Web Assembly pour le Web sémantique

Julian Bruyat
Encadré par Pierre-Antoine Champin

Université Claude Bernard Lyon 1, France

Résumé L'augmentation du nombre de machines connectées et de la quantité de données disponibles est un contexte favorable au Web sémantique, dont l'objectif est d'automatiser le traitement de ces données. Dans ce mémoire, nous nous intéressons à ce que peuvent apporter Rust et Web Assembly, des langages compilés, pour proposer de nouvelles implémentations du Web sémantique qui sont utilisables en Javascript ainsi que les avantages et inconvénients de créer une interopérabilité entre les bibliothèques de deux langages très différents. Nous nous intéressons plus particulièrement aux interfaces proposées par Sophia en Rust et RDF.JS en Javascript, que nous améliorons en proposant de nouvelles implémentations basées sur des B-Arbres pour améliorer la vitesse de recherche de quads répondant à un motif précis.

Mots-clés : Web Assembly · Web sémantique · Indexation

Abstract The constant increase of connected machines and volume of available data is a beneficial context for the semantique Web, whose goal is to process automatically a lot of data. In this master thesis, we study Rust and Web Assembly, which are compiled languages, and how they can contribute to implement better semantic Web libraries for Javascript. We also study the benefits and drawbacks of ensuring interoperability between their main available toolkits : Sophia for Rust and RDF.JS for Javascript. For both toolkits, we propose new implementations based on B-Trees to improve pattern matching speed.

Keywords: Web Assembly · Semantic Web · Indexing

1 Introduction

Le Web sémantique est une extension du Web classique visant à permettre aux machines d'exploiter les connaissances disponibles sur internet. Le principal modèle de données utilisé est RDF [22] (*Resource Description Framework*), un standard du W3C visant à décrire les connaissances sous la forme de graphes, dont les noeuds et les arcs sont étiquetés, le plus souvent par des IRI identifiant les ressources concernées.

Aujourd'hui, un grand nombre de bibliothèques et d'applications se reposant sur RDF existe. Certaines d'entre elles comme N3.js sont écrites en Javascript ce qui permet de les utiliser sur un serveur Web mais également chez le client à travers

son navigateur. Pour améliorer les performances des applications serveurs, utiliser des programmes dans des langages compilés comme le C est une pratique courante. Mais avant l'arrivée récente de Web Assembly, il n'était pas possible pour un serveur Web de faire exécuter du code compilé au client. Cette technologie ouvre la possibilité de déployer à la fois sur client et serveur de nouvelles applications RDF compilées en amont, donc plus performantes que des solutions dont le code est interprété.

1.1 Conditions du stage

Ce stage a été effectué au sein du LIRIS (Laboratoire d'Informatique en Images et Systèmes d'Information) et plus particulièrement au sein de l'équipe TWEAK (Traces, Web, Education, Adaptation and Knowledge). Il s'est déroulé du 3 février au 31 juillet 2020 et a été encadré par Pierre-Antoine Champin.

Le stage s'inscrit dans le projet REPID (*A Reasoner that is Efficient, Portable, Incremental and Distributed*), un projet commun entre le LIRIS et le laboratoire Hubert Curien à Saint Etienne. Comme son nom l'indique, l'objectif du projet REPID est de proposer des outils pour la réalisation d'un raisonneur RDF. Parmi les précédentes réalisations, on retrouve HylaR [26], un raisonneur écrit en Javascript exécutant des tâches de raisonnement soit sur le serveur, soit sur le client selon les performances de chacun ; Inferray [24], un moteur d'inférence efficace, originellement écrit en Java et qui a été porté en Rust [9] ; et enfin Sophia [12], une bibliothèque visant à offrir une API commune pour l'implémentation de bibliothèques RDF en Rust.

