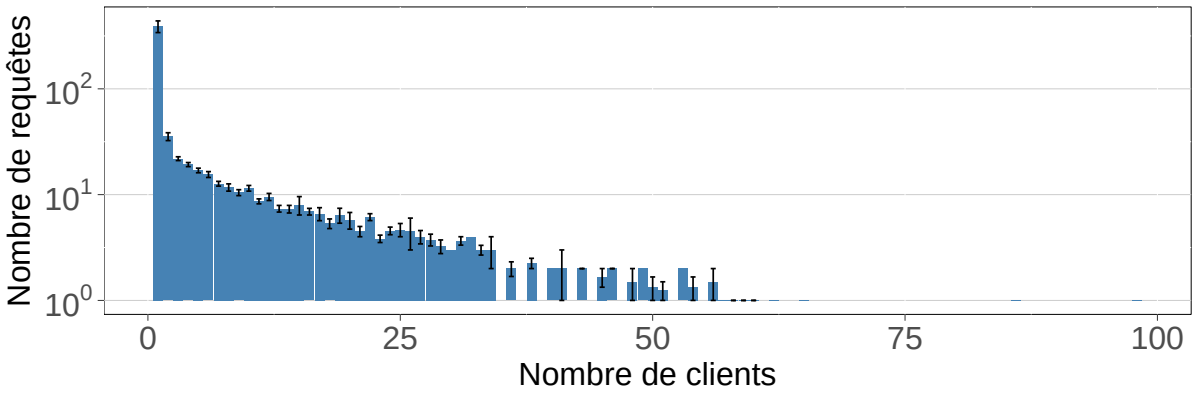


FIGURE 4.1 – Nombre de clients ayant exécuté le même nombre de requêtes en moyenne sur 24 heures, sur le seveur SPARQL public de DBpedia, entre le 26 août à 05:00 et le 27 août à 04:00, en 2012. Avec un écart-type sur le nombre de requêtes exécutées par le nombre de clients.

Ladda a pour but de construire une fédération de consommateurs de données où les consommateurs de données peuvent profiter du temps libre de leurs voisins. Pour illustrer la délégation de requêtes, supposons une fédération de trois consommateurs de données, C_1, C_2 et C_3 avec les mêmes capacités d'exécution et exécutant des requêtes sur le même serveur

[1] Le journal inclus les requêtes exécutées sur le serveur public de DBpedia entre le 26 août à 05:00 et le 25 août à 04:00 pour l'année 2012. Le log des requêtes est disponible à ftp://download.openlinksw.com/support/dbpedia/.

Ordonnancement	Temps de réponse	C_1	C_2	C_3
s_1	$t_0 + et$	Q_1	Q_4	Q_6
	$t_0 + 2et$	Q_2	Q_5	
	$t_0 + 3et$	Q_3		
s_2	$t_0 + et$	Q_1	Q_4	Q_6
	$t_0 + 2et + \varepsilon$	Q_2	Q_5	Q_3

TABLE 4.1 – Stratégies d'ordonnancement possibles pour Q_1 à Q_6 sur les consommateurs de données C_1, C_2 et C_3.

TPF. Dans cet exemple nous supposons qu'un client ne peut exécuter qu'une requête à la fois. Considérons un ensemble de six requêtes indépendantes Q_1 à Q_6 avec le même temps d'exécution (et) et arrivant sur différents consommateurs de données au temps t_0. Le client C_1 reçoit Q_1, Q_2, Q_3 ; le client C_2 reçoit Q_4, Q_5 et le client C_3 reçoit Q_6.

Le tableau 4.1 présente deux stratégies possibles pour l'exécution des requêtes :

- Dans la solution s_1 un consommateur de données exécute ses propres requêtes. C'est le comportement actuel des clients TPF. Le temps de réponse de la fédération est $3et$, soit le temps requis par le consommateur de données le plus chargé, C_1, pour exécuter ses requêtes.
- Dans la solution s_2 un consommateur de données délègue ses requêtes. Au temps $t0 + et$, C_1 détecte que C_3 est libre, donc C_1 délègue Q_3 à C_3 et exécute Q_2 localement. Lors de l'exécution d'une requête, un client peut être interrompu par une ou plusieurs demandes de la part de ses voisins pour exécuter leurs requêtes, ce qui peut ralentir l'exécution de ses propres requêtes. Le coût de cette communication est : ε. Par conséquent, les résultats de Q_3 sont attendus au temps $t0 + et + \varepsilon$. Si $\varepsilon < et$, la solution s_2 réduit le temps d'exécution global de la fédération et le temps de réponse du consommateur de données C_1, comparé à la solution s_1.

Grâce à l'équilibrage de charge et au parallélisme inter-requêtes, le temps d'exécution de $Q1$ à $Q6$ est réduit si le coût de délégation reste faible ($\varepsilon < et$). D'un autre côté, le parallélisme inter-requêtes a une limite. Comme les clients n'hébergent pas les données, les requêtes $Q1$ à $Q6$ sont exécutées sur le même serveur TPF. Même si beaucoup de clients sont libres, il est inutile de paralléliser à partir d'un certain seuil déterminé par les capacités des serveurs TPF. Ladda adresse le problème suivant :

> Étant donné une fédération de consommateurs de données, un ensemble de requêtes distribuées à travers les consommateurs de données, comment déléguer des requêtes afin de réduire le temps de réponse de l'ensemble des requêtes de la fédération et le temps de réponse pour chaque consommateur de données ?

4.2 Définitions et énoncé du problème

Comme dans le chapitre précédent (cf. chapitre 3 p. 37), Ladda s'appuie sur une fédération de consommateurs de données, connectés à travers un réseau superposé d'échantillonnage aléatoire des pairs [64] (*Random Peer Sampling* (RPS)). Chaque participant est connecté à un nombre fixe de voisins qui constitue sa *vue*. Cette vue est renouvelée périodiquement en la

mélangeant avec la vue de ses voisins. Ainsi, le réseau superposé a les propriétés d'un graphe aléatoire. Il est résistant aux départs et aux arrivées des clients ainsi qu'aux coupures réseau. Nous supposons qu'il n'y a pas de clients malicieux dans la fédération.

Définition 2 (Fédération de consommateurs de données)**.** *$F = \{C_1, ..., C_n\}$ une fédération de n consommateurs de données, où chaque consommateur de données a un identifiant unique C_i.*

Dans Ladda un consommateur de données exécute une seule requête à la fois. L'exécution en parallèle de requêtes sur un même client pourrait ralentir leur exécution à cause de la bande passante. En effet, une requête SPARQL est décomposée en plusieurs appels HTTP, ce nombre peut varier d'une dizaine à plusieurs milliers d'appels (cf. chapitre 2 p. 17). La bande passante peut alors être un goulot d'étranglement si plusieurs requêtes générant beaucoup d'appels HTTPs sont exécutées en parallèle sur un seul et même client. C'est pourquoi, nous limitons un consommateur de données à l'exécution d'une seule requête à la fois en local.

Définition 3 (Consommateur de données)**.** *Un consommateur de données C_i est un tuple $\langle S, K, W \rangle$ où :*

- *$S \in \{$*libre, occupé*$\}$ est le statut de C_i ;*
- *K est l'ensemble des voisins de C_i ;*
- *W est une file de requêtes attendant d'être exécutées.*

Lorsqu'un consommateur de données exécute l'une de ses requêtes ou celle d'un voisin localement, son statut devient *occupé*, sinon il est *libre*. L'ensemble des requêtes exécutées par un consommateur de données est un flux infini non continu de requêtes qui arrivent à n'importe quel moment. Les requêtes entrantes sont ajoutées à la file W du consommateur de données.

Définition 4 (Requête)**.** *Les requêtes sont exprimées de manière abstraite comme des tuples $Q_i = \langle C_i, q, at, et, rt \rangle$ tels que :*

- *$C_i \in F$ est l'identifiant du consommateur de données où Q_i est arrivée ;*
- *q est la description de la requête à exécuter ;*
- *at est le temps d'arrivée de Q_i ;*
- *et est le temps d'exécution de Q_i ;*
- *rt est le temps de réponse de Q_i, où $Q_i.rt \geq Q_i.at + Q_i.et$.*

Un consommateur de données ne connait pas à l'avance ni le temps d'arrivée des requêtes ni leur temps d'exécution. Il peut *déléguer* l'exécution de ses requêtes en attente d'exécution (W) à ses voisins (K). La *délégation* consiste à allouer une requête à l'un de ses voisins. Si la délégation échoue, la requête est replanifiée.

Définition 5 (Allocation)**.** *Une allocation est une opération* allouer*(Q_i, C_i), où :*

- *Q_i est la requête à exécuter ;*
- *C_i est le consommateur de données qui doit exécuter la requête.*

Un consommateur de données peut s'allouer une requête ou l'allouer à l'un de ses voisins. Si l'on reprend l'exemple du tableau 4.1, C_1 alloue ses requêtes dans la solution s_2 de la manière suivante : $C_1.allouer(Q_1, C_1)$, $C_1.allouer(Q_2, C_1)$ et $C_1.allouer(Q_3, C_3)$. C_2 alloue ses requêtes comme suit : $C_2.allouer(Q_4, C_2)$ et $C_2.allouer(Q_5, C_2)$ et C_3 exécute sa requête localement : $C_3.allouer(Q_6, C_3)$.

Énoncé du problème Étant donnée une fédération de consommateurs de données F et un ensemble de requêtes distribuées à travers des consommateurs de données, comment minimiser le temps de réponse des requêtes pour les consommateurs de données. Formellement :

$$\forall C_i \in F \wedge \forall Q_i \in C_i.W \mid \Delta = (Q_i.rt - Q_i.at) \textit{ est minimisé.}$$

Δ est le temps écoulé entre le temps de réponse ($Q.rt$) et le temps d'arrivée ($Q.at$) d'une requête. Dans le tableau 4.1 (p. 57), s_2 est une solution au problème de délégation.

4.3 Approche de Ladda

Ladda suit une allocation aléatoire des requêtes aux voisins. Cette stratégie a deux avantages : (i) théoriquement, une allocation aléatoire de requêtes sur le réseau entier équilibre la charge sur les consommateurs de données [39], (ii) elle évite la collecte d'informations sur l'état des voisins et réduit ainsi le coût de la délégation.

L'allocation aléatoire permet le parallélisme inter-requêtes. Cependant, comme les consommateurs de données n'hébergent pas les données et accèdent au même serveur TPF, celui-ci peut devenir un goulot d'étranglement pour la parallélisation. Il existe donc un seuil au-delà duquel, la parallélisation n'améliore plus les performances.

4.3.1 Limites de parallélisation

Nous cherchons à déterminer le seuil à partir duquel la parallelisation des requêtes n'améliore pas les performances. Pour cela, nous exécutons un même ensemble de requêtes distribuées sur 1, 21 et 50 clients.

Nous avons extrait 1509 requêtes du journal de DBpedia 3.8 (cf. section 4.4 p. 64). Les données de DBpedia 3.8 sont hébergées sur un serveur TPF. Le serveur est un HPC DELL R720 avec 40 processeurs virtuels, 130 Go de mémoire et un serveur de cache Web.

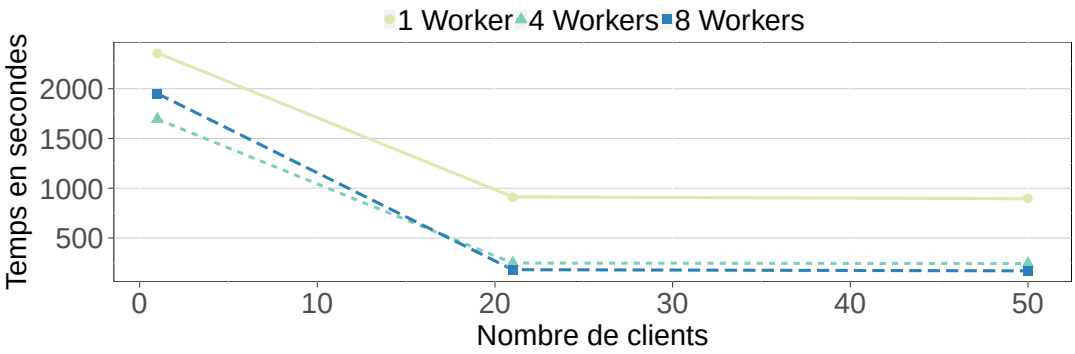


FIGURE 4.2 – Temps d'exécution de 1509 requêtes du journal de DBpedia 3.8, allouées de façon statique à 1, 21 et 50 clients TPF. Le serveur TPF est configuré avec 1, 4 et 8 sous-processus (*workers*).

Pour allouer les requêtes aux clients de manière optimale, nous appliquons l'algorithme *Longest Processing Time* (LPT) [47]. L'algorithme LPT s'exécute en deux étapes :

1. dans la première étape, les requêtes sont triées dans l'ordre décroissant selon leur temps d'exécution ;

2. dans la deuxième étape, en suivant l'ordre de la liste de la première étape, la requête en haut de la pile est sélectionnée et allouée au client le moins chargé. La charge d'un client est définie par la somme des temps d'exécution des requêtes qui lui sont déjà allouées $charge = \sum_{Q_i \in C_i.W} Q_i.et$.

Exemple 4. *Considérons trois clients C_1 à C_3 et les six requêtes suivantes, avec un temps d'exécution exprimé en secondes : $W = \{Q_1{:}10, Q_2{:}23, Q_3{:}5, Q_4{:}8, Q_5{:}12, Q_6{:}30\}$. Dans sa première étape LPT va trier les requêtes dans l'ordre décroissant selon leur temps d'exécution : $W = \{Q_6{:}30, Q_2{:}23, Q_5{:}12, Q_1{:}10, Q_4{:}8, Q_3{:}5\}$. Puis pour chaque requête il va la placer sur le client le moins chargé :*

- *On commence avec la requête $\{Q_6{:}30\}$, aucun des clients n'est encore chargé, la requête est donc allouée à C_1, $C_1 = \{Q_6{:}30\}, C_2 = \{\}, C_3 = \{\}$.*
- *Puis la requête $\{Q_2{:}23\}$ est allouée au client C_2, $C_1 = \{Q_6{:}30\}, C_2 = \{Q_2{:}23\}, C_3 = \{\}$.*
- *La requête $\{Q_5{:}12\}$ est allouée au client C_3, $C_1 = \{Q_6{:}30\}, C_2 = \{Q_2{:}23\}, C_3 = \{Q_5{:}12\}$.*
- *A la requête $\{Q_1{:}10\}$ tous les clients ont déjà une requête allouée, mais si l'on regarde au niveau du temps d'exécution des requêtes, c'est le client C_3 qui est le moins chargé, la requête lui est donc allouée, $C_1 = \{Q_6{:}30\}, C_2 = \{Q_2{:}23\}, C_3 = \{Q_5{:}12, Q_1{:}10\}$.*
- *Une fois toutes les requêtes allouées on obtient la distribution suivante : $C_1 = \{Q_6{:}30\}, C_2 = \{Q_2{:}23, Q_3{:}5\}, C_3 = \{Q_5{:}12, Q_1{:}10, Q_4{:}8\}$.*

Les 1509 requêtes sont exécutées deux fois : (i) la première exécution permet d'initialiser le cache Web ; (ii) la deuxième exécution sert à mesurer les temps d'exécution de chaque requête.

