

Évaluation d'une architecture de stockage RDF distribuée

Maeva Antoine, Françoise Baude, Fabrice Huet

▶ **To cite this version:**

Maeva Antoine, Françoise Baude, Fabrice Huet. Évaluation d'une architecture de stockage RDF distribuée. IC 2012 - 23èmes journées francophones d'Ingénierie des Connaissances, Jun 2012, Paris, France. hal-00908461

HAL Id: hal-00908461
https://inria.hal.science/hal-00908461

Submitted on 23 Nov 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Évaluation d'une architecture de stockage RDF distribuée

Maeva Antoine[1], Françoise Baude[1], Fabrice Huet[1]

[1]INRIA MÉDITERRANÉE (ÉQUIPE OASIS), UNIVERSITÉ NICE SOPHIA-ANTIPOLIS, I3S CNRS
prénom.nom@inria.fr

Résumé : Stocker des informations du web sémantique implique d'être capable de pouvoir potentiellement gérer de très importants volumes de données. D'où le besoin d'opter pour une solution forcément distribuée, entre autres de type pair-à-pair, pour pouvoir passer à l'échelle. Un système de stockage RDF réparti requiert de mettre en place un algorithme particulier pour la résolution des requêtes SPARQL. Les données étant distribuées à travers un réseau de pairs, il est nécessaire d'exécuter une partie de la requête sur certains de ces pairs, puis d'agréger et d'appliquer d'éventuelles conditions de filtrage sur les différents résultats intermédiaires obtenus, avant de pouvoir retourner les résultats finaux. Il existe actuellement une douzaine de benchmarks pour le RDF, mais aucun d'entre eux ne se présente comme étant pensé pour s'adapter à une architecture de stockage réparti. Dans cet article, nous mettons en évidence le nombre de résultats intermédiaires générés par les requêtes SPARQL, un aspect important dans un contexte distribué, et qui nous semble à l'heure actuelle négligé par les benchmarks pour le RDF.

1 Introduction

Stocker des informations du web sémantique implique d'être capable de pouvoir potentiellement gérer de très importants volumes de données. D'où le besoin d'opter pour une solution forcément distribuée pour pouvoir passer à l'échelle. Le problème rencontré à l'heure actuelle est que les benchmarks pour le RDF (Klyne *et al.*, 2004) se destinent à des architectures centralisées et ne prennent pas en considération certains aspects spécifiques à la résolution de requêtes sur un système distribué. En effet, le principal avantage du stockage réparti qui est de permettre plus facilement le passage à l'échelle est contrebalancé par des contraintes liées à l'algorithme appliqué pour récupérer auprès des nœuds constituant le système distribué les données puis les agré-

ger. Par la suite, nous dénoterons ces nœuds "pairs", car nous nous plaçons dans le cadre d'un système réparti de type pair-à-pair (ne présentant de ce fait aucun goulot d'étranglement). De nouvelles étapes, impliquant de contacter un certain nombre de pairs à travers un réseau de taille plus ou moins étendue, viennent s'ajouter au traitement habituel effectué pour une requête sur un système centralisé. Les pairs du système contiennent chacun une partie de l'ensemble des données à stocker. Chaque requête SPARQL (Prud'hommeaux & Seaborne, 2008) doit être divisée en sous-requêtes, qui sont ensuite exécutées sur les différents pairs du réseau, qui renvoient donc des résultats intermédiaires. Les données sont ensuite agrégées et filtrées pour que les résultats finaux soient retournés à l'utilisateur.