1.2 Objectifs

Un des objectifs de REPID est de permettre aux outils présentés d'être interopérables. Néanmoins, le décalage entre les différents langages (en particulier entre Javascript d'une part, un langage interprété s'exécutant sur navigateur, utilisé par HylaR, et Rust d'autre part, un langage compilé qui est utilisé pour développer la bibliothèque Sophia) est un obstacle.

Web Assembly [16] est un standard conçu par Mozilla, Google, Microsoft, Apple et le W3C permettant d'exécuter du code compilé dans un navigateur. De plus, Rust, un langage poussé par Mozilla [1] possède à ce jour la chaîne de compilation la plus aboutie pour compiler un programme codé dans un langage haut niveau vers du Web Assembly : wasm_bindgen [4].

Nous nous intéressons ici aux questions suivantes :

— Peut-on utiliser Web Assembly pour obtenir de meilleures performances que les bibliothèques déjà existantes en Javascript pour le Web sémantique ?

1. Mozilla est une organisation orientée vers le Web dont le produit le plus connu est le navigateur Mozilla Firefox. Il n'est donc pas surprenant que le langage Rust, également développé par Mozilla, entretienne une relation étroite avec Web Assembly.

— Quels sont les apports et les contraintes liés à l'utilisation de Rust et de Sophia ? Ces outils sont-ils des choix adaptés ?

La section 2 présente les différentes technologies utilisées, en particulier le Web sémantique, le langage Rust, le Web Assembly et les liens entre elles. La section 3 présente les différentes solutions qui ont été implémentées pour faire converger le Web Assembly et le Web sémantique. La section 4 présente les mesures faites des solutions proposées, ainsi que les analyses que nous pouvons en faire, et qui ont motivé les contributions suivantes afin de gagner en performances. Enfin, nous en tirons une conclusion dans la section éponyme 5.

1.3 Positionnement par rapport au Master Intelligence Artificielle

Ce travail s'inscrit aux frontières des domaines de l'intelligence artificielle, des technologies Web et de la gestion de données, de la compilation et du génie logiciel :

— on se positionne ici dans le cadre du Web sémantique, qui est un sous-domaine de l'intelligence artificielle symbolique. Bien que nous ne contribuions pas directement aux aspects raisonnements du Web sémantique, ce travail est une phase préliminaire afin de permettre à des travaux passés et futurs de bénéficier à l'écosystème du Web sémantique, que ce soit pour des tâches centralisées (amélioration des performances d'un serveur qui pourra alors traiter plus de requêtes) ; mais également de manière distribuée en fournissant une solution plus efficace (car compilé en un *bytecode* proche du code machine) et plus flexible que les solutions existantes (car légèrement réinterprété) [2] ;
— des technologies Web et de la gestion de données car le Web sémantique s'intéresse à la manipulation d'un grand nombre de connaissances partagées, dont la sémantique et l'utilisation sont profondément liées au Web : il est possible à tout moment de déréférencer une IRI (consulter le contenu pointé par celle-ci) ce qui entre alors dans le cadre d'utilisation classique du Web ;
— de la compilation car Web Assembly est une technologie récente, et les écarts entre un langage interprété avec *garbage collector* et un langage compilé sans *garbage collector* ont de profondes implications dans la conception et l'interprétation des expérimentations et de leurs résultats ;
— enfin en s'inscrivant dans une démarche d'interopérabilité entre plusieurs langages et interfaces, et d'élaboration de solutions visant à tirer profit des différentes technologies utilisées, on entre dans le domaine du génie logiciel.

Ainsi, si les questions de recherche se centrent sur des domaines annexes, les implications en termes d'ingénierie ont des répercussions sur le domaine de l'intelligence artificielle symbolique.

2. C'est dans ce sens qu'est actuellement développé l'outil Wasmtime dont l'objectif est de permettre d'exécuter directement du code Web Assembly, sans l'intermédiaire de Javascript, que ce soit sur des machines variées peu puissantes (contexte du *Web des objets*) ou des serveurs puissants

2 État de l'art

Nous travaillons dans le domaine du Web sémantique, et plus particulièrement de ce que Rust et Web Assembly peuvent lui apporter. Nous allons ici décrire ces différentes notions et technologies.