La figure 4.2 montre les résultats de l'expérience avec plusieurs configurations de sous-processus (*worker*) du serveur TPF. L'ajout de sous-processus permet de mieux traiter l'exécution en parallèle de requêtes TPF.

Dans un premier temps, la parallélisation avec 21 clients réduit significativement le temps de réponse de l'ensemble des requêtes comparé à une exécution séquentielle, c'est-à-dire avec un client. La parallélisation réduit le temps libre du serveur TPF et rend le cache Web plus efficace comme observé dans [61] (cf. expérience 4.4.2.5 p. 73). Mais la parallélisation de l'ensemble des requêtes sur 50 clients ne réduit pas de façon significative le temps global d'exécution des requêtes, comparé à la parallélisation sur 21 clients. De plus, augmenter le nombre de sous-processus sur le serveur TPF améliore peu le temps de réponse. Par conséquent, dans cette configuration, paralléliser sur au plus 21 voisins est suffisant.

4.3.2 Est-ce que la localité est importante ?

Chaque client a un cache local de données, rempli selon les requêtes exécutées localement par ce client. L'exécution de requêtes extérieures peut potentiellement réduire l'efficacité de ce cache local. Les voisins doivent-ils être choisis selon leur cache local comme dans [24] ou

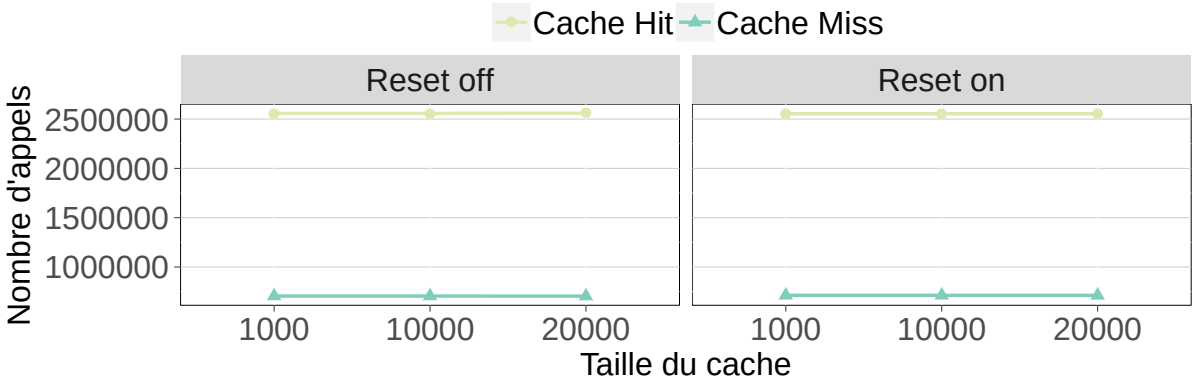


FIGURE 4.3 – Un client TPF exécutant des requêtes DBpedia, avec un cache local effacé (*reset on*) ou non (*reset off*) entre l'exécution de requêtes. Les *cache hit* correspondent aux appels résolus dans le cache local, tandis que les *cache miss* correspondent aux appels résolus sur le serveur TPF.

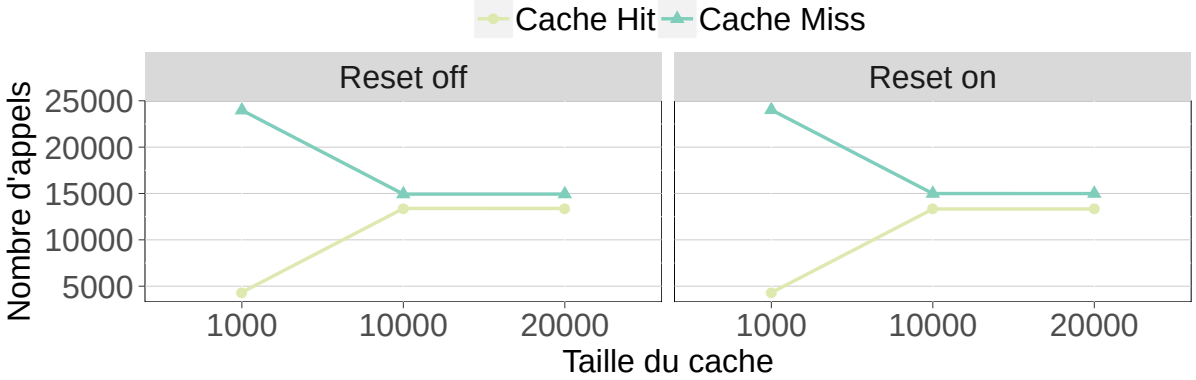


FIGURE 4.4 – Un client TPF exécutant des requêtes BSBM, avec un cache local effacé (*reset on*) ou non (*reset off*) entre l'exécution de requêtes. Les *cache hit* correspondent aux appels résolus dans le cache local, tandis que les *cache miss* correspondent aux appels résolus sur le serveur TPF.

de façon aléatoire ? Pour répondre à cette question, nous effectuons deux expériences pour vérifier la réutilisation du cache local entre l'exécution des requêtes.

Dans la première expérience, un client TPF exécute 1509 requêtes de DBpedia sans effacement du cache (*reset off*) entre l'exécution des requêtes. Puis, il exécute à nouveau les mêmes requêtes avec un effacement du cache local (*reset on*) entre l'exécution des requêtes. La figure 4.3 montre qu'effacer le cache local entre les requêtes n'a pas d'impact sur le nombre d'appels pris en charge localement, même si la taille du cache local varie. Le cache local est principalement utilisé durant l'exécution d'une même requête, mais pas d'une requête à l'autre.

Pour s'assurer que ce résultat n'est pas dépendant du jeu de données et des requêtes, nous effectuons une deuxième expérience avec le jeu de données Berlin Benchmark (BSBM) [11], qui simule une application Web. Les données et les requêtes sont moins diverses que dans DBpedia. Les données du cache local ont donc plus de chance d'être réutilisées d'une requête à l'autre.

Dans cette expérience un client TPF exécute un ensemble de 25 requêtes BSBM avec

effacement du cache (*reset on*) et sans (*reset off*), entre l'exécution des requêtes. La figure 4.4 présente les résultats de l'expérience. Les résultats corroborent ceux obtenus avec DBpedia.

D'après les résultats obtenus, les voisins d'un consommateur de données peuvent être choisis de façon aléatoire. La délégation de requêtes ne détériore pas l'efficacité du cache local.

4.3.3 Algorithmes

Algorithme 2: ChoisirVoisins($K = \{K_1, ..., K_i\}$: voisins, $B = \{B_1, ..., B_j\}$: voisins *occupés*, m : nombre de destinations)

```
1 ChoisirVoisins(K, B, m):
2     S = {k | k ∈ K ∧ k ∉ B}
3     D = random(S, m)
4     return D
```

L'algorithme 2 détaille le processus de sélection des voisins pour déléguer des requêtes. Un consommateur de données a un ensemble de voisins K. De plus, il maintient un ensemble de voisins B à qui il a déjà délégué des requêtes et dont il attend les réponses. Il sait donc que les voisins dans l'ensemble B sont *occupés*. Tandis que le statut des voisins restant S est inconnu du consommateur de données. Il choisit m voisins aléatoirement dans l'ensemble S à contacter pour déléguer ses requêtes (lignes 2 – 3).

Algorithme 3: Ladda(Q : ensemble de requêtes entrantes, C : consommateur de données, $K = \{K_1, ..., K_i\}$: voisins, $B = \{B_1, ..., B_j\}$: voisins *occupés*, m : nombre de destinations, $W = \{W_1, ..., W_\}$: requêtes en attente, t : timeout pour une requête

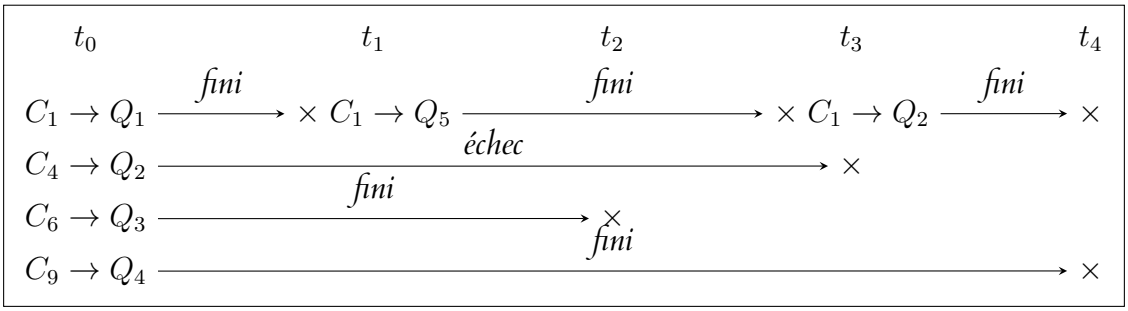
```
1  exReq(C, K, B, m, W, t):
2      if taille(W) > 0 then
3          if Libre(C) then
4              allouer(C, défiler(W))
5          if taille(W) > 0 then
6              D=ChoisirVoisins(K, B, m)
7              foreach d ∈ D do
8                  if taille(W) > 0 then
9                      allouer(d, défiler(W))
10                     setTimeout(t, query)
11                     B = B ∪ {d}
12 ↯ onRequête(requête, e):
13     if Libre(C) ∧ vide(W) then
14         allouer(C, requête)
15     else
16         émettreÉchec(e, requête)
17 ↯ onArrivée(Q, C):
18     W = W ∪ Q ; exReq(C, K, B, m, W, t)
19 ↯ onÉchec(d, query):
20     B = B − {d}
21     if Libre(C) then
22         exReq(C, K, B, m, W, t)
23 ↯ onRésultats(d, requête, résultats):
24     if requête ∈ W then
25         B = B − {d} ; W = W − {requête}
26         traiterRésultats(requête, résultats)
27         if taille(W) > 0 then
28             exReq(C, K, B, m, W, t)
29     else
30         émettreRésultats(d, requête, résultats)
31 ↯ onTimeout(requête):
32     if requête ∈ W then
33         exReq(c, K, B, m, W, t)
```

L'algorithme 3 détaille la délégation de requêtes. Quand une requête arrive sur le consommateur de données C, il l'ajoute dans sa file de requêtes en attente W et l'ordonnanceur essaye

d'allouer la requête (ligne 18). Dans un premier temps, si le consommateur de données est *libre* la première requête est allouée localement (lignes 3 – 4). Ensuite, s'il reste des requêtes en $attente$, C sélectionne m voisins aléatoirement pour la délégation de requêtes (ligne 6) et pour tous les voisins choisis dans D, il alloue une requête en $attente$ à exécuter (lignes 7 – 9). Si la délégation échoue, la requête est replanifiée (lignes 19 – 22). Sinon, si la délégation réussit, le consommateur de données traite les résultats et s'il y a toujours des requêtes en $attente$ il essaye de les allouer (lignes 27 – 28). Pour chaque requête déléguée, un *timeout* est mis en place (ligne 10). Quand le *timeout* est atteint, si C n'a pas encore reçu de réponse, il replanifie la requête (lignes 31 – 33). Cela permet de faire face aux possibles pertes de messages ou départ d'un voisin. Le consommateur de données C peut exécuter des requêtes pour ses voisins, seulement s'il est *libre* et qu'il n'a pas de requêtes en $attente$ (lignes 12 – 16).

Les requêtes d'un consommateur de données sont prioritaires par rapport aux requêtes de ses voisins. Un client ayant des requêtes en attente est considéré comme *occupé*. Ainsi, l'algorithme empêche la famine des consommateurs de données.

Exemple 5. *Supposons une fédération de 10 consommateurs de données C_1 à C_{10}, où chaque consommateur de données a trois voisins, $taille(K) = 3$. Le consommateur de données C_1 a un ensemble de 5 requêtes, $C_1.W = [Q_1, ..., Q_5]$ et les voisins suivants : C_4, C_6 et C_9.*



Au temps t_0, *C_1 alloue ses requêtes comme suit : $(Q_1 \to C_1)$, $(Q_2 \to C_4)$, $(Q_3 \to C_6)$ et $(Q_4 \to C_9)$, sa liste de voisins* occupés *est la suivante : $C_1.B = [C_4, C_6, C_9]$.*

Au temps t_1, *C_1 a fini l'exécution de Q_1, C_1 devient* libre *et a seulement une requête en attente, $C_1.W = [Q_5]$. C_1 exécute donc Q_5 : $(Q_5 \to C_1)$.*

Au temps t_2, *l'exécution de Q_3 est terminée. Comme toutes les requêtes sont déjà allouées, il ne se passe rien.*

Au temps t_3, *la délégation de Q_2 échoue et l'exécution de Q_5 est terminé. C_1 est* libre*, donc il exécute la requête Q_2.*

Au temps t_4, *Q_2 et Q_4 sont finies.*

4.3.4 Coût de Ladda

Le coût de Ladda correspond à la communication entre les clients pour maintenir le réseau et effectuer les délégations. Pour maintenir les consommateurs de données connectés, chaque client effectue un échange périodique de voisins (*shuffle*) avec son voisin le plus vieux [64]. Cette procédure est effectuée en temps constant. Par conséquent, un voisin interrompt un de ses voisins périodiquement, mais il est aussi interrompu périodiquement par ses voisins.

Un échange de table de voisins toutes les 30 secondes a un impact négligeable sur les performances. Bien évidemment, un échange des tables toutes les secondes dégrade de façon significative les performances.