Un benchmark pour le RDF se compose d'un jeu de données RDF à insérer dans une structure de stockage et d'un jeu de requêtes SPARQL à exécuter. Une douzaine de ces benchmarks est actuellement répertoriée par le W3C [1]. L'utilisation d'un benchmark pour le RDF a pour but de valider et d'aider à optimiser les performances d'un système de stockage RDF précis. Cet article illustre notre démarche actuelle consistant à sélectionner un benchmark RDF pertinent pour notre système de stockage RDF. La configuration que nous avons choisie pour notre propre plateforme de stockage distribué (dénommée par la suite "Event Cloud") utilise l'infrastructure pair-à-pair CAN (Content Addressable Network) (Ratnasamy *et al.*, 2001) qui a, entre autres avantages, celui de faciliter l'insertion et la récupération des données auprès des nœuds du réseau. Comme nous allons l'expliquer dans cet article, le stockage RDF au sein de l'Event Cloud se base sur un CAN à 4 dimensions (les axes représentent respectivement la valeur de graphe, le sujet, le prédicat et l'objet d'un quadruplet RDF).

L'utilisation d'un benchmark pour le RDF a donc ici pour but de valider et d'optimiser le système de l'Event Cloud. Les performances relatives à la manipulation des données (temps pour les insérer et les récupérer via des requêtes SPARQL) sont mesurées pour nous permettre de porter une réflexion sur l'algorithme de résolution de requêtes optimal à appliquer dans le cadre d'un environnement distribué. Un benchmark nous permettra également de comparer nos performances avec celles d'autres architectures de stockage RDF, réparties ou non.

1. `http://www.w3.org/wiki/RdfStoreBenchmarking`

L'objectif de notre travail est donc de tenter de mettre en évidence les besoins concernant les benchmarks pour des architectures réparties et les aspects spécifiques à prendre en compte pour répondre à cette problématique, à savoir quel benchmark est-il pertinent de sélectionner pour évaluer un système de stockage RDF réparti. Dans la section 2, nous présentons la structure et l'algorithme de stockage et requêtage de l'Event Cloud. Dans les sections 3 et 4, nous présentons les raisons qui nous ont amenés à sélectionner le benchmark Berlin (Bizer & Schultz, 2009) et les conditions de son déploiement sur l'Event Cloud. Dans la section 5, nous présentons les résultats de notre exécution du benchmark. Enfin, la section 6 conclut cet article en mettant en avant les aspects qui nous semblent utiles de prendre en compte lors de l'élaboration d'un benchmark pour le RDF dans un contexte de stockage distribué.

2 Présentation de l'Event Cloud

L'Event Cloud [2] (Filali *et al.*, 2011) est un système de stockage RDF distribué en pair-à-pair gérant des requêtes SPARQL à travers son architecture. Sur l'Event Cloud, le support au stockage de données sémantiques de manière répartie se traduit par un réseau composé de pairs, chacun disposant du moteur Jena (Carroll *et al.*, 2004) pour y stocker des données du web sémantique (RDF) et exécuter des requêtes SPARQL. Les données sont stockées sous la forme de quadruplets (une valeur de graphe représentant l'origine du triplet est ajoutée aux classiques sujet, prédicat, objet). Le stockage peut s'effectuer soit en mémoire, soit de manière persistante avec écriture sur le disque.