2.1 Le Web sémantique

2.1.1 Graphes et triplets RDF

RDF [13] est langage de représentation de connaissances. Il se repose sur un modèle de graphe où les noeuds et les arcs sont étiquetés par des termes. Dans ce modèle, le nœud de départ d'un arc est nommé **sujet**, le terme étiquetant l'arc est nommé **prédicat** et le nœud d'arrivée est nommé **objet**.

Les termes possibles dans un graphe sont :

— les IRI [14] (ou *Internationalized Resource Identifier*), qui sont une évolution des URL. Les IRI permettent de décrire une ressource de manière non ambiguë, la sémantique étant choisie par le propriétaire de l'IRI. Leur utilisation ayant tendance à produire des noeuds avec un identifiant long, il est fréquent d'utiliser des préfixes pour abréger les IRI, comme par exemple `rdf:` qui est souvent utilisé comme raccourci de `http://www.w3.org/1999/02/22-rdf-syntax-ns\#` ;

— les littéraux, uniquement en position objet d'un triplet. Ils permettent de décrire littéralement une connaissance (par exemple la chaîne de texte « Lama », la chaîne « Lama » en français (la langue faisant partie du terme et faisant que ce terme est différent du précédent), le nombre entier 3 (écrit avec la chaîne « 3 » et une IRI décrivant le type nombre entier) ou encore le nombre flottant 7,77 (écrit avec la chaîne « 7.77 » et une IRI décrivant le type flottant) ;

— les noeuds anonymes ou vierges, qui ne possèdent pas un identifiant défini par l'utilisateur. Comme leur nom l'indique, ils ne peuvent pas être utilisés pour décrire un arc. Ils permettent de décrire une connaissance sans la nommer explicitement.

2.1.2 Dataset et quad RDF

Un **dataset** est un ensemble de **graphes**. Il est constitué d'un graphe appelé « graphe par défaut » et peut être complété par d'autres graphes qui sont identifiés par des IRI ou des nœuds vierges, appelés « graphes nommés ».

Dans ce modèle, on ne parle plus de triplet mais de quad : un quad étant un triplet dans un graphe particulier. Une autre manière de le voir est de dire qu'un quad est désigné par son sujet, son prédicat, son objet et le nom du graphe auquel il appartient.

```
@prefix :     <https://www.bruy.at/>
@prefix rdf:
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix pac:  <https://www.champin.net/>
```

```
<https://www.bruy.at/julian> a foaf:Person.
pac:pierre-antoine a foaf:Person.
:julian foaf:knows pac:pierre-antoine

:croyance {
  :julian foaf:knows :julian
}
```

Listing 1. Un exemple de dataset au format TriG

Dans le dataset décrit au format TriG [23] par le listing 1, la ressource décrite par l'IRI `https://www.bruy.at/julian` a pour type (`a` étant un raccourci pour `rdf:type`) la ressource décrite par `http://xmlns.com/foaf/0.1/Person`, ce qu'un humain peut interpréter comme étant le fait que « Julian Bruyat » est une personne. Aucun graphe n'étant indiqué pour ce quad, il est dans le graphe par défaut.

Le dernier quad décrit par ce même listing est le fait que dans le graphe `https://www.bruy.at/croyance`, il existe un triplet indiquant que la ressource décrite par `https://www.bruy.at/julian` se connaît elle-même.

La notion de graphe nommé a d'abord été définie formellement par Caroll et al [11]. Elle a ensuite été standardisée avec RDF 1.1 en 2014 [21] dans une version plus relâchée afin de se conformer aux usages qui en étaient faits, qui ne respectaient pas toujours la sémantique stricte qui en était proposée. Ainsi, si leur sémantique n'est pas toujours la même selon les datasets, les graphes nommés sont souvent utilisés pour fournir des informations sur le contexte ou la provenance d'un triplet.