La délégation peut également interrompre les clients. Dans Ladda, un consommateur de données délègue par paquet de m requêtes. À chaque fois qu'il essaye d'allouer ses requêtes en attente il interrompt au plus m de ses voisins. Si la délégation réussit le coût de la délégation est compensé par le parallélisme, sinon il est perdu. Les pires cas pour Ladda sont les suivants :

1. Si tous les consommateurs de données dans la fédération sont *occupés*, ils vont interrompre leurs voisins en vain.
2. Si un consommateur C_j accepte d'exécuter une requête complexe Q_i pour C_i, il risque de retarder l'exécution de ses propres requêtes si tous les autres clients sont *occupés*. Par exemple, si une requête de C_j, Q_j, arrive pendant que C_j est entrain d'exécuter Q_i et qu'aucun de ses m voisins est *libre* pour exécuter Q_j l'exécution de Q_j sera retardée.

Ces cas sont peu probables si la proportion de consommateurs de données disponibles est élevée, comme supposé dans la section 4.1 (p. 56). Néanmoins, Ladda a été testé avec des clients fortement chargés dans les expérimentations, comme on peut le voir dans la section suivante.

4.4 Étude expérimentale

Dans le cadre de ces expériences, nous voulons vérifier si le mécanisme de délégation permet de réduire le temps global d'exécution de l'ensemble des requêtes.

4.4.1 Environnement d'expérimentation

Dans cette section nous décrivons l'environnement, les configurations d'expérimentation et les métriques observées, récapitulés dans le tableau 4.2.

Jeu de données et requêtes Nous utilisons le jeu de données DBpedia 3.8, qui comprend les données en anglais seulement. Les 1509 requêtes des 50 premiers clients sont extraites d'un journal réel de DBpedia 3.8, entre le 26 août 2012 à 05:00:00 et le 26 août à 06:00:00[2].

Implémentation Nous étendons le client TPF [3] avec les algorithmes décrits en section 4.3.3 (p. 62). Le code et des résultats complémentaires sont disponibles à : `https://github.com/pfolz/Ladda`.

Environnement L'environnement d'expérimentation est composé :

- d'un serveur TPF exposant le jeu de données DBpedia 3.8 au format HDT, configuré avec 4 sous-processus (*workers*) ;

[2] Le log des requêtes est disponible à `ftp://download.openlinksw.com/support/dbpedia/`.
[3] `https://github.com/LinkedDataFragments/Client.js`

Paramètre	Valeurs
Nombre de clients	50
Vue RPS	21
Cache Local	1000
Temps de *shuffle*	30s
Jeu de données	DBpedia 3.8
Requêtes	1 509 d'un log réel
Configurations	*Tous chargés, 1/2 chargés, 1/4 chargés* et *1 chargé*
Approches	*Ladda 2, Ladda K* et *Sans délégation*

TABLE 4.2 – Récapitulatif des paramètres d'expérimentation pour Ladda.

- d'un serveur Web de mise en cache, NGINX. Il est configuré avec un cache de 1 Go ;
- et de 50 clients.

Un serveur HPC exécute le serveur TPF, le serveur de cache Web et les clients. Le serveur HPC a la configuration suivante : 40 processeurs, 130 Go de mémoire, tournant sur Debian 7.8.

La fédération contient 50 clients Ladda-TPF, chacun ayant 21 voisins. L'échange des voisins s'effectue toutes les 30 secondes. Dans ces expériences nous supposons qu'il n'y a pas de pertes de messages, ni de clients qui quittent la fédération pendant l'expérience. Le *timeout*, décrit en section 4.3.3 (p. 62), est donc désactivé. Toutes les expériences se déroulent en deux étapes. La première étape permet d'initialiser le réseau et le serveur de cache Web. La deuxième étape est lancée après une barrière de synchronisation. Cette étape est dédiée aux mesures des expériences. Dans les deux étapes, chaque client exécute le même ensemble de requêtes.

Configuration Dans le journal de DBpedia 3.8, les requêtes sont estampillées selon l'heure à laquelle elles ont été exécutées, leur temps d'arrivée précis est inconnu. Dans ces expériences nous considérons le pire des cas, c'est-à-dire que nous supposons que les requêtes ont toutes le même temps d'arrivée. Les requêtes sont ensuite distribuées sur les 50 clients TPF selon différentes stratégies :

Tous chargés : les requêtes sont regroupées suivant leurs adresses IP et chaque client est responsable des requêtes qui ont été évaluées par la même adresse. Cette stratégie est considérée comme le pire des cas pour Ladda. En effet, au début tous les clients sont *occupés*, donc les premières délégations à des voisins échouent.

1/2 chargés : les requêtes sont distribuées sur 25 clients et les 25 clients restants n'ont pas de requêtes à exécuter, ils sont *libres*. Cette stratégie est considérée comme le cas moyen pour Ladda. La moitié des clients est *libre* au début de l'expérience.

1/4 chargés : les requêtes sont distribuées sur 12 clients et 38 clients n'ont pas de requêtes à exécuter.

1 chargé : les requêtes sont exécutées par un seul client. Comme c'est le seul client à être *occupé* dans la fédération, toutes les délégations réussissent. C'est le meilleur cas pour Ladda.

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
Tous chargés	Q_{1_1}	$Q_{2_{1-5}}$	$Q_{3_{1-5}}$	Q_{4_1}	Q_{5_1}	$Q_{6_{1-12}}$	$Q_{7_{1-2}}$	$Q_{8_{1-7}}$	$Q_{9_{1-3}}$	Q_{10_1}
1/2 chargés		Q_{1_1} $Q_{2_{1-5}}$		$Q_{3_{1-5}}$ Q_{4_1}		Q_{5_1} $Q_{6_{1-12}}$		$Q_{7_{1-2}}$ $Q_{8_{1-7}}$		$Q_{9_{1-3}}$ Q_{10_1}
1/4 chargés			Q_{1_1} $Q_{2_{1-5}}$ $Q_{3_{1-5}}$			Q_{4_1} Q_{5_1} $Q_{6_{1-12}}$				$Q_{7_{1-2}}$ $Q_{8_{1-7}}$ $Q_{9_{1-3}}$ Q_{10_1}
1 chargé	Q_{1_1} $Q_{2_{1-5}}$ $Q_{3_{1-5}}$ Q_{4_1} Q_{5_1} $Q_{6_{1-12}}$ $Q_{7_{1-2}}$ $Q_{8_{1-7}}$ $Q_{9_{1-3}}$ Q_{10_1}									

TABLE 4.3 – Distribution de requêtes selon les stratégies : *Tous chargés, 1/2 chargés, 1/4 chargés* et *1 chargé*.

Exemple 6. *Considérons une fédération de dix clients C_1 à C_{10} et les requêtes Q_{i_j} suivantes, où i est l'identifiant d'un client et j l'identifiant de la requête : $W = \{Q_{1_1}, Q_{2_{1-5}}, Q_{3_{1-5}}, Q_{4_1}, Q_{5_1}, Q_{6_{1-12}}, Q_{7_{1-2}}, Q_{8_{1-7}}, Q_{9_{1-3}}, Q_{10_1}\}$. Le tableau 4.3 présente la distribution des requêtes selon les différentes stratégies.*

Enfin, nous faisons varier le nombre de voisins m contactés lors de l'allocation de ses requêtes. En effet, un client peut utiliser tous ses voisins ou seulement une partie ; donc $m \leq |K|$. Ladda est nommé *Ladda 2* (L2) quand $m = 2$ et *Ladda K* (LK) quand $m = |K|$. *Ladda 2* réduit le coût de la délégation mais peut manquer quelques voisins *libres*. *Ladda K* trouve les voisins *libres* mais augmente le coût de la délégation. Nous comparons *Ladda 2* et *Ladda K* avec une approche sans délégation : *Sans Délégation* (SD).

Métriques Au cours des expériences nous calculons les métriques suivantes :

Temps local d'exécution : temps qu'un client met pour exécuter l'ensemble de ses requêtes, qu'elles soient déléguées ou non. Ce temps ne comprend pas l'exécution des requêtes qu'il a exécutées pour ses voisins.

Temps global d'exécution : temps d'exécution de toutes les requêtes de la fédération, soit le temps d'exécution local du client le plus long.

Débit : nombre de requêtes terminées par unité de temps (ex : par minute).

4.4.2 Résultats

Cette section présente l'ensemble des résultats des expérimentations.

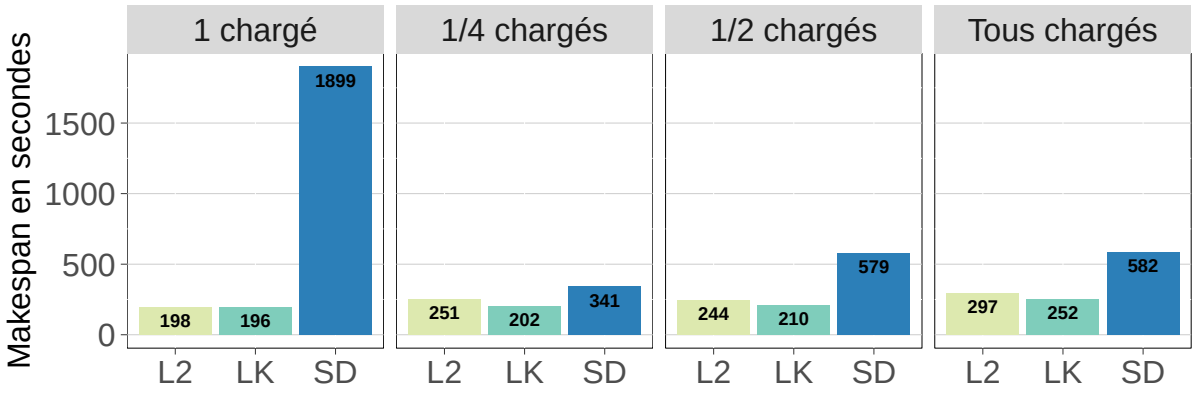


FIGURE 4.5 – Temps global d'exécution (*makespan*) pour *Ladda 2* (L2), *Ladda K* (LK) et l'approche de référence *Sans délégation* (SD) dans les configurations *1 chargé*, *1/4 chargés*, *1/2 chargés* et *Tous chargés*.

4.4.2.1 Temps global d'exécution

La délégation de requêtes permet à un consommateur de données de parallaléliser ses requêtes. À un instant t, il y a donc plus de requêtes qui s'exécutent en parallèle sur le *même* serveur TPF. De part sa nature, la délégation sollicite donc plus le serveur. Si le serveur est trop chargé cela peut dégrader le temps global d'exécution des requêtes par rapport à une exécution séquentielle.

Objectif : Est-ce que la délégation de requêtes réduit le temps global d'exécution ?

Description : Les 1509 requêtes extraites du journal de DBpedia 3.8, correspondant aux 50 premiers clients d'une heure, sont distribuées sur les clients TPF selon les configurations présentées précédemment. Les voisins sont choisis de manière aléatoire lors de la délégation de requêtes, les résultats peuvent varier d'une expérience à l'autre. C'est pourquoi nous exécutons l'expérience trois fois et faisons la moyenne sur ces trois expériences.

Résultats : La figure 4.5 montre la moyenne sur trois expériences du temps global d'exécution de la fédération avec *Ladda 2*, *Ladda K* et l'approche de référence *Sans Délégation* pour les quatre configurations : *1 chargé*, *1/4 chargés*, *1/2 chargés* et *Tous chargés*. Dans toutes les configurations, Ladda est capable de réduire le temps global d'exécution.

Explication : Dans la configuration *1 chargé*, l'approche *Sans Délégation* a un temps global d'exécution très élevé car les 1509 requêtes sont exécutées de manière séquentielle sur le même client. *Ladda 2* et *K* sont dans ce cas très efficaces car toutes les délégations réussissent.

Dans les configurations *1/4 chargés* , *1/2 chargés* et *Tous chargés*, l'ensemble des requêtes est distribué sur les clients, les requêtes sont donc naturellement parallélisées. Par conséquent, le temps global d'exécution de l'approche *Sans Délégation* est grandement réduit, comparé au temps global d'exécution de la configuration *1 chargé*. Malgré tout, Ladda arrive encore à réduire le temps d'exécution. Au début de l'expérience, beaucoup de délégations échouent car de nombreux clients sont *occupés*. Cependant, dès qu'un client a terminé ses requêtes

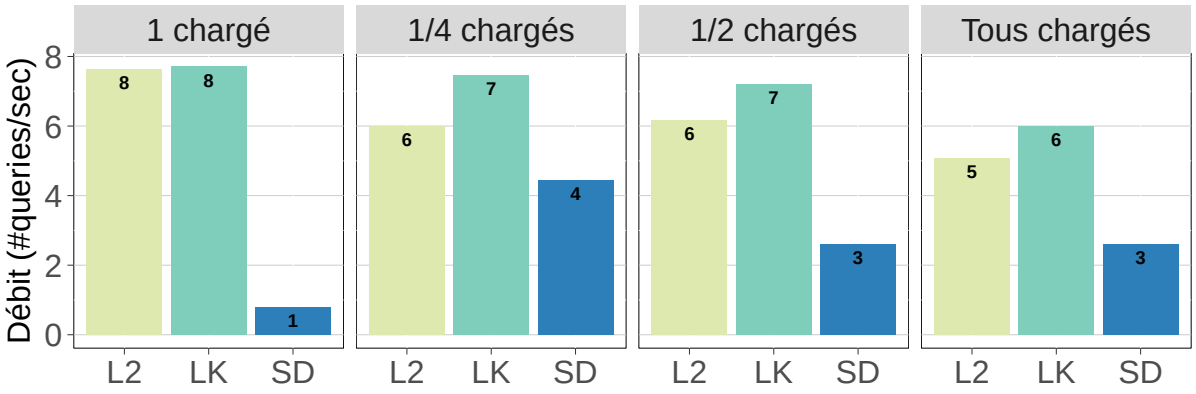


FIGURE 4.6 – Le nombre de requêtes exécutées par seconde (*throughput*) pour *Ladda 2* (L2) , *Ladda K* (LK) et l'approche de référence *Sans délégation* (SD) dans les configurations *1 chargé*, 1/4 *chargés*, 1/2 *chargés* et *Tous chargés*.