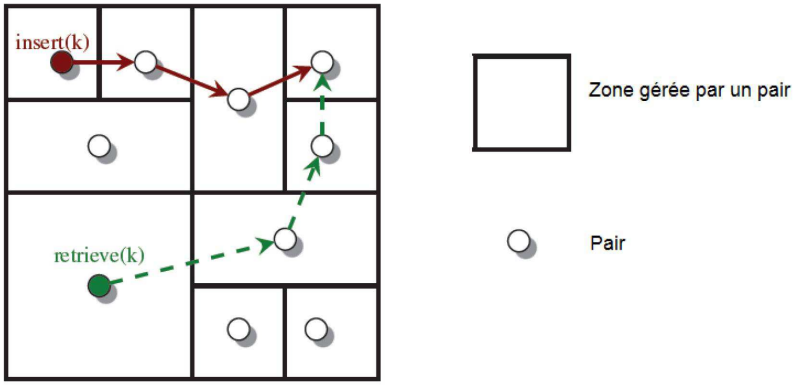
2.1 L'infrastructure CAN

Il existe plusieurs solutions techniques permettant de gérer des données RDF en pair-à-pair. Un état de l'art de ces différentes approches est décrit dans (Filali *et al.*, 2010). La configuration que nous avons choisie pour l'Event Cloud utilise l'infrastructure pair-à-pair CAN (Content Addressable Network). L'utilisation du pair-à-pair, et en particulier l'infrastructure distribuée de type CAN, guide la manière dont se fait la communication entre les nœuds du réseau et donc l'insertion et la récupération des données. Un CAN est une architecture pair-à-pair structurant un espace partitionné entre tous les pairs de telle sorte que chaque pair a la responsabilité de stocker toutes les données

2. "Event" car, ultimement, les données RDF représenteront des évènements, et "Cloud" car le système peut être déployé sur une infrastructure de type Cloud.

d'une zone de cet espace. Lors de l'insertion d'une nouvelle donnée dans le réseau, on utilise généralement une fonction de hachage permettant d'obtenir les coordonnées de la zone où insérer la donnée dans le CAN, et par conséquent le pair responsable de la zone concernée et qui la stockera. Comme le montre la Figure 1 (représentant un CAN à deux dimensions), la donnée en question sera routée depuis un pair d'origine pouvant être quelconque jusqu'à lui en passant d'un pair à son voisin ayant les coordonnées se rapprochant le plus de celles du pair responsable de la zone recherchée. Pour rechercher une donnée, la technique de routage sera similaire à celle appliquée pour l'insertion.

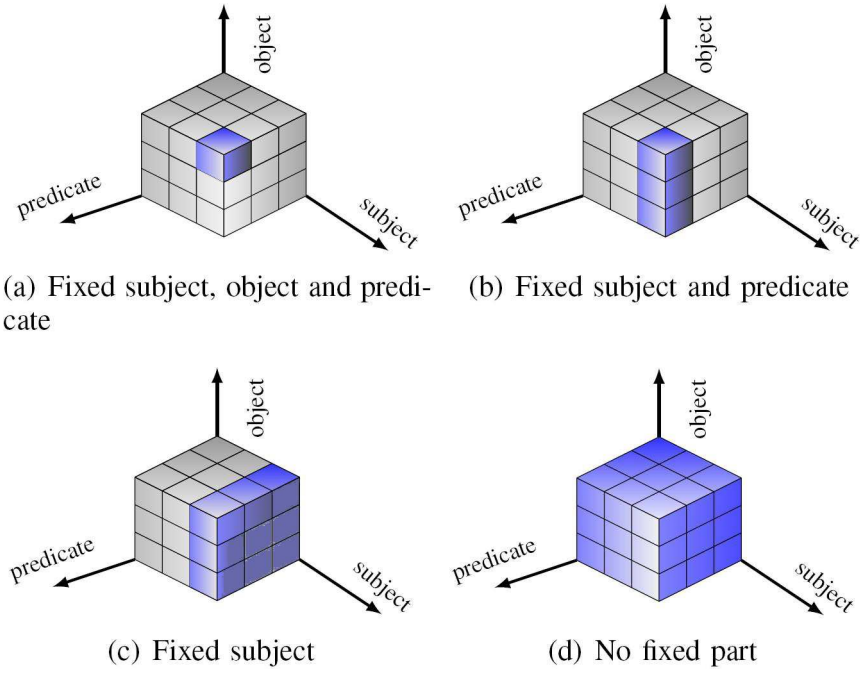
FIGURE 1 – Routage dans le CAN : insertion et récupération de données.