2.1.3 Indexation

Lors de la conception d'une structure de données visant à représenter un graphe ou un dataset, il existe deux manières de procéder :

— la première est de stocker directement dans la structure de données les termes. C'est ce que fait par exemple Graphy [3] ;
— la seconde est de stocker d'un côté une table de correspondance entre les termes et des index sous forme de valeur entières, et de stocker dans la structure de données les index. C'est l'approche utilisée par HDT [15] (pour *Header Dictionnary Triples*), un format de stockage compact de graphes.

La seconde méthode est plus compacte en mémoire et facilite des opérations comme le tri, mais impose de faire la correspondance entre les index et les termes lorsque l'on souhaite retrouver la sémantique d'un terme, par exemple pour l'afficher à l'utilisateur.

3. Graphy (https://www.graphy.link) est une librairie permettant de manipuler des datasets écrite en Javascript par Blake Regalia depuis décembre 2018

2.2 Rust

Rust [19] est un langage initié par Graydon Hoare et soutenu par Mozilla. Il est développé depuis juillet 2010.

Il s'agit d'un langage impératif de bas niveau comme C. Il n'y a pas de notion d'héritage. En revanche contrairement à C, les structures peuvent contenir des méthodes. Rust intègre un système de traits, semblable aux interfaces en Java, permettant de spécifier qu'une structure implémente certaines méthodes. L'avantage de cette démarche est de permettre aux programmeurs de garder les habitudes de la programmation objet pour le découpage du code tout en ayant une abstraction n'ayant aucun coût à l'exécution.

Rust a pour objectif d'être un langage dont les programmes sont à la fois sûrs (grâce au système d'emprunts décrit dans l'annexe B) et rapides.

2.3 Web Assembly

Les raisons et la démarche ayant amené à la conception de Web Assembly sont décrites dans l'annexe A. Aujourd'hui, bien que les navigateurs Web intègrent un JIT (*Just In Time Compiler*) permettant d'améliorer grandement les performances du code Javascript, c'est dans un contexte où le nombre d'applications sur navigateur est en constante augmentation que Web Assembly est développé depuis 2015.

Son objectif est de permettre d'exécuter du code écrit en assembleur depuis le navigateur. Cela permet de proposer aux utilisateurs des fichiers plus petits (des fichiers binaires en Web Assembly au lieu de fichiers textuels en Javascript) et dont la compilation est faite *Ahead of time*. Dans la pratique, afin d'être à la fois le plus performant possible tout en pouvant être exécuté sur n'importe quel processeur et donc garder la portabilité du Javascript, le code Web Assembly est exécuté sur une machine virtuelle dont les instructions sont pensées pour être très proches des instructions machines réelles.

Afin de garantir la sécurité, le code Web Assembly ne peut travailler que sur une **mémoire linéaire** qui lui est fourni. Cette mémoire linéaire est fournie par le code Javascript sous la forme d'un tableau d'entiers qu'il pré-alloue. Les deux environnements ne communiquent que par l'intermédiaire d'appels de fonctions et la modification de ce tableau.

2.4 L'outil wasm_bindgen

Comme nous venons de le voir, la machine virtuelle de Web Assembly ne peut travailler que sur un tableau de nombres. Cela implique que Web Assembly est très performant pour faire des calculs dont le résultat final est un nombre ou un ensemble de nombres (par exemple une liste de pixels à afficher). Pour des applications plus complexes, par exemple pour pouvoir travailler en Javascript avec des structures de données, comme des datasets RDF, nous avons besoin d'un niveau d'abstraction supplémentaire.

Grâce à WASM_BINDGEN, RUST est le seul langage à proposer de compiler vers du WEB ASSEMBLY en permettant à JAVASCRIPT de manipuler des objets. Pour cela, il utilise une surcouche de code JAVASCRIPT qui permet de faire le lien entre les objets que l'on souhaite manipuler et leur représentation dans la mémoire interne de WEB ASSEMBLY.