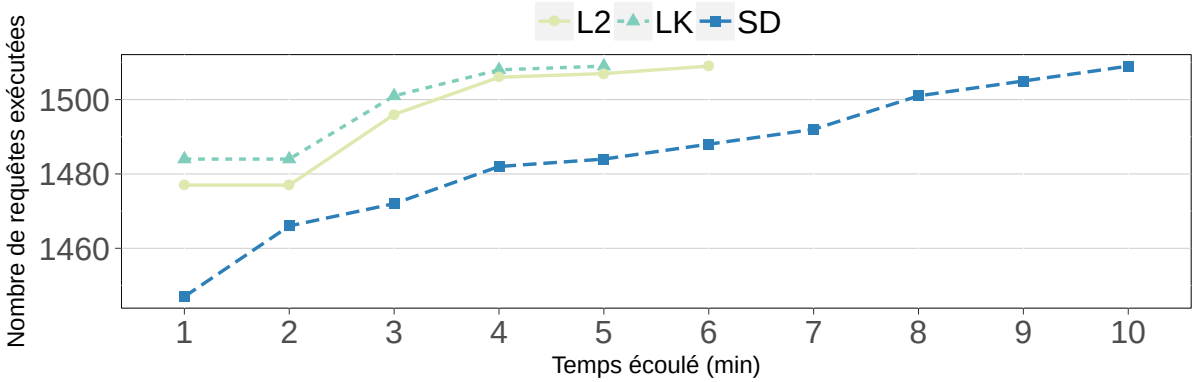


FIGURE 4.7 – Le nombre de requêtes exécutées par minute par *Ladda 2* (L2) , *Ladda K* (LK) et l'approche de référence *Sans délégation* (SD) dans la configuration *Tous chargés*.

locales, il accepte les délégations des clients n'ayant pas encore fini. Ladda devient donc efficace quand le nombre de clients libres augmente au cours de l'expérience.

Dans toutes les configurations, on peut observer que *Ladda K* est toujours meilleur que *Ladda 2*. *Ladda K* est plus efficace pour trouver les voisins *libres*, ce qui permet de commencer l'exécution des requêtes aussi vite que possible et garde les clients occupés. Le coût généré par *Ladda 2* est plus faible, mais il peut rater des opportunités de délégation. On peut donc en conclure que commencer l'exécution des requêtes aussi vite que possible est crucial.

4.4.2.2 Débit

L'expérience précédente nous montre que Ladda est capable de réduire le temps global d'exécution de la fédération, peu importe la configuration. Nous souhaitons maintenant savoir si, globalement, le débit de requêtes par minute est meilleur avec Ladda.

Objectif : Est-ce que la délégation de requêtes augmente le nombre de requêtes exécutées par seconde ?

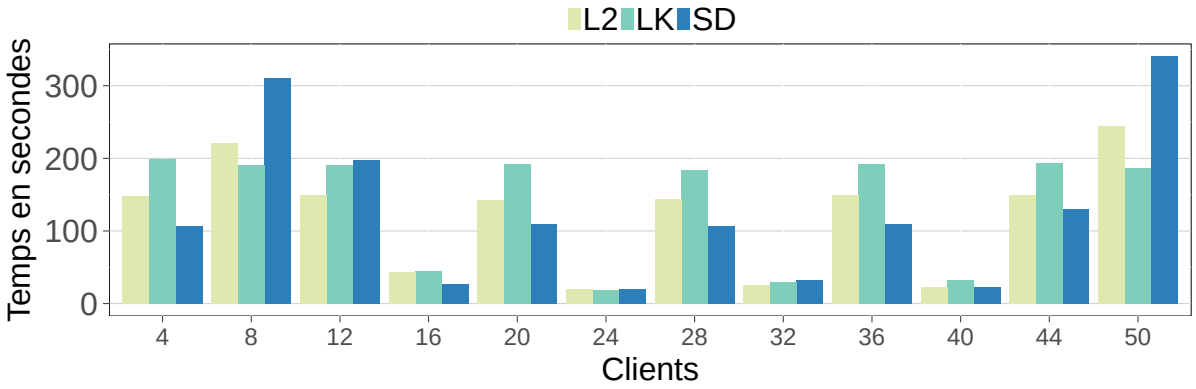


FIGURE 4.8 – Temps de réponse par client avec *Ladda 2* (L2), *Ladda K* (LK) et *SD* (Sans délégation) pour la configuration 1/4 *chargés* : 12 clients avec des IDs discontinus.

Résultat : La figure 4.6 montre la moyenne sur trois expériences du débit de requêtes exécutées par seconde avec *Ladda 2*, *Ladda K* et l'approche de référence *Sans Délégation* pour les quatre configurations : *1 chargé*, 1/4 *chargés*, 1/2 *chargés* et *Tous chargés*. Dans toutes les configurations, Ladda a un meilleur débit de requêtes exécutées par seconde.

La figure 4.7 décrit le débit de requêtes par minute jusqu'à l'exécution complète de l'ensemble des requêtes pour la configuration *Tous chargés*. Les figures des autres configurations sont disponibles en annexe A (p. 81). Comparée aux figures précédentes, on peut voir le comportement du système minute par minute.

Explication : La configuration *Tous chargé* est un véritable défi pour Ladda. Quand l'expérience commence, tous les clients sont *occupés* et les délégations échouent. Cependant, après une minute d'exécution, on observe dans la figure 4.7 que l'approche *Sans Délégation* a exécuté moins de 1450 requêtes et que quelques clients sont probablement *libres*. Comme nous l'avons vu dans la figure 4.1 (p. 56), beaucoup de clients ont seulement quelques requêtes et peuvent terminer très rapidement. *Ladda 2* et *K* profitent de ces clients disponibles afin de rééquilibrer la charge. *Ladda 2* et *K* ont exécuté $\approx$ 1480 requêtes après une minute. Ils allouent ensuite les 120 requêtes restantes au 50 clients. Au même moment, l'approche *Sans Délégation* concentre ses requêtes sur quelques clients. Cela explique pourquoi Ladda, en général, finit si rapidement. Si nous comparons *Ladda 2* et *K*, *Ladda K* parallélise plus de requêtes que *Ladda 2* ; ce qui lui permet de finir, pendant les premières minutes, l'exécution des requêtes plus rapidement que *Ladda 2*.

4.4.2.3 Temps local d'exécution par client

Au travers des expériences précédentes, on peut observer que Ladda a un meilleur temps global d'exécution et un meilleur débit de requêtes par seconde dans toutes les configurations. L'expérience suivante a pour objectif d'observer l'impact de la délégation, client par client.

Objectif : Est-ce que la délégation impacte le temps de réponse par client ?

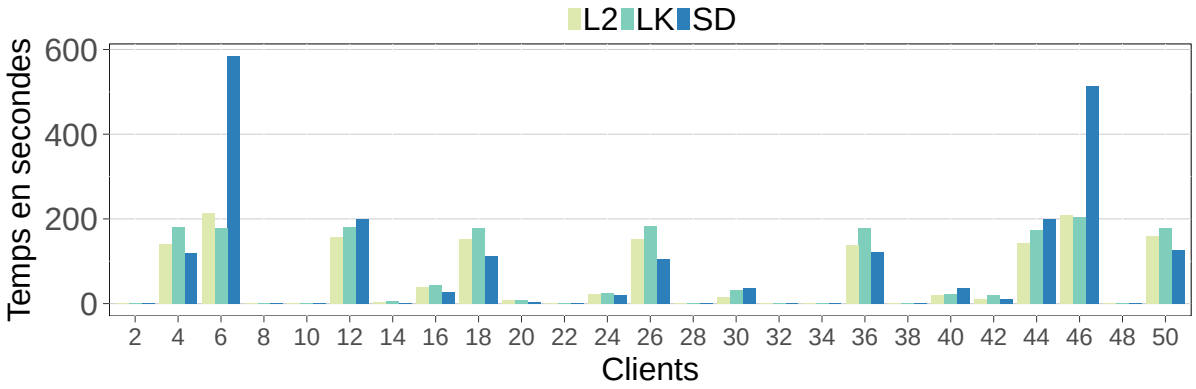


FIGURE 4.9 – Temps de réponse par client avec *Ladda 2* (L2), *Ladda K* (LK) et *SD* (Sans délégation) pour la configuration 1/2 *chargés* : 25 clients avec des IDs discontinus.

Résultats : La figure 4.8 représente les temps local d'exécution par client pour *Ladda 2*, *Ladda K* et *Sans Délégation* dans la configuration 1/4 *chargés*. Dans cette configuration, seulement 12 clients ont des requêtes à exécuter au début de l'expérience. Les mesures comprennent le temps local d'exécution de toutes les requêtes initialement affectées au client, c'est-à-dire que si un client a reçu une requête à exécuter via la délégation, il ne fait pas partie de cette figure. Pour les résultats de l'approche *Sans Délégation*, comme attendu, l'affectation des requêtes aux clients n'est pas optimal. Comparer le temps de résultats par client de Ladda et *Sans Délégation* révèle deux cas :

1. *Ladda 2* ou *K* finit avant *Sans Délégation*. Dans ce cas, cela signifie que pour ce client, Ladda a délégué des requêtes avec succès à d'autres clients et cela compense le coût de la délégation. C'est le cas pour le client 8 et le client 50, par exemple.

2. *Ladda 2* ou *K* finit après *Sans Délégation*. Dans ce cas, cela signifie que pour ce client, les délégations qui ont réussi ne compensent pas le coût de la délégation.

Dans les deux cas, si le temps local d'exécution est dégradé à cause de la délégation, la dégradation est limitée. Si la délégation réussit, cela peut être fortement profitable. Ce comportement est plus visible dans la figure 4.9, qui affiche le temps local d'exécution par client pour la configuration *1/2 chargés*. Comme les clients n'ont pas connaissance par avance des tâches qu'ils vont devoir exécuter, la délégation est une bonne stratégie.

Explication : Dans les figures 4.8 et 4.9, on peut observer que contrairement à ce qui a été établi dans l'expérience 4.4.2.1 (p. 67) *Ladda K* n'est pas toujours meilleure que *Ladda 2*. Globalement *Ladda K* permet de finir plus tôt, mais individuellement les clients sont perdants comparé à *Ladda 2*. Dans le cadre de *Ladda K* un client trouve plus rapidement des voisins *libres*, à un instant t il y a donc plus de requêtes qui s'exécutent en parallèle qu'avec *Ladda 2* et c'est alors le serveur TPF qui devient un goulot d'étranglement. Pour confirmer cette hypothèse nous regardons le temps local d'exécution par client en faisant varier le nombre de sous-processus (*workers*) sur le serveur ; plus il y a de sous-processus, plus le serveur peut traiter de requêtes en parallèle.

Pour cela nous reprenons la distribution statique des requêtes sur 21 clients, présentée dans la section 4.3.1 (p. 59). Les résultats de cette expérience sont présentés dans la

FIGURE 4.10 – Temps de réponse par client avec l'approche *Sans Délégation* et la distribution statique des requêtes, sur 21 clients.

figure 4.10 [4]. On peut constater que, à part pour le client 21, augmenter le nombre de sous-processus a un impact positif sur le temps local d'exécution par client. On peut également constater que passer de 4 sous-processus à 8 sous-processus réduit grandement le temps local d'exécution par client, mais ce n'est pas le cas quand on passe de 8 sous-processus à 16 sous-processus. Le serveur TPF est constitué d'un processus *maître* et de x sous-processus *esclaves*. Le processus *maître* répartit les appels HTTP sur les sous-processus *esclaves*, mais toutes les réponses des requêtes doivent transiter par le processus *maître* avant d'être délivrées aux clients, créant un goulot d'étranglement.

4.4.2.4 Distribution des requêtes déléguées

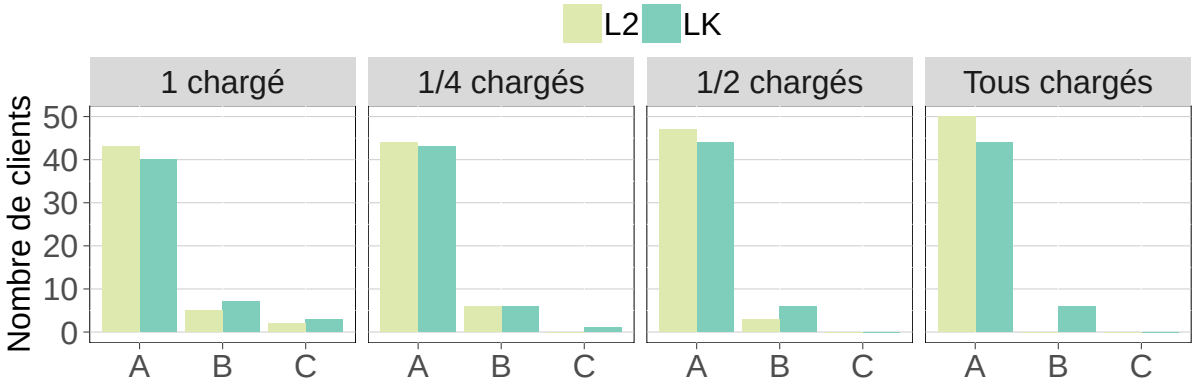


FIGURE 4.11 – Distribution des requêtes déléguées pour *Ladda 2* (L2) et *Ladda K* (LK), dans les configurations : *1 chargé*, *1/4 chargés*, *1/2 chargés* et *Tous chargés* ; où la catégorie A correspond aux clients ayant effectués moins de 60 requêtes déléguées, la catégorie B correspond aux clients ayant effectués entre 60 et 120 requêtes déléguées et la catégorie C correspond aux clients ayant effectués plus de 120 requêtes déléguées. Moyenne sur 3 exécutions.

Dans cette expérience nous souhaitons savoir si les requêtes déléguées sont bien réparties à travers les consommateurs de données de la fédération ou si quelques clients exécutent la majorité des requêtes déléguées.

[4] Les figures pour les autres configurations sont disponibles en annexe A p. 81)

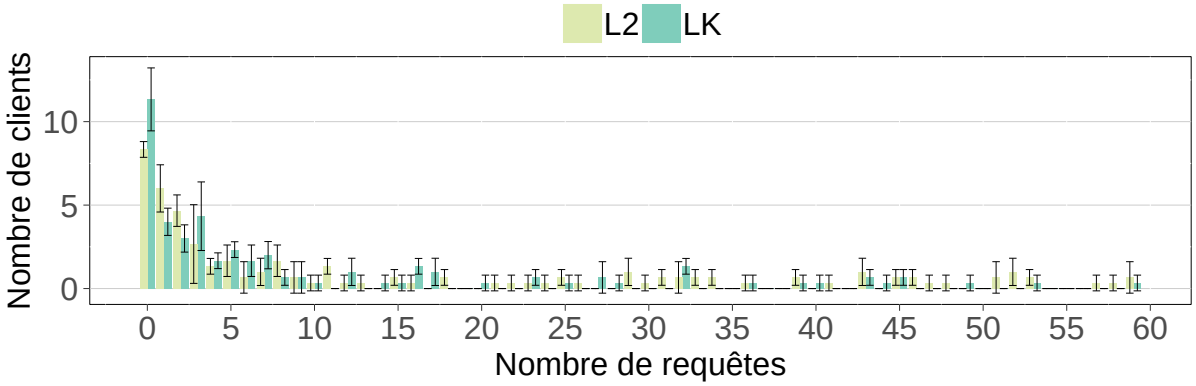


FIGURE 4.12 – Nombre de clients pour un nombre de requêtes exécutées donné, avec un écart type sur le nombre de clients pour la configuration *Tous chargés* et les approches : *Ladda 2* (L2) et *Ladda K* (LK).