Les données RDF sont représentées par des URI (sous la forme d'un préfixe suivi d'une valeur). L'utilisation d'une fonction de hachage détruirait l'ordre naturel et disséminerait dans le réseau des informations ayant potentiellement des valeurs sémantiques très proches, augmentant considérablement le temps pour récupérer toutes les informations et donc le temps d'exécution d'une requête SPARQL sur ces données. C'est pour éviter cet inconvénient que le stockage RDF dans l'Event Cloud est implémenté en utilisant l'ordre lexicographique sur un CAN à quatre dimensions, chacune représentant respectivement la valeur de graphe, le sujet, le prédicat et l'objet. Pour plus de clarté, la Figure 2 et les explications données ci-après ne mentionnent que 3 axes : sujet, prédicat et objet. Les zones du CAN y sont délimitées selon 6 points : $z_{Sujet_{min}}$, $z_{Sujet_{max}}$, $z_{Predicat_{min}}$, $z_{Predicat_{max}}$, $z_{Objet_{min}}$ et $z_{Objet_{max}}$, les limites minimales et maximales de la zone d'un pair selon l'axe du sujet ($z_{Sujet_{min}}$, $z_{Sujet_{max}}$), l'axe du prédicat ($z_{Predicat_{min}}$, $z_{Predicat_{max}}$) et l'axe

de l'objet ($z_{Objet_{min}}$, $z_{Objet_{max}}$). Un triplet représente donc un point dans l'espace du CAN. Cette architecture permet de trouver facilement dans quelle(s) zone(s) du réseau un triplet recherché est susceptible d'être stocké, en se basant sur les valeurs fixées d'un triplet d'une sous-requête SPARQL.

FIGURE 2 – Portée d'un message en fonction des parties constantes de la requête.



(a) Fixed subject, object and predicate

(b) Fixed subject and predicate

(c) Fixed subject

(d) No fixed part

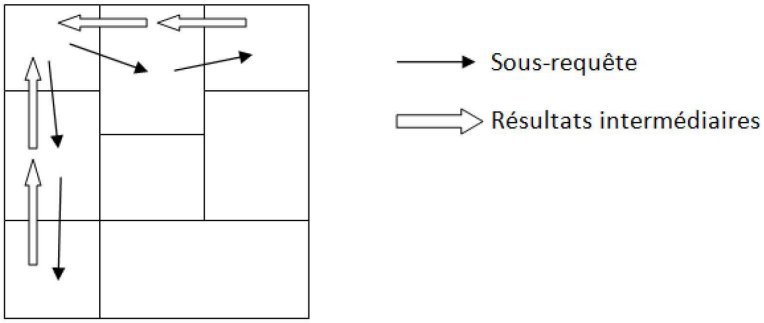
L'avantage de l'ordre lexicographique est qu'il préserve les informations sémantiques des données, ce qui permet de regrouper des triplets partageant le même préfixe (et ainsi obtenir une sorte de clustering). En d'autres termes, cette approche permet de stocker les triplets ayant des valeurs sémantiques proches sur des pairs étant proches dans le réseau.

En cas d'arrivée d'un nouveau pair dans le système, une des zones existantes du CAN sera découpée en deux zones et le nouveau pair sera responsable des données d'une de ces deux nouvelles zones. En cas de départ (non inopiné) d'un des pairs, les données se trouvant dans sa zone seront transférées vers un autre pair, dont la zone sera agrandie puisqu'elle aura englobé la zone du pair ayant quitté l'Event Cloud.

2.2 Résolution des requêtes

Le système étant réparti, la résolution d'une requête SPARQL implique de contacter l'ensemble des pairs susceptibles de stocker le résultat d'une partie de cette requête, c'est-à-dire d'une des conditions (un des triplets) de la clause WHERE. Un premier pair, celui auprès duquel la requête SPARQL a été déposée dans le système, découpe la requête initiale en sous-requêtes. Chacune d'entre elles équivaut à un triplet dont la valeur d'aucune, d'une, voire de plusieurs de ses trois parties est connue. Les sous-requêtes sont ensuite routées en parallèle à travers le réseau CAN vers le ou les pairs susceptible(s) de stocker des informations concordantes avec le triplet. Une fois la sous-requête arrivée sur le pair correspondant, celui-ci l'exécute et renvoie les résultats (dits résultats intermédiaires) vers le pair ayant initié la requête. Une fois tous les résultats intermédiaires récupérés, ils sont agrégés et d'éventuelles conditions (clause FILTER en langage SPARQL) sont appliquées pour enfin retourner les résultats de la requête initiale.