David POJUNAS, lors de son stage, a effectué un premier travail d'exploration des différentes possibilités offertes par WASM_BINDGEN et de comment elles pourraient être mises en oeuvre dans le cadre de SOPHIA (en proposant une implémentation d'une structure simple permettant d'ajouter des quads gérés par SOPHIA et de les lister).

2.5 Sophia

SOPHIA [12] est une bibliothèque écrite en RUST développée par Pierre-Antoine CHAMPIN depuis 2018.

Son objectif est de proposer des traits (notion définie dans la section 2.2) qui permettent d'unifier la manière de manipuler des datasets entre différentes implémentations possibles. Afin de faciliter son utilisation, des implémentations concrètes de ces traits sont également disponibles, comme un tableau de quads ou le `FastDataset` que nous allons présenter dans la sous-section suivante.

2.5.1 Le FastDataset de SOPHIA

Une première implémentation d'un ensemble de quads proposée par défaut par SOPHIA est le `LightDataset`. Il est implémenté comme un `HashSet` de quads. Cette structure ne permet néanmoins pas de retrouver rapidement des quads correspondant à un certain motif, tel que « retrouver tous les quads ayant pour sujet `https://www.bruy.at/julian` dans le graphe par défaut » : il faut parcourir tous les quads et filtrer ceux qui ne sont pas désirés.

Afin d'améliorer le temps de réponse de certaines requêtes, le `FastDataset` utilise deux structures supplémentaires spécialisées pour y répondre. La figure 1 présente le système de composition de ses différents *wrappers*. Ceux-ci indexent pour chaque préfixe de quad (dans un ordre donné) les quads correspondants. Par exemple, le `OGPSWrapper` associe pour tous les objets la liste de tous les quads contenant cet objet. Cette indexation se fait sur plusieurs niveaux. Plus spécifiquement, au lieu d'associer tous les objets, il associe tous les graphes, et c'est la paire objet - graphe qui est associée à la liste de tous les prédicats (signifiant qu'il existe au moins un quad contenant le prédicat, l'objet et le graphe, dont il est possible de récupérer la liste des sujets pour finir de reconstruire les quads). Le terme `FastDataset` n'est en réalité qu'un raccourci pour désigner un `Dataset` de type `OGPSWrapper<GPSOWrapper<LightDataset>>`, un utilisateur de SOPHIA étant libre d'imbriquer comme il le souhaite les *wrappers*, et même d'en créer de nouveaux.

Lors du traitement d'une requête, chaque *wrapper* s'interroge sur sa capacité à y répondre de manière optimale. Dans notre exemple où nous cherchons la liste de tous les quads dont le sujet est `https://www.bruy.at/julian` qui sont

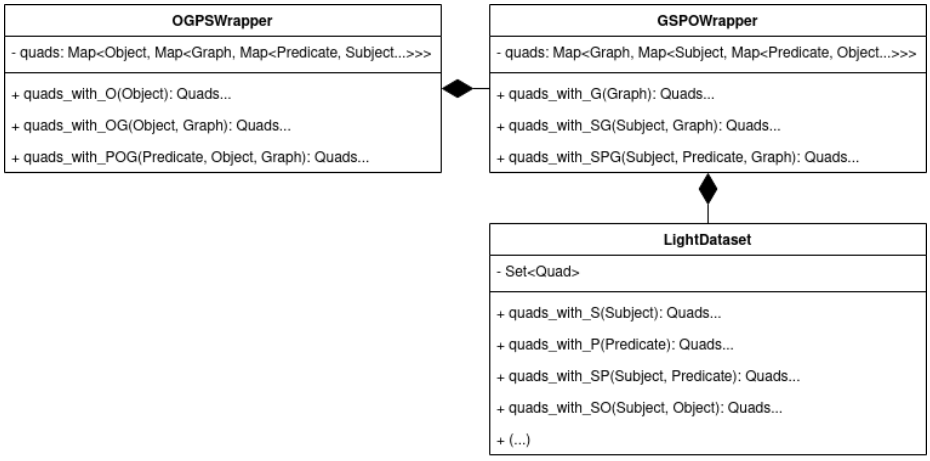

Figure 1. Les éléments composant un FastDataset afin de répondre plus rapidement à certaines requêtes. À noter qu'un quadruplet `(Subject, Predicate, Object, Graph)` est un `Quad`.