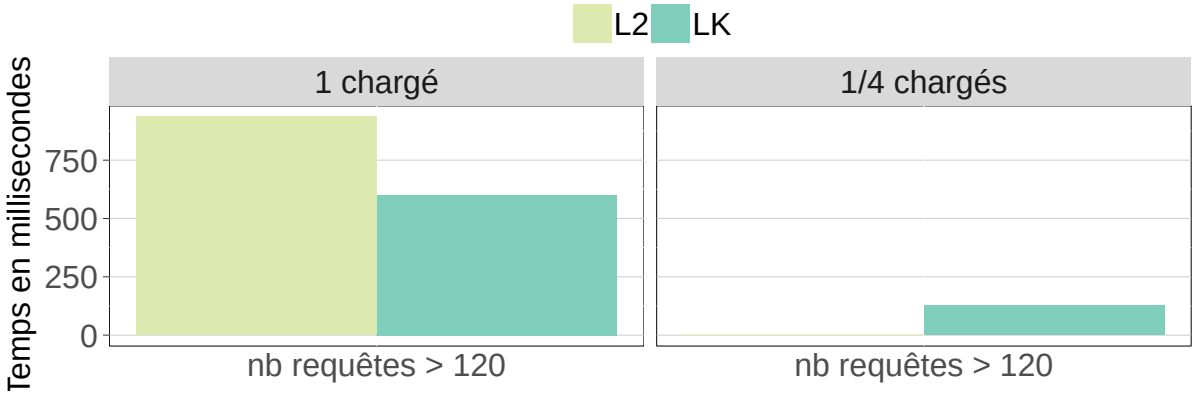


FIGURE 4.13 – Temps moyen en millisecondes d'une requête pour les clients ayant exécutés plus de 120 requêtes déléguées, pour les configurations : *1 chargé* et *1/4 chargés* et les approches *Ladda 2* (L2) et *Ladda K* (LK).

Objectif : Distribution des requêtes déléguées à travers les clients.

Description : Dans la figure 4.11 les clients sont groupés par le nombre de requêtes déléguées qu'ils ont exécutées. Ainsi la catégorie A regroupe les clients ayant effectué moins de 60 requêtes déléguées, la catégorie B regroupe les clients ayant effectué entre 60 et 120 requêtes déléguées et la catégorie C regroupe les clients ayant effectué plus de 120 requêtes déléguées. On peut voir par approche : *Ladda 2* et *Ladda K* et par configuration : *1 chargé*, *1/4 chargés*, *1/2 chargés* et *Tous chargés*, combien les clients exécutent de requêtes déléguées. Les résultats présentés sont une moyenne sur trois expériences.

Résultats : On peut constater que peu importe la configuration et l'approche, beaucoup de clients exécutent peu de requêtes déléguées et peu exécutent beaucoup de requêtes. La figure 4.12 [5] confirme cette tendance. Elle présente le nombre de requêtes déléguées exécutées par nombre de clients, pour les clients de la catégorie A, où les clients de la catégorie A ont exécuté moins de 60 requêtes déléguées. La figure 4.12 présente le nombre de clients qui ont exécuté un nombre donné de requêtes déléguées pour la configuration *Tous chargés*.

[5] Les figures pour les autres configurations sont présentées dans la section A (p. 81).

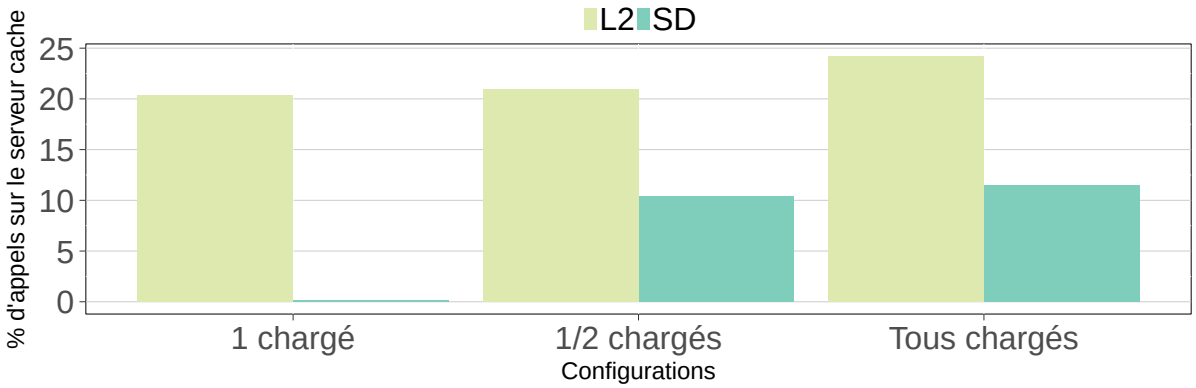


FIGURE 4.14 – Pourcentage des appels résolus grâce au serveur de cache Web, sur les appels HTTP non résolus localement, avec les approches *Ladda 2* (L2) et *Sans délégation* (SD).

Il y a plus de 10 clients avec *Ladda K* qui n'ont exécuté aucune requête déléguée. On peut constater qu'en réalité la plupart des clients ont exécuté moins de 10 requêtes déléguées.

Explication : Les clients qui exécutent beaucoup de requêtes déléguées, n'exécutent en réalité que des requêtes très peu coûteuses, qui s'exécutent rapidement. Cette hypothèse est confirmée par les résultats de la figure 4.13 où l'on observe le temps moyen en millisecondes des requêtes déléguées aux clients ayant exécuté plus de 120 requêtes déléguées (catégorie C). Les configurations *Tous chargés* et *1/2 chargés* ne sont pas présentes dans la figure, car aucun client dans ces configurations n'a exécuté plus de 120 requêtes déléguées. Néanmoins on voit que pour la configuration *1 chargé* et *Ladda 2*, les requêtes ne prennent pas plus de 800 millisecondes en moyenne.

4.4.2.5 Performance du serveur de cache Web

Dans Ladda le producteur de données met à disposition un serveur TPF et un serveur de cache Web pour absorber la charge.

Objectif : Est-ce que la parallélisation de requêtes impacte l'utilisation du serveur de cache Web ?

Description : Cette expérience mesure l'efficacité du serveur de cache Web après un tour d'initialisation. Les mesures sont donc effectuées sur un cache rempli.

Résultats : La figure 4.14 présente, pour les appels externes aux clients, le pourcentage qui a été pris en charge par le serveur de cache Web. L'expérience a été effectuée pour les approches *Ladda 2* et *Sans délégation*, avec les configurations : *1 chargé*, *1/2 chargés* et *Tous chargés*. De manière générale, le cache Web est plus efficace quand les requêtes sont exécutées en parallèle (*Ladda 2*), plutôt qu'en séquentiel (*Sans Délégation*). La différence de pourcentage d'appels interceptés par le cache Web est particulièrement prononcée pour la configuration *1 chargé*. Dans cette configuration, avec l'approche *Sans Délégation*, le cache Web n'intercepte environ que 1% des appels HTTP. Dans la même configuration, avec l'approche *Ladda 2*, le cache Web intercepte environ 20% des appels HTTP. On peut également constater que

le nombre d'appels absorbés par le serveur de cache Web est plus important avec l'approche *Tous chargés* qu'avec l'approche *1/2 chargés*, soit ≈20% pour *1/2 chargés* et ≈24% pour *Tous chargés*.

Explication : Dans la configuration *1 chargé*, les 1509 requêtes sont placées sur le même client au départ. Tandis que pour la configuration *Tous chargés*, les requêtes sont distribuées sur les 50 clients et donc naturellement parallélisées. Ainsi, lorsque les requêtes sont exécutées, il y a plus de requêtes exécutées en parallèle avec la configuration *Tous chargés*, qu'avec la configuration *1 chargé*. Ce parallélisme est accru grâce à la délégation, notamment pour les configurations *1 chargé* et *1/2 chargés*. On peut donc en conclure que plus il y a de requêtes exécutées en parallèle, plus le cache Web est efficace.

4.5 État de l'art

L'approche Triple Pattern Fragment (TPF) [61, 62] propose de déplacer le traitement complexe des requêtes des serveurs aux clients dans le but de réduire le coût du côté serveur. TPF permet de réduire le coût de publication des données comparé aux serveurs SPARQL. Les clients TPF décomposent les requêtes SPARQL en de multiples requêtes élémentaires et effectuent les opérations de jointure en local. Par conséquent, le traitement de requêtes SPARQL est distribué entre les clients et les serveurs. Comparé à Ladda, les clients TPF ne collaborent par entre eux pour traiter les requêtes.

Ladda a de nombreux points communs avec les bases de données fédérées et la gestion de données sur réseau pair-à-pair [44, 54, 55]. Dans ces approches, la fédération est construite du côté des producteurs de données, c'est-à-dire que chaque pair héberge un fragment des données et peut exécuter des requêtes. Avec Ladda, la fédération est déployée du côté des consommateurs de données, c'est-à-dire que les données restent sur les serveurs TPF. Ladda implémente la parallélisation inter-requêtes du côté des clients. Les gains en performance sont obtenus par un meilleur usage du temps libre du serveur TPF et des caches.

Dans CyCLaDEs [24], les clients TPF sont modifiés pour construire un cache collaboratif dans le but de réduire le nombre d'appels aux serveurs TPF. Quand un client exécute une requête, il vérifie si le triplet est dans son cache local, puis dans le cache local de ses voisins. CyCLaDEs déplace les données entre les voisins, tandis que Ladda déplace les requêtes entre les voisins. Les deux approches sont complémentaires et peuvent être utilisées ensemble.

Le problème scientifique derrière Ladda se situe dans le domaine de la théorie de l'ordonnancement, développée principalement dans le domaine des systèmes distribués [16, 36]. Dans Ladda les consommateurs de données traitent des requêtes hétérogènes et indépendantes, sans connaître leurs temps d'exécution, en suivant un modèle "open arrival", c'est-à-dire que les requêtes arrivent n'importe quand sur n'importe quel nœud. Les consommateurs de données sont à la fois consommateurs et ressources. Ils fournissent un ensemble de ressources hétérogènes géographiquement distribuées.

L'algorithme proposé dans Ladda appartient aux algorithmes d'ordonnancement dynamiques distribués non-coopératifs [3, 16, 36, 49]. Le problème *balls and bins* [39] est représentatif de cette catégorie, m balles doivent être allouées à n poubelles. L'une des stratégies est d'allouer chaque balle de façon aléatoire à une poubelle. Cependant, les résultats montrent que choisir deux poubelles de façon aléatoire et choisir la moins chargée, réduit de façon exponentielle la charge maximale. Mais, cela suppose que le temps d'exécution des tâches

soit connu et de choisir de façon aléatoire parmi toutes les poubelles pour chaque balle. Malheureusement, dans le contexte de Ladda, le temps d'exécution des requêtes n'est pas connu et par conséquent la charge des clients non plus. Nous ne pouvons également pas choisir de façon aléatoire parmi tous les clients, un client voit seulement un sous-ensemble aléatoire du réseau. Finalement, les clients dans Ladda ne délèguent pas seulement les requêtes, il doivent aussi attendre les résultats. Comparé à *balls and bins*, Ladda implémente une allocation aléatoire des balles dans les poubelles, elles-mêmes choisies de manière aléatoire parmi l'ensemble des poubelles.

Comme dans *balls and bins*, Ladda est naturellement décentralisé parce que les requêtes peuvent arriver à n'importe quel moment sur n'importe quel nœud. Se baser sur un ordonnanceur centralisé dynamique introduirait un point de défaillance unique et un coût élevé. Comme dans *balls and bins*, Ladda est non-coopératif pour éviter tout coût supplémentaire et toute forme de synchronisation.

Selon [36, 49], les solutions dynamiques décentralisées sont soit *receiver-initiative/pull-based*, soit *sender-initative/push-based*. Dans les solutions *push-based*, quand une tâche arrive sur un nœud, le nœud exécute la tâche ou la transfère. Selon [21], les approches *push-based* sont plus adaptées dans les réseaux faiblement ou moyennement chargés, tandis que les approches *pull-based* sont meilleures pour les réseaux fortement chargés. Dans Ladda, nous supposons que la fédération n'est pas fortement chargée et nous suivons une approche *push-based*.

Trouver des nœuds *libres* est essentiel pour les performances de Ladda. Dans l'approche Balanced Overlay Networks [13] (BON), la disponibilité des nœuds est encodée dans le réseau superposé. Chaque nœud maintient un nombre d'arcs en entrée proportionnel aux ressources libres. Cela permet de trouver simplement les nœuds disponibles grâce à une marche aléatoire. Dans Ladda, l'état du client change rapidement pendant l'exécution des requêtes. Dans ce contexte, le coût de BON devient trop important.

4.6 Conclusion

Ladda permet de construire une fédération de consommateurs de données où les consommateurs de données sont connectés entre eux. Grâce à cette fédération les consommateurs de données peuvent partager leurs capacités de traitement de requêtes SPARQL (temps de calcul et leur bande passante). Ainsi, les consommateurs de données peuvent paralléliser leurs requêtes en les déléguant à leurs voisins disponibles. Ladda implémente un équilibreur de charge dynamique qui permet la parallélisation inter-requêtes sur les consommateurs de données. Ladda réduit significativement le temps global d'exécution et améliore le débit de requêtes de la fédération. Ladda exploite mieux le temps libre du serveur TPF et les caches Web.

Ladda propose du parallélisme inter-requêtes. Une autre direction de recherche est de considérer le parallélisme intra-requêtes. La décomposition de requêtes SPARQL et la délégation de sous-requêtes ouvrent des perspectives intéressantes. En effet, la localité des données n'a pas d'impact sur le parallélisme inter-requêtes, mais peut jouer un rôle important dans le parallélisme intra-requêtes. La réplication des fragments souvent utilisés sur les consommateurs de données et la possibilité de parallélisme intra-requêtes a le potentiel d'améliorer significativement le temps d'exécution des requêtes.

5

Conclusion

Sommaire

Ce chapitre présente un résumé des contributions et détaille les perspectives découlant des travaux présentés dans cette thèse.

5.1 Résumé des contributions

Dans cette thèse, nous nous sommes intéressés à l'exécution de requêtes SPARQL sur le Web des données. En suivant une approche basée sur une multitude de médiateurs accédant aux données, nous nous sommes posé la question suivante :

> Si les médiateurs collaborent en partageant leurs ressources en espace disque, capacité de calcul et bande passante, est-il possible d'obtenir un meilleur compromis entre performances et disponibilité des données ?