FIGURE 3 – Exemple de routage des sous-requêtes et résultats intermédiaires dans le CAN.



3 Déploiement d'un benchmark

Le benchmark que nous souhaitons utiliser pour valider le fonctionnement et les performances de l'Event Cloud doit répondre à certaines contraintes liées à notre architecture. Nous avons étudié quelques uns des benchmarks pour le RDF répertoriés par le W3C pour trouver celui susceptible de s'adapter le plus facilement à notre configuration. Nous avons finalement opté pour le benchmark Berlin (BSBM) car son générateur de données nous permet de

maîtriser le volume de données générées, et plusieurs formats de fichiers RDF (parmi lesquels N-Triples, Turtle, TriG) compatibles avec notre architecture sont proposés. Le jeu de requêtes proposé par BSBM contient certaines requêtes comportant des clauses actuellement non supportées par l'Event Cloud (OPTIONAL, fonction regex). Nous avons donc retiré ces requêtes de notre expérimentation pour ne conserver finalement que 4 requêtes (numéro 1, 5, 10 et 11) [3].

Le scénario suivi par notre utilisation du benchmark sur l'Event Cloud est le suivant : un fichier de données RDF est parsé et les quadruplets en sont extraits. Ils sont ensuite insérés dans le CAN (dont le nombre de pairs qui le composent est spécifié en paramètre), chacun étant routé jusqu'au pair devant le stocker. Nous mesurons le temps mis pour que tous les quadruplets soient insérés. Puis, nous exécutons la liste de requêtes SPARQL. Nous comptons ensuite le nombre de résultats retournés, le nombre de résultats intermédiaires ayant transité sur le réseau pour permettre de résoudre chaque requête, le temps d'exécution des sous-requêtes sur le moteur de stockage et le temps total pour que les résultats de la requête arrivent jusqu'à l'utilisateur. Ces mesures ont un rôle important pour l'évaluation des performances d'une architecture de stockage RDF répartie. Elles peuvent nous permettre de détecter une anomalie au niveau de la communication entre les pairs ou de l'algorithme de résolution des requêtes. Nous pouvons également porter une réflexion sur le moteur de stockage à adopter si nous constatons que le temps pour exécuter les sous-requêtes prend une place trop importante par rapport au temps total pour retourner les résultats.

4 Environnement expérimental

Les jeux de données utilisés durant notre expérimentation ont été obtenus grâce au générateur de données de BSBM, configuré pour générer des fichiers au format TriG contenant respectivement 1000, 2000, 3000 et 4000 produits, ce qui équivaut à la génération de 299119, 583487, 867794 et 1160279 quadruplets. Les requêtes de BSBM utilisées ici sont les requêtes numéro 1, 5, 10 et 11 (qui sont toutes des requêtes interrogatives). Notre expérimentation a été lancée sur une machine avec une JVM de 10 Go (option server), avec une structure de stockage composée de 20 pairs.

3. `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/ExploreUseCase/index.html#queries`

5 Analyse des résultats

Produits	Quadruplets	Rés. intermédiaires	Volume échangé	Rés. finaux
1000	299119	115635	43 Mo	9
2000	583487	221315	85 Mo	8
3000	867794	327192	125 Mo	9
4000	1160279	439424	170 Mo	8

TABLE 1 – Récapitulatif des résultats obtenus, exprimés en nombre de données (quadruplets) RDF.