dans le graphe par défaut, l'`OGPSWrapper` est incapable d'y répondre de manière optimale car il doit connaître l'objet : il demande donc au dataset qu'il contient (le `GPSOWrapper<LightDataset>`) de traiter la requête. La `Map` de `GSPOWrapper` permettant de trouver directement le graphe (le graphe par défaut) et le sujet (`https://www.bruy.at/julian`) dans sa structure, puis d'itérer sur les quads en retrouvant les prédicats et les objets associés à ce prédicat, c'est la fonction `quads_with_sg` de celui-ci qui est utilisée, ignorant l'implémentation naïve faite par `LightDataset`.

Notons que dans cette implémentation, les *wrappers* ne savent pas quelles sont les structures qu'ils contiennent, ce qui a des implications que la section 3.2.1 développe.

2.6 Une proposition d'interface unifiée pour les Datasets RDF en Javascript : RDF.JS

L'objectif du stage est d'utiliser Web Assembly afin de faire, à terme, du raisonnement. Néanmoins, l'implémentation d'un moteur de raisonnement est une tâche complexe. Plutôt que de nous intéresser à l'implémentation en elle-même d'un moteur, nous nous intéressons plutôt à l'interopérabilité de différents moteurs.

Un groupe communautaire s'est réuni afin de proposer une interface standard pour les bibliothèques de Web sémantique utilisables en Javascript. Cette spécification détermine les méthodes qui doivent être implémentées pour les termes, les datasets et les stores (un store étant un dataset dont les méthodes sont prévues pour une utilisation asynchrone). Nous nommons ici « RDF.JS » le

regroupement de ces trois spécifications.

Le point fort de RDF.JS est de permettre à plusieurs bibliothèques indépendantes de pouvoir se reposer sur d'autres bibliothèques. Ce choix a été fait en opposition à d'autres langages comme Java où le Web sémantique est dominé par quelques bibliothèques monolithiques comme Jena [18]. Par ailleurs, Sophia a été conçu dans un but similaire à RDF.JS : devenir une interface commune à toutes les implémentations en Rust de `Graph` et de `Dataset`.

Un exemple de logiciel tirant profit de RDF.JS est Comunica [25]. Celui-ci se donne pour objectif d'être hautement modulaire, et pour ce faire, permet d'exécuter des requêtes SPARQL [17] en se reposant sur n'importe quelle implémentation d'un store RDF.JS. Plus généralement, permettre à n'importe quel dataset implémenté en Rust de pouvoir être utilisé en Javascript via une implémentation en Web Assembly de RDF.JS permettrait de mutualiser les efforts des deux communautés, d'autant plus si les performances des bibliothèques compilées de Rust vers Web Assembly offrent de meilleures performances.

3 Contributions techniques et choix d'implémentations

Notre première contribution technique est la mise en place d'une architecture permettant d'exporter, grâce à Web Assembly, n'importe quelle implémentation du trait `Dataset` de Sophia vers le monde Javascript. La seconde est l'implémentation d'un nouveau type de `Dataset` se reposant sur plusieurs B-Arbres et une indexation paresseuse afin de pouvoir répondre à n'importe quelle requête sans parcourir tous les quads. La dernière est la réalisation d'une autre approche pour l'implémentation d'un dataset répondant à la spécification de RDF.JS Dataset [3], mais également d'un Store (spécifié par la spécification RDF.JS Stream [2]), se reposant sur une architecture mixte entre Rust / Web Assembly et Javascript, mais délaissant Sophia.