L'approche proposée est originale dans le sens où seuls les médiateurs collaborent et non les sources de données. Ce problème est difficile car toute collaboration à un coût et le choix de ne pas collaborer peut être plus avantageux. Nous avons développé deux contributions :

- CyCLaDEs permet aux médiateurs de partager leur cache local. Nous avons montré que 20% des appels peuvent être résolus au niveau du cache collaboratif et améliorer la disponibilité des serveurs.

- Ladda permet de paralléliser des requêtes sur des médiateurs libres. Nous montrons qu'une meilleure utilisation des caches serveurs permet d'améliorer les temps d'exécution d'un ensemble de requêtes sous certaines conditions.

5.1.1 Cache collaboratif décentralisé

Dans l'approche TPF, les caches locaux et les caches serveurs jouent un rôle très important pour la disponibilité des données. Dans CyCLaDEs, nous partageons les caches locaux entre médiateurs. Dans cette approche, de faibles latences d'accès aux caches locaux sont primordiales. C'est pourquoi nous avons suivi une approche basée sur un cache comportemental. Afin de limiter le surcoût au maximum, un médiateur doit avoir une connexion directe établie avec un sous-ensemble de médiateurs utiles pour ses requêtes. Pour un médiateur donné, CyCLaDEs calcule ses meilleurs voisins sur la base d'un profil des requêtes exécutées dans un passé récent.

Lors de l'exécution d'une requête SPARQL, un client va d'abord vérifier dans son cache local, puis dans le cache local de ses voisins similaires et en dernier recours la requête sera transmise à la source de données. Les résultats démontrent que le cache collaboratif décentralisé est capable d'absorber environ ≈20% de la charge initialement supportée par le producteur de données. Le profilage des médiateurs permet de rassembler les médiateurs interrogeant les mêmes sources de données ensemble. L'ouverture des caches locaux aux autres médiateurs augmente sensiblement la probabilité qu'une entrée puisse être ré-utilisée.

5.1.2 Délégation de requêtes

Dans CyCLaDEs, les médiateurs partagent principalement leurs ressources de stockage, tandis que dans Ladda nous cherchons à partager les ressources en capacité de calcul et en bande passante. L'approche TPF transforme chaque médiateur en un serveur SPARQL où une partie importante du traitement d'une requête est effectuée dans le médiateur. Cette approche transfère beaucoup plus de données qu'une approche basée sur des serveurs SPARQL. Si, *in fine*, toutes les données sont accédées sur les mêmes serveurs TPF, un médiateur peut être limité par sa capacité de calcul et sa bande passante.

Dans Ladda, un médiateur peut déléguer l'exécution de certaines de ses requêtes à d'autres médiateurs choisis aléatoirement. La délégation échoue si le médiateur est occupé et la requête doit être replanifiée. Les résultats démontrent que le temps global d'exécution d'un ensemble de requêtes peut être grandement réduit. Dans une situation favorable, où un seul client est chargé et tous ses voisins sont libres, l'exécution passe de $\approx$ 30 min à $\approx$ 3 min grâce à la délégation de requêtes. Dans une situation défavorable, où tous les clients ont des requêtes à exécuter, l'exécution passe de $\approx$ 10 min à $\approx$ 4 min.

5.2 Perspectives

Cette section développe les différentes améliorations possibles des travaux présentés dans cette thèse.

5.2.1 Analyse de coût de CyCLaDEs

Dans CyCLaDEs, nous nous sommes focalisés sur l'efficacité du cache collaboratif. Nous n'avons pas mesuré l'impact en terme de temps d'exécution des requêtes. La latence du cache collaboratif est : $2 \times l$, où l est la latence du voisin le plus rapide, la première fois pour envoyer la requête et la deuxième fois pour recevoir la réponse. Le temps d'exécution des requêtes est dégradé si les serveurs sont peu chargés. Si la charge des serveurs augmente, alors la meilleure disponibilité des données doit compenser le surcoût d'accès au cache collaboratif. Mesurer dans quelles conditions les temps d'exécution des requêtes s'améliorent avec un cache collaboratif est une perspective intéressante.

CyCLaDEs ne tient pas compte des latences réseaux entre les médiateurs et entre les médiateurs et les sources de données. Intégrer les latences réseaux dans le profil d'un médiateur doit permettre d'obtenir un meilleur voisinage.

5.2.2 Délégation de requêtes en environnement réel

Ladda parallélise les requêtes entre les différents médiateurs, mais *in fine*, tous les médiateurs accèdent aux mêmes sources. Si Ladda ne contrôle pas le flux sortant de requêtes vers les serveurs TPF, alors Ladda peut surcharger les serveurs. Une perspective intéressant est d'intégrer à Ladda un mécanisme de régulation du flot de requêtes sortant. L'idée est la suivante : si la fédération de médiateurs détecte que les serveurs TPF sont chargés alors la probabilité de déléguer une requête diminue et il faut diminuer le nombre de requêtes qu'un médiateur peut déléguer. Il s'agit donc de coupler la probabilité de déléguer une requête à l'estimation de la charge du serveur.

5.2.3 Combiner le cache collaboratif décentralisé et la délégation de requête

Avec CyCLaDEs, la pression sur le producteur de données est diminuée car une partie des appels est pris en charge par le cache collaboratif. Tandis que Ladda, en faisant de la parallélisation inter-requêtes, rend le serveur de cache plus efficace. Il est possible de combiner le cache collaboratif décentralisé à la délégation de requêtes. Ainsi lorsqu'une requête est exécutée chez un médiateur celui-ci peut faire appel au cache local de ses voisins afin de réduire la pression sur le producteur de données. De manière à ne pas perturber l'exécution des requêtes SPARQL, les demandes d'accès au cache local devraient être gérées par un sous-processus dédié. La combinaison de ces deux approches devrait permettre de transférer une partie des requêtes préalablement résolues chez le producteur de données sur le cache collaboratif décentralisé, tout en permettant de réduire le temps d'exécution pour les consommateurs de données.

5.2.4 Matérialisation de fragment

Lors de l'exécution de requêtes, un client TPF décompose la requête SPARQL en requêtes de fragment. Chaque réponse à ces requêtes de fragment est stockée dans le cache local du client TPF. Chaque entrée du cache local est un tuple (*clé*, *valeur*) où la *clé* est le *sélecteur* de la requête de triplet et la *valeur* est le fragment de triplet associé. On distingue alors deux

types de fragments de triplet : les fragments *complets* (*download fragment*) et les fragments *partiels* (*biding fragment*). Les fragments *complets* sont des fragments dont le *sélecteur* de la requête est de type : `s = { ?s rdf:type ?p}` , où le sujet et l'objet sont des variables. Alors que les fragments *partiels* ont au moins le sujet ou l'objet instancié en plus du prédicat : `s = {dbpedia:"A"_Is_for_Alibi rdf:type ?author}`. Les fragments complets ont plus de chance d'être réutilisés dans une autre requête que les fragments partiels car ils sont moins sélectifs.

Chaque fragment, qu'il soit complet ou partiel contient également des méta-données, comme le nombre de triplets correspondant au sélecteur qui lui est attaché (cf. chapitre 2 p. 17). Dans les expériences de Ladda (cf. chapitre 4 section 4.3.2 p. 60) nous avons démontré que le cache local d'un client est principalement réutilisé durant l'exécution d'une même requête. Après analyse, les appels résolus lors de l'exécution d'une requête grâce au cache local servent principalement à récupérer la cardinalité d'un fragment. Matérialiser seulement les fragments complets pourrait donc impacter les performances du client TPF. Il faudrait alors avoir deux caches, un de méta-données et un autre contenant les fragments complets. De plus, le client TPF devrait être étendu pour gérer l'inclusion d'un fragment dans un autre, afin de tirer pleinement profit des deux caches. Supposons les fragments avec les sélecteurs suivants : $f1_s =$`{ ?s rdf:type ?p }` et $f2_s =$`{ ?s rdf:type dbpedia-owl:Book }`. En faisant une comparaison lexicale, les sélecteurs des deux fragments sont différents. Cependant, le sélecteur du fragment $f2$ est un sous-ensemble du fragment $f1$. Les réponses du fragment $f2$ sont donc incluses dans les réponses du fragment $f1$.

Grâce à la matérialisation de fragments complets, une communauté donnée pourrait alors se coordonner pour matérialiser les fragments les plus populaires comme proposé dans [6]. En présence de fragments complets, il devient possible de décomposer les requêtes pour profiter de ces fragment complets, comme proposé dans [40]. La décomposition de requête permet de profiter de la localité des données et introduit du parallélisme intra-requête.

A

Résultats d'expérimentation complémentaires de Ladda

Sommaire

Ce chapitre présente des résultats complémentaires pour les expériences de Ladda, présentées dans le chapitre 4 section 4.4 (p. 64).

A.1 Nombre de requêtes exécutées par minute

Dans Ladda nous mesurons le débit global de requêtes par minute, mais il est aussi intéressant de voir le nombre de requêtes exécutées par minute.

Objectif : Est-ce que le débit de requêtes est toujours meilleur dans le temps ?

Description : Suivant l'expérience, le temps d'exécution pour chaque approche peut varier un peu. Il n'est donc pas possible de faire une moyenne sur les trois expériences comme précédemment. Les figures qui suivent ne présentent donc que les résultats pour une expérience.

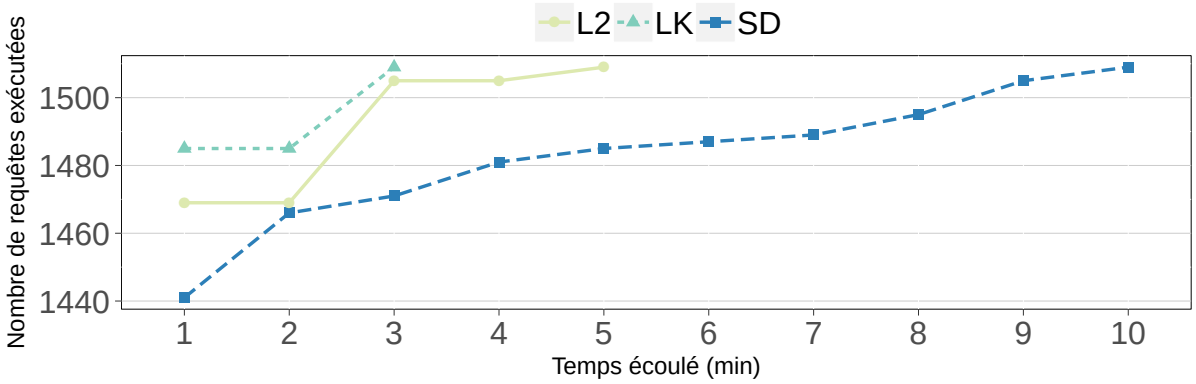


FIGURE A.1 – Nombre de requêtes exécutées par minute par *Ladda 2* (L2) , *Ladda K* (LK) et l'approche de référence *Sans délégation* (SD) dans la configuration 1/2 *chargés*.

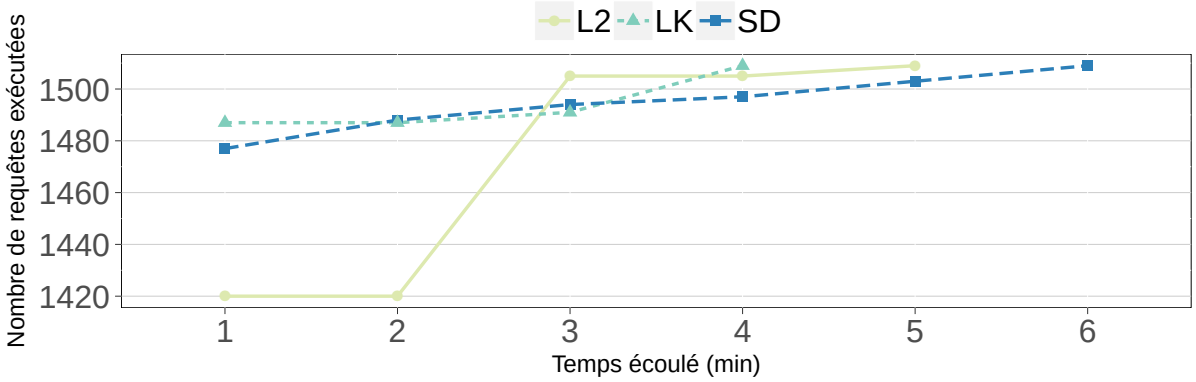


FIGURE A.2 – Nombre de requêtes exécutées par minute par *Ladda 2* (L2) , *Ladda K* (LK) et l'approche de référence *Sans délégation* (SD) dans la configuration 1/4 *chargés*.

Résultats : La figure A.1 présente le nombre de requêtes exécutées par minute pour les approches *Ladda 2*, *Ladda K* et *Sans délégation* dans la configuration *1/2 chargés*. On peut constater deux choses : (i) l'approche *Ladda K* finit avant *Ladda 2*; (ii) *Ladda 2* et *Ladda K* ont déjà exécuté plus de requêtes que l'approche *Sans délégation* dès la première minute.

La figure A.2 présente le nombre de requêtes exécutées par minute pour les approches *Ladda 2*, *Ladda K* et *Sans délégation* dans la configuration *1/4 chargés*. Contrairement à la configuration *1/2 chargés*, dans la configuration *1/4 chargés Ladda* n'a pas toujours le meilleur débit de requêtes dans le temps. Pour l'approche *Ladda 2*, il faut attendre la 3ème minute pour qu'elle ait un meilleur débit que l'approche *Sans délégation*.

La figure A.3 présente le nombre de requêtes exécutées par minute pour les approches *Ladda 2*, *Ladda K* et *Sans délégation* dans la configuration *1 chargé*. Dans cette configuration l'approche *Sans délégation* exécute les 1509 requêtes en séquentiel comme elles sont toutes sur le même client, alors qu'avec *Ladda* l'exécution se fait en parallèle grâce à la délégation de requêtes. Par conséquent, au bout de la première minute, l'approche *Sans délégation* n'a exécutée qu'une seule requête alors que *Ladda 2* et *Ladda K* ont exécuté ≈1500 requêtes, soit quasiment toutes les requêtes.