Les résultats de la Table 1 montrent qu'en moyenne, l'exécution des 4 requêtes du benchmark provoque le transfert entre les pairs d'environ 38% du total des données stockées dans l'Event Cloud, en tant que résultats intermédiaires. Dans notre expérience avec 1160279 quadruplets, cela représente un volume de 170 Mo de données intermédiaires qui transitent entre les pairs. Au niveau d'une architecture distribuée, ces chiffres impactent inévitablement les temps de récupération des résultats finaux à cause du routage de pair en pair des résultats intermédiaires jusqu'au pair initiateur (cf. Figure 3). En moyenne, pour chacun des tests de notre expérimentation, le temps d'exécution des sous-requêtes sur le moteur de stockage des pairs représente 1,5% du temps total écoulé entre l'envoi de la requête et l'affichage des résultats. Ce qui signifie que 98,5% du temps est passé à transférer entre les pairs des sous-requêtes et des résultats intermédiaires puis les agréger. Plus le volume de données insérées dans l'Event Cloud augmente, plus cette tendance s'accentue, les transferts entre les pairs atteignant 99,3% du temps pour l'expérience avec 1160279 quadruplets.

Afin de constater le déséquilibre qui peut être provoqué par le nombre variable de résultats intermédiaires en fonction de la nature d'une sous-requête, nous allons détailler l'exécution d'une requête de BSBM sur l'Event Cloud, en l'occurence la requête numéro 5 (cf. Figure 4). Lorsque cette requête est envoyée sur l'Event Cloud, elle est tout d'abord divisée par un premier pair en 7 sous-requêtes, chacune de ces sous-requêtes étant ensuite routée vers le ou les pairs susceptibles de stocker des données matchant cette sous-requête. Au final, chaque pair contacté renvoie au pair initiateur les résultats obtenus (dits résultats intermédiaires) pour une sous-requête. Lors de notre expérimentation pour 1160279 quadruplets stockés, répartis sur 20 pairs, 103932

FIGURE 4 – Requête numéro 5 de BSBM utilisée sur l'Event Cloud.

```
SELECT DISTINCT ?product ?productLabel
WHERE { GRAPH ?g {
      ?product rdfs:label ?productLabel .
      FILTER (dataFromProducer1:Product1 != ?product)
      dataFromProducer1:Product1 bsbm:productFeature
?prodFeature .
      ?product bsbm:productFeature ?prodFeature .
      dataFromProducer1:Product1
bsbm:productPropertyNumeric1 ?origProperty1 .
      ?product bsbm:productPropertyNumeric1
?simProperty1 .
      FILTER (?simProperty1 < (?origProperty1 + 120)
&& ?simProperty1 > (?origProperty1 - 120))
      dataFromProducer1:Product1
bsbm:productPropertyNumeric2 ?origProperty2 .
      ?product bsbm:productPropertyNumeric2
?simProperty2 .
      FILTER (?simProperty2 < (?origProperty2 + 170)
&& ?simProperty2 > (?origProperty2 - 170))
} }
ORDER BY ?productLabel
LIMIT 5
```

résultats intérmédiaires ont ainsi été renvoyés vers le pair initiateur de la requête numéro 5 de BSBM. La Table 2 résume le nombre de résultats reçus pour chacune des sous-requêtes. Il est possible de constater que ce sont les sous-requêtes dont seule la valeur de prédicat est connue qui génèrent le plus de données intermédiaires, en particulier la sous-requête numéro 3 qui génère à elle seule près de 82 % de la totalité des résultats intermédiaires. Une fois tous les résultats intermédiaires reçus par le pair initiateur de la requête, celui-ci les agrège et se charge d'appliquer les fonctions de filtre (FILTER), pour ne retourner finalement aucun résultat matchant l'intégralité de la requête.