Dans la suite de ce rapport, lorsque nous parlons d'une application ou une librairie utilisable avec Javascript, mais que nous sommes agnostiques sur l'utilisation ou non de Web Assembly, nous dirons que cette application ou librairie est à destination d'une **plate-forme Web** [4].

3.1 Adaptateur de l'interface Sophia vers l'interface RDF.JS

L'interface RDF.JS propose d'implémenter deux classes *Factory*. La première est dans le *Data Model* [1] et permet de construire des termes à partir de chaînes de caractères et des quads à partir de termes. La seconde, qui est dans l'interface

4. Cette désignation est inspirée de la notion de *Open Web Platform* du W3C (World Wide Web Consortium) qui englobe toutes les technologies utilisables sur le Web. Elle inclut bien sûr Javascript et Web Assembly mais également HTML, CSS, le format d'image SVG et même des protocoles comme le protocole HTTP.

Dataset Model, permet d'instancier des datasets vide ou à partir d'une séquence de quads.

Sophia a une structure similaire dans le fait qu'on y trouve également une structure pour les termes [5], et des datasets construits à partir de ces termes. L'implémentation du *data model* de RDF.JS à partir de Sophia est triviale et ne sera pas documentée ici.

La figure 2 présente la méthode que nous utilisons pour exporter vers des plates-formes Web des datasets issus de Sophia :

— l'élément de base est un `Dataset` implémentant le trait correspondant dans Sophia ;

— nous demandons au développeur d'englober cet objet dans une nouvelle structure implémentant le trait `ExportableDataset` (suivant ainsi le patron de conception « adaptateur »). Ce trait, qui peut être vu comme étant ce que serait le trait `Dataset` de RDF.JS s'l avait été implémenté en Rust, propose des implémentations par défaut de toutes les méthodes requises. Le développeur n'a besoin que d'implémenter trois méthodes : `get_dataset() -> Dataset` qui renvoie une référence immutable vers le dataset à exporter, `get_mutable_dataset() -> Dataset` qui renvoie une référence mutable vers le dataset à exporter et enfin `wrap(Dataset) -> Self` [6] qui permet de construire une nouvelle instance englobant le dataset. Toutes les autres méthodes sont implémentées à partir de ces trois méthodes, et en utilisant le fait que le dataset obtenu par les méthodes `get` implémente le trait `Dataset` de Sophia [7].

— une macro prend en paramètre un `ExportableDataset`, construit une nouvelle structure qui contient une instance de cet objet et se contente de créer des fonctions annotées avec wasm_bindgen qui appellent les fonctions de l'objet englobé.

— Nous fournissons également une macro prenant une structure implémentant le trait `Dataset` de Sophia et créant automatiquement les deux structures englobantes et les liens wasm_bindgen (dans le cas ou le développeur ne souhaite surcharger aucune implémentation par défaut de `ExportableDataset`).

Cette approche a les avantages suivants :

— cette implémentation se repose uniquement sur du code écrit en Rust qui est compilé en Web Assembly : on peut donc profiter de toutes les optimisations actuelles et à venir des outils utilisés (wasm_bindgen, Web Assembly, Rust et Sophia) ;

5. Plus exactement au début du projet une énumération, sachant que les énumérations en Rust peuvent contenir des attributs. Depuis, un *refactoring* de Sophia a été fait et les termes sont désormais des traits dont l'ancienne énumération n'est qu'une implémentation.

6. `Self` signifie « du même type que l'objet dont on a appelé la méthode »

7. Ces implémentations étant faites dans l'implémentation du trait `ExportableDataset`, il est possible pour un développeur d'implémenter des versions moins naïves utilisant les particularités du dataset à exporter.

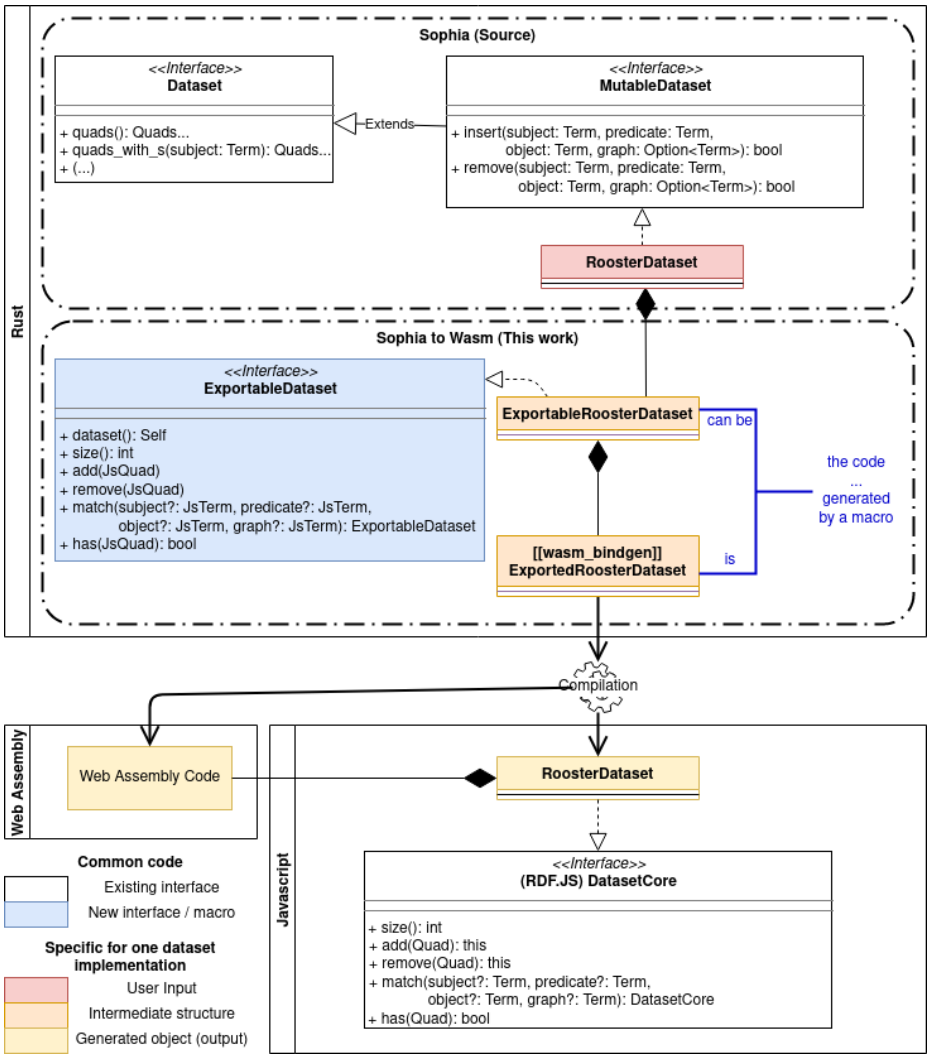

Figure 2. Processus pour exporter un Dataset de Sophia vers une plate-forme Web, dans cet exemple une implémentation nommée RoosterDataset

— les macros permettent d'exporter en une ligne n'importe quel `Dataset` de Sophia vers les plates-formes Web ;
— la classe annotée n'ayant aucun type générique [8], nous avons la garantie de pouvoir exporter n'importe quel dataset que le développeur demande ;
— on peut facilement redéfinir une fonction précise. Les annotations wasm_bindgen étant placées le plus tard possible, et n'ayant pas besoin d'être écrites explicitement par l'utilisateur, il est possible de débugger le comportement en Rust natif [9].

Il y a néanmoins plusieurs défauts à cette approche :

— nous sommes fortement dépendants des évolutions, et donc des changements d'*API* des différents outils utilisés ;
— nous ne pouvons pas respecter entièrement la spécification en raison des limitations de wasm_bindgen