Explication : Dans la figure A.1 *Ladda K* finit avant *Ladda 2*, car dès les premières minutes il a réussi à trouver plus de clients *libres* et a donc pu paralléliser plus de requêtes que

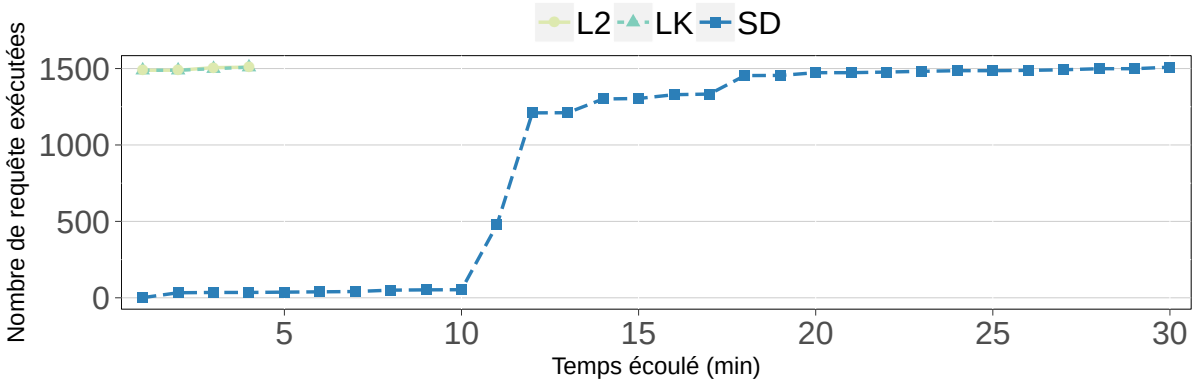


FIGURE A.3 – Nombre de requêtes exécutées par minute par *Ladda 2* (L2) , *Ladda K* (LK) et l'approche de référence *Sans délégation* (SD) dans la configuration *1 chargé*.

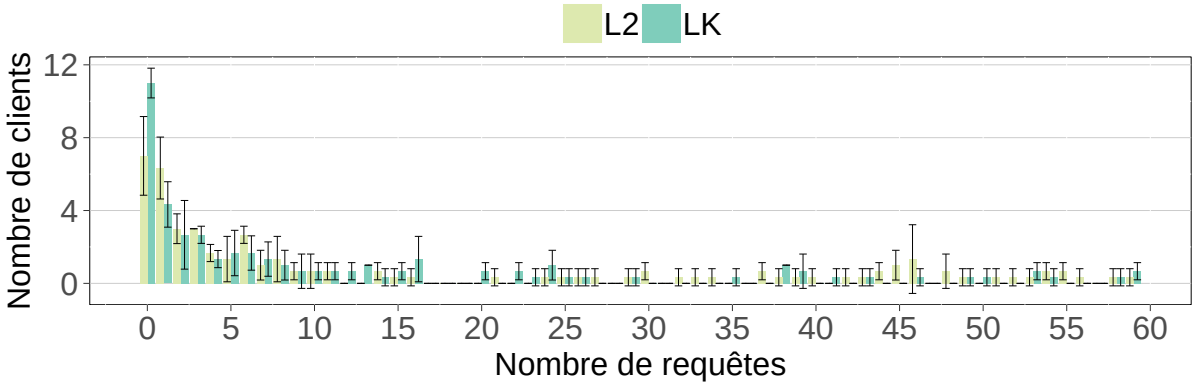


FIGURE A.4 – Nombre de clients pour un nombre de requêtes exécutées donné, avec un écart type sur le nombre de clients pour la configuration *1/2 chargés* et les approches : *Ladda 2* (L2) et *Ladda K* (LK).

Ladda 2, alors que dans la figure A.2 l'approche *Sans délégation* n'est pas loin de *Ladda K*. On peut constater que dans la configuration *1/4 chargés* l'approche *Sans délégation* a déjà plus de requêtes exécutées au bout de la première minute que dans la configuration *1/2 chargés*, environ ≈1480 requêtes pour *1/4 chargés* et environ ≈1440 requêtes pour *1/2 chargés*. Bien que les requêtes soient déjà naturellement parallélisées dans la configuration *1/2 chargés* et *1/4 chargés*, le nombre de requêtes exécutées à un instant t n'est pas le même pour les deux configurations. Pour *1/2 chargés* et l'approche *Sans délégation* il y a au plus 25 requêtes exécutées en parallèle dans la fédération. Pour *1/4 chargés* et l'approche *Sans délégation* il y a au plus 12 requêtes exécutées en parallèle dans la fédération. Comme on a pu le voir dans l'expérience décrite au chapitre 4 §4.4.2.3 (p. 69), il est possible de gagner globalement grâce à la délégation mais ce n'est pas le cas pour tous les clients de la fédération. Le nombre de requêtes exécutées à un instant t impacte le temps d'exécution des requêtes. C'est ce qu'il se passe entre l'exécution *1/2 chargés* et *1/4 chargés*.

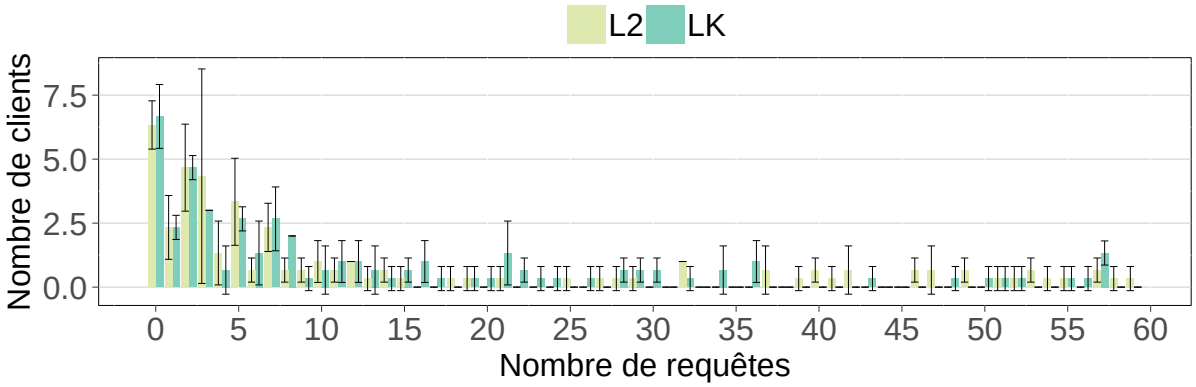


FIGURE A.5 – Nombre de clients pour un nombre de requêtes exécutées donné, avec un écart type sur le nombre de clients pour la configuration *1/4 chargés* et les approches : *Ladda 2* (L2) et *Ladda K* (LK).

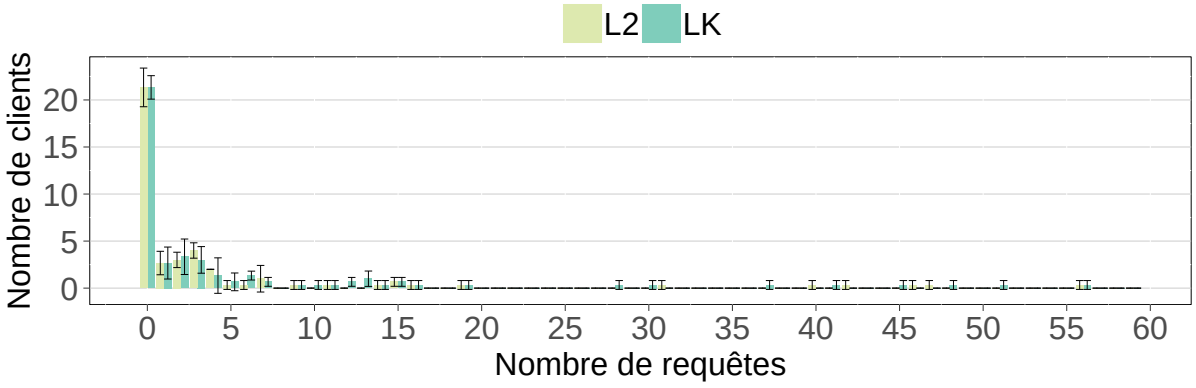


FIGURE A.6 – Nombre de clients pour un nombre de requêtes exécutées donné, avec un écart type sur le nombre de clients pour la configuration *1 chargé* et les approches : *Ladda 2* (L2) et *Ladda K* (LK).

A.2 Distribution des requêtes déléguées

Dans Ladda un consommateur de données peut déléguer ses requêtes à l'un de ses voisins *libres*.

Objectif : Est-ce qu'il y a un point chaud dans la délégation de requêtes ?

Description : Les figures A.4 à A.6 présentent le nombre de clients ayant exécuté un nombre de requêtes donné pour les clients de la catégorie A, soit les clients ayant exécutés entre 0 et 60 requêtes déléguées. Ces figures sont des moyennes sur trois expériences.

Résultats : La figure A.4 présente la distribution des requêtes déléguées par nombre de clients pour la configuration *1/2 chargés* et les approches *Ladda 2*, *Ladda K* et *Sans délégation*. On peut constater que la majorité des clients exécute moins de 15 requêtes déléguées. Cependant, avec l'approche *Ladda K* il y a plus de clients qui n'exécute aucune requête déléguée qu'avec l'approche *Ladda 2*.

La figure A.5 présente la distribution des requêtes déléguées par nombre de clients pour la configuration *1/4 chargés* et les approches *Ladda 2*, *Ladda K* et *Sans délégation*. Comme précédemment la majorité des clients exécute moins de 15 requêtes déléguées.

La figure A.6 présente la distribution des requêtes déléguées par nombre de clients pour la configuration *1 chargé* et les approches *Ladda 2*, *Ladda K* et *Sans délégation*. Dans cette configuration les requêtes sont placées au départ sur un seul client. On peut constater que ≈42% des clients n'exécutent aucune requête déléguée et pour les clients qui en exécutent, ils ne dépassent pas les 5 requêtes déléguées.

Explication : Si l'on compare la figure A.5 et A.4, on peut voir que globalement il y a plus de clients qui exécutent des requêtes déléguées dans la configuration *1/4 chargés* par rapport à la configuration *1/2 chargés*. Cela est lié à la distribution initiale des requêtes. Dans la configuration *1/2 chargés*, 21 clients ont des requêtes au début et 29 clients sont *libres*. Alors que dans la configuration *1/4 chargés*, 12 clients ont des requêtes et 38 clients sont *libres*. Par conséquent, dans la configuration *1/4 chargés*, les clients ont plus d'opportunités pour déléguer leurs requêtes.

Dans la configuration *1 chargé*, beaucoup de clients n'exécutent aucune requête déléguée. En effet, le client qui a les 1509 requêtes ne voit qu'un sous-ensemble du réseau, soit 21 voisins. Par conséquent, la charge de travail n'est pas suffisamment conséquente en terme de temps pour mobiliser l'ensemble de la fédération.

A.3 Temps local d'exécution

Dans Ladda nous avons vu que la délégation de requêtes est profitable à la fédération (cf. chapitre 4 §4.4.2.1 p. 67). Il est intéressant de savoir si cela est également vrai individuellement.

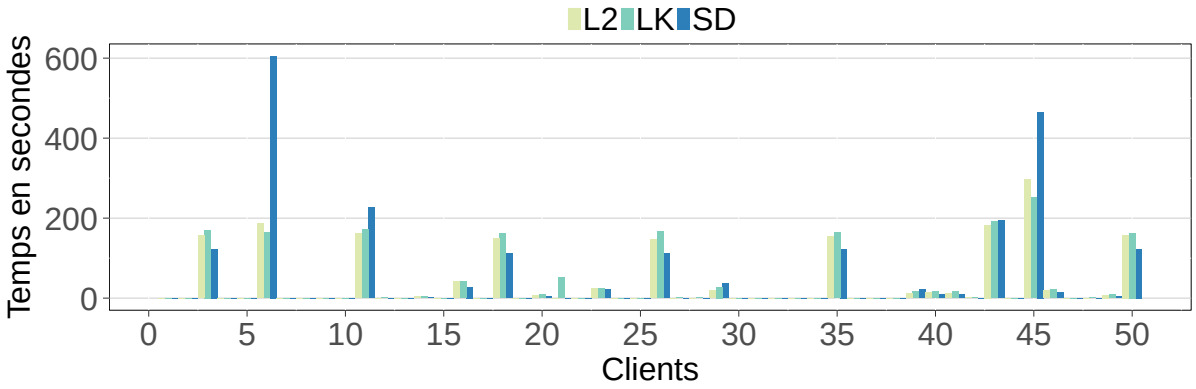


FIGURE A.7 – Temps local d'exécution par client avec *Ladda 2* (L2), *Ladda K* (LK) et *SD* (Sans délégation) pour la configuration *Tous chargés*.

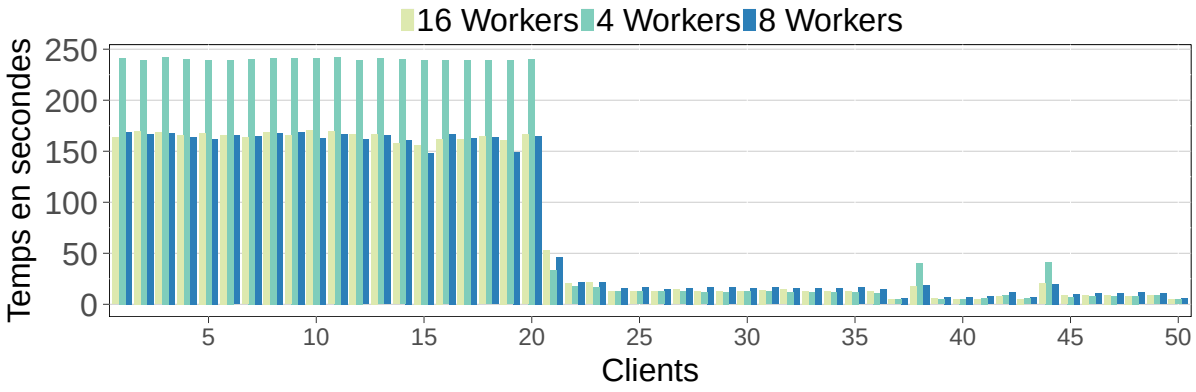


FIGURE A.8 – Temps local d'exécution par client avec l'approche *Sans Délégation* et la distribution statique des requêtes, sur 50 clients.

Objectif : Est-ce que la parallalélisation de requêtes profite à tous ?

Description : Le temps d'exécution local d'un client correspond au temps qu'il a mis pour exécuter ses requêtes, qu'il les ait exécutées localement ou déléguées. Moyenne sur trois expériences.

Résultats : La figure A.7 présente le temps d'exécution local par client pour la configuration *Tous chargés* et les approches : *Ladda 2*, *Ladda K* et *Sans délégation*. Comme pour les autres configurations (cf. chapitre 4 §4.4.2.3 p. 69), la distribution des requêtes n'est pas optimale. Lorsqu'un client gagne grâce à la parallélisation il gagne beaucoup, jusqu'à ≈200 secondes et lorsqu'il perd c'est de peu.

Explication : Comme précédemment, on peut constater que lorsqu'un client perd, il perd un peu plus avec l'approche *Ladda K* qu'avec *Ladda 2*. Dans le cadre de *Ladda K* un client trouve plus rapidement des voisins *libres*. À un instant t il y a donc plus de requêtes qui s'exécutent en parallèle qu'avec *Ladda 2* et c'est alors le serveur TPF qui devient un goulot d'étranglement. Cette hypothèse est confirmée lorsqu'on observe la distribution des 1509 requêtes en suivant l'algorithme LPT (cf. chapitre 4 section 4.3.1 p. 59) et qu'on varie le nombre de sous-processus (*workers*) sur le serveur. Les résultats sont présentés dans la fi-

gure A.8. On peut constater que pour les clients qui ont un temps d'exécution supérieur à 10 secondes, augmenter le nombre de sous-processus sur le serveur réduit le temps d'exécution du client.

Bibliographie

[1] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. Anapsid : an adaptive query processing engine for sparql endpoints. *The Semantic Web–ISWC 2011*, pages 18–34, 2011.

[2] P. Adjiman, P. Chatalic, F. Goasdoué, M.-C. Rousset, and L. Simon. Scalability study of peer-to-peer consequence finding. In *International Joint Conference on Artificial Intelligence*, 2005.

[3] A. M. Alakeel. A guide to dynamic load balancing in distributed computer systems. *International Journal of Computer Science and Information Security*, 10(6) :153–160, 2010.

[4] C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche. SPARQL Web-Querying Infrastructure : Ready for Action ? In *12th International Semantic Web Conference (ISWC 2013)*, volume 8219 of *LNCS*, pages 277–293. Springer, 2013.

[5] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia : A nucleus for a web of open data. *The semantic web*, pages 722–735, 2007.

[6] N. Bame. *Complex data management for ecological niche modeling*. Theses, Université Pierre et Marie Curie - Paris VI, June 2015.

[7] W. Beek, L. Rietveld, H. R. Bazoobandi, J. Wielemaker, and S. Schlobach. Lod laundromat : a uniform way of publishing other people's dirty data. In *International Semantic Web Conference*, pages 213–228. Springer, 2014.

[8] M. Bertier, D. Frey, R. Guerraoui, A. Kermarrec, and V. Leroy. The gossple anonymous social network. In *11th International Middleware Conference 'Middleware 2010) - ACM/IFIP/USENIX*, volume 6452 of *LNCS*, pages 191–211. Springer, 2010.

[9] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *International Journal of Semantic Web and Information Syststems*, 5(3) :1–22, 2009.

[10] C. Bizer, T. Heath, and T. Berners-Lee. Linked data-the story so far. *Semantic services, interoperability and web applications : emerging concepts*, pages 205–227, 2009.

[11] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems*, 5(2) :1–24, 2009.

[12] M. A. Blaze. *Caching in Large-scale Distributed File Systems*. PhD thesis, Princeton University, Princeton, NJ, USA, 1993. UMI Order No. GAX93-11182.

[13] J. S. Bridgewater, V. P. Roychowdhury, and P. O. Boykin. Balanced overlay networks (BON) : an overlay technology for decentralized load balancing. *IEEE transactions on parallel and distributed systems*, 18(8), 2007.

[14] M. Cai and M. Frank. Rdfpeers : a scalable distributed rdf repository based on a structured peer-to-peer network. In *Proceedings of the 13th international conference on World Wide Web*, pages 650–657. ACM, 2004.

[15] M. Cai, M. Frank, J. Chen, and P. Szekely. Maan : A multi-attribute addressable network for grid information services. In *Grid Computing, 2003. Proceedings. Fourth International Workshop on*, pages 184–191. IEEE, 2003.

[16] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transaction Software Engineering*, 14(2) :141–154, Feb. 1988.

[17] G. Cormode and S. Muthukrishnan. An improved data stream summary : the count-min sketch and its applications. *Journal of Algorithms*, 55(1) :58–75, 2005.

[18] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching : Using remote client memory to improve file system performance. In *1st USENIX Conference on Operating Systems Design and Implementation (OSDI 1994)*, Berkeley, CA, USA, 1994.

[19] M. d'Aquin and E. Motta. Watson, more than a semantic web search engine. *Semantic Web*, 2(1) :55–63, 2011.

[20] A. Doan, A. Halevy, and Z. Ives. *Principles of data integration*. Elsevier, 2012.

[21] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance evaluation*, 6(1) :53–68, 1986.

[22] M. D. Ekstrand, J. T. Riedl, J. A. Konstan, et al. Collaborative filtering recommender systems. *Foundations and Trends® in Human–Computer Interaction*, 4(2) :81–173, 2011.

[23] M. El Dick, E. Pacitti, and B. Kemme. Flower-cdn : A hybrid p2p overlay for efficient query processing in cdn. In *Proceedings of the 12th International Conference on Extending Database Technology : Advances in Database Technology*, EDBT '09, pages 427–438, New York, NY, USA, 2009. ACM.

[24] P. Folz, H. Skaf-Molli, and P. Molli. CyCLaDEs : a decentralized cache for Triple Pattern Fragments. In *ESWC : Extended Semantic Web Conference*, 2016.

[25] D. Frey, M. Goessens, and A. Kermarrec. Behave : Behavioral cache for web content. In *Distributed Applications and Interoperable Systems (DAIS 2014)*, volume 8460 of *LNCS*, pages 89–103. Springer, 2014.

[26] L. Gong et al. Project jxta : A technology overview. Technical report, Technical report, SUN Microsystems, April 2001. http ://www.jxta.org/project/www/docs/TechOverview. pdf, 2001.

[27] O. Görlitz and S. Staab. Federated data management and query optimization for linked open data. In *New Directions in Web Data Management 1*, pages 109–137. Springer, 2011.

[28] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. 1997.

[29] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer data management system. *IEEE Transactions on Knowledge and Data Engineering*, 16(7) :787–798, 2004.

[30] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza : Data management infrastructure for semantic web applications. In *Proceedings of the 12th international conference on World Wide Web*, pages 556–567. ACM, 2003.

[31] O. Hartig. How caching improves efficiency and result completeness for querying linked data. In *WWW2011 Workshop on Linked Data on the Web, Hyderabad, India, March 29, 2011*, 2011.

[32] O. Hartig. Squin : a traversal based query execution system for the web of linked data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1081–1084. ACM, 2013.

[33] O. Hartig. Linked data query processing based on link traversal. In A. Harth, K. Hose, and R. Schenkel, editors, *Linked Data Management.*, pages 263–283. Chapman and Hall/CRC, 2014.

[34] S. Iyer, A. Rowstron, and P. Druschel. Squirrel : A decentralized peer-to-peer web cache. In *Twenty-first Annual Symposium on Principles of Distributed Computing (PODC 2002)*, pages 213–222, New York, NY, USA, 2002. ACM.

[35] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 11, pages 47–55. ACM, 1982.

[36] R. V. Lopes and D. Menascé. A taxonomy of job scheduling on distributed computing systems. *Transactions on Parallel and Distributed Systems*, 27 :3412 – 3428, 2016.

[37] F. Manola and E. Miller. Resource description framework (rdf) primer. *W3C Recommendation*, 10 :5, 2004.

[38] M. Martin, J. Unbehauen, and S. Auer. Improving the performance of semantic web applications with sparql query caching. In *Extended Semantic Web Conference (ESWC 2010)*, pages 304–318, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10) :1094–1104, 2001.

[40] G. Montoya, H. Skaf-Molli, P. Molli, and M.-E. Vidal. Federated sparql queries processing with replicated fragments. In *International Semantic Web Conference*, pages 36–51. Springer International Publishing, 2015.

[41] W. Nejdl, W. Siberski, and M. Sintek. Design issues and challenges for rdf-and schema-based peer-to-peer systems. *ACM SIGMOD Record*, 32(3) :41–46, 2003.

[42] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. Edutella : a p2p networking infrastructure based on rdf. In *Proceedings of the 11th international conference on World Wide Web*, pages 604–615. ACM, 2002.

[43] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Löser. Super-peer-based routing and clustering strategies for rdf-based peer-to-peer networks. In *Proceedings of the 12th international conference on World Wide Web*, pages 536–543. ACM, 2003.

[44] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[45] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris. Graph-aware, workload-adaptive sparql query caching. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1777–1792. ACM, 2015.

[46] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3) :16, 2009.

[47] M. Pinedo. *Scheduling : Theory, Algorithms, and Systems (3nd ed.)*. Springer Science and Business Media, 2008.

[48] E. Prud, A. Seaborne, et al. Sparql query language for rdf. 2006.

[49] H. G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *IEE Proceedings-Computers and Digital Techniques*, 141(1) :1–10, 1994.

[50] M. T. Schlosser, M. Sintek, S. Decker, and W. Nejdl. Hypercup-hypercubes, ontologies, and efficient search on peer-to-peer networks. In *AP2PC*, volume 2530, pages 112–124. Springer, 2002.

[51] M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of the linked data best practices in different topical domains. In *13th International Semantic Web Conference (ISWC 2014)*, pages 245–260, 2014.

[52] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench : A benchmark suite for federated semantic data query processing. *The Semantic Web–ISWC 2011*, pages 585–600, 2011.

[53] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx : Optimization techniques for federated query processing on linked data. In *International Semantic Web Conference*, pages 601–616. Springer, 2011.

[54] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys (CSUR)*, 22(3) :183–236, 1990.

[55] S. Staab and H. Stuckenschmidt. *Semantic Web and peer-to-peer*. Springer, 2006.

[56] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4) :149–160, 2001.

[57] H. Stuckenschmidt. Similarity-based query caching. In *Flexible Query Answering Systems*, volume 3055 of *Lecture Notes in Computer Science*, pages 295–306. Springer Berlin Heidelberg, 2004.

[58] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of disco. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 449–457. IEEE, 1996.

[59] G. Tummarello, R. Cyganiak, M. Catasta, S. Danielczyk, R. Delbru, and S. Decker. Sig. ma : Live views on the web of data. *Web Semantics : Science, Services and Agents on the World Wide Web*, 8(4) :355–364, 2010.

[60] G. Tummarello, R. Delbru, and E. Oren. Sindice. com : Weaving the open linked data. In *The Semantic Web*, pages 552–565. Springer, 2007.

[61] R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. Van de Walle. Querying datasets on the Web with high availability. In *Proceedings of the 13th International Semantic Web Conference*, volume 8796 of *Lecture Notes in Computer Science*, pages 180–196. Springer, Oct. 2014.

[62] R. Verborgh, M. Vander Sande, P. Colpaert, S. Coppens, E. Mannens, and R. Van de Walle. Web-scale querying through Linked Data Fragments. In *Proceedings of the 7th Workshop on Linked Data on the Web*, volume 1184 of *CEUR Workshop Proceedings*, Apr. 2014.

[63] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple Pattern Fragments : a low-cost knowledge graph interface for the Web. *Journal of Web Semantics*, 37–38 :184–206, Mar. 2016.

[64] S. Voulgaris, D. Gavidia, and M. Van Steen. Cyclon : Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2) :197–217, 2005.

[65] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3) :38–49, 1992.

[66] M. Yang and G. Wu. Caching intermediate result of sparql queries. In *Proceedings of the 20th international conference companion on World wide web*, pages 159–160. ACM, 2011.

Liste des tableaux

Table des figures

Nantes Métropole

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Pauline FOLZ

Collaboration dans une fédération de consommateurs de données liées

Collaboration in a Federation of Linked Data Consumers

Résumé

Les producteurs de données ont publié des millions de faits RDF sur le Web en suivant les principes des données liées. N'importe qui peut récupérer des informations utiles en interrogeant les données liées avec des requêtes SPARQL. Ces requêtes sont utiles dans plusieurs domaines, comme la santé ou le journalisme des données. Cependant, il y a un compromis entre la performance des requêtes et la disponibilité des données lors de l'exécution des requêtes SPARQL.
Dans cette thèse, nous étudions comment la collaboration des consommateurs de données ouvre de nouvelles opportunités concernant ce compromis. Plus précisément, comment la collaboration des consommateurs de données peut : améliorer les performances sans dégrader la disponibilité, ou améliorer la disponibilité sans dégrader les performances.
Nous considérons que les données liées permettent à n'importe qui d'exécuter un médiateur compact qui peut interroger des sources de données sur le Web grâce à des requêtes SPARQL. L'idée principale est de connecter ces médiateurs ensemble pour construire une fédération de consommateurs de données liées. Dans cette fédération, chaque médiateur interagit avec un sous-ensemble du réseau. Grâce à cette fédération, nous avons construit : (i) un cache décentralisé hébergé par les médiateurs. Ce cache côté client permet de prendre en charge une part importante des sous-requêtes et d'améliorer la disponibilité des données avec un impact faible sur les performances. (ii) un algorithme de délégation qui permet aux médiateurs de déléguer leurs requêtes à d'autres médiateurs. Nous démontrons que la délégation permet d'exécuter un ensemble de requêtes plus rapidement quand les médiateurs collaborent. Cela améliore les performances sans dégrader la disponibilité des données.

Mots clés

Web sémantique, cache collaboratif, ordonnancement, fédération, SPARQL, TPF, réseaux superposés.

Abstract

Following the Linked Data principles, data providers have published billions of RDF facts on the web. Anyone can retrieve some relevant information from the Linked Data by executing SPARQL queries. Such queries are useful in many domains including health or data journalism. However, there is a trade-off between performances of the queries and data availability when executing SPARQL queries. In this thesis, we have investigated how the collaboration of data consumers is opening new opportunities in this trade-off. More precisely, how the collaboration of data consumers can improve performances without degrading availability, or can improve availability without degrading performances.
We consider that Linked Data can allow anyone to run a compact mediator that executes SPARQL queries over data sources on the web. The main idea is to connect these mediators together to build a federation of Linked Data consumers. In this federation, each mediator interacts with a subset of the network. Thanks to this federation, we have built : (i) a decentralized cache hosted by mediators. This client-side cache is able to handle a significative part of subqueries and then improve data availability without a low impact on performances. (ii) a delegation algorithm that allows mediators to delegate their queries to other mediators. We have demonstrated that delegation allows to run the workloads faster when collaborating. This clearly improves performances without degrading data availability.

Key Words

Semantic Web, decentralized cache, scheduling, load balancing, federation, SPARQL, TPF, overlay network, peer-to-peer.