Il apparaît donc que la gestion des résultats intermédiaires représente un des facteurs majeurs à appréhender lors de la résolution de requêtes sur un

Num.	Sous-requête	Résultats
1	`?g ?product rdfs:label ?productLabel`	10740
2	`?g dataFromProducer1:Product1 bsbm:productFeature ?prodFeature`	20
3	`?g ?product bsbm:productFeature ?prodFeature`	85170
4	`?g dataFromProducer1:Product1 bsbm:productPropertyNumeric1 ?origProperty1`	1
5	`?g ?product bsbm:productPropertyNumeric1 ?simProperty1`	4000
6	`?g dataFromProducer1:Product1 bsbm:productPropertyNumeric2 ?origProperty2`	1
7	`?g ?product bsbm:productPropertyNumeric2 ?simProperty2`	4000

TABLE 2 – Nombre de résultats (quadruplets RDF) obtenus sur l'Event Cloud pour chaque sous-requête de la requête 5 de BSBM.

système de stockage distribué. Or, à l'heure actuelle, aucun benchmark pour le RDF ne présente une estimation des résultats intermédiaires que ses requêtes génèrent. Nous pensons qu'il serait pertinent de considérer davantage cet aspect, qui joue un rôle majeur dans un contexte réparti. Notamment pour des benchmarks utilisant un jeu aux données volumineuses, dont le transfert entre les nœuds du réseau pourrait se révéler particulièrement coûteux. Les quadruplets extraits de BSBM ont l'avantage d'être d'une taille raisonnable et n'entravent donc pas excessivement les temps pour récupérer les résultats. Mais ces mesures seraient probablement impactées pour un test effectué avec un benchmark dont les triplets ont une taille très volumineuse. Nous pensons qu'il pourrait donc être judicieux à l'avenir de porter une réflexion sur la structuration des requêtes SPARQL des benchmarks pour le RDF. Charge ensuite aux solutions benchmarkées d'optimiser éventuellement le support de ces requêtes dans le but de réduire le nombre de résultats intermédiaires générés, ce qui permettrait d'améliorer sensiblement les performances. De façon plus générale, la constitution des benchmarks pourrait gagner à proposer, pour une même requête SPARQL, plusieurs optimisations possibles associées dans le but de générer des volumes de résultats intermédiaires différents.

6 Conclusion

Dans cet article, nous avons présenté le fonctionnement d'un système de stockage en pair-à-pair pour le RDF. Dans le cadre de ce système, nous avons tenté de mettre en avant les aspects qui nous semblent les plus pertinents afin qu'un benchmark soit utilisable sur une telle architecture de stockage RDF distribuée. Les mesures enregistrées sur l'Event Cloud avec le benchmark Berlin nous ont permis de constater l'important volume de résultats intermédiaires généré par les requêtes. Nous avons montré que, dans un contexte de stockage distribué, le transfert d'informations entre les différents nœuds du réseau représente la part la plus importante du temps mis pour récupérer les résultats d'une requête. La solution d'un système distribué étant la meilleure option à l'heure actuelle pour supporter le passage à l'échelle, nous pensons donc que cet aspect devrait être davantage pris en compte lors de l'élaboration d'un benchmark pour le RDF.

Références

BIZER C. & SCHULTZ A. (2009). The berlin sparql benchmark.

CARROLL J., DICKINSON I., DOLLIN C., REYNOLDS D., SEABORNE A. & WILKINSON K. (2004). Jena : implementing the semantic web recommendations. p. 74–83.

FILALI I., BONGIOVANNI F., HUET F. & BAUDE F. (2010). RDF Data Indexing and Retrieval : A survey of Peer-to-Peer based solutions. Number RR-7457.

FILALI I., PELLEGRINO L., BONGIOVANNI F., HUET F. & BAUDE F. (2011). Modular P2P-based Approach for RDF Data Storage and Retrieval. In *Proceedings of The Third International Conference on Advances in P2P Systems (AP2PS 2011)*.

KLYNE G., CARROLL J. & MCBRIDE B. (2004). Resource description framework (RDF) : Concepts and abstract syntax.

PRUD'HOMMEAUX E. & SEABORNE A. (2008). SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`.

RATNASAMY S., FRANCIS P., HANDLEY M., KARP R. & SCHENKER S. (2001). A scalable content-addressable network. **31**(4), 161–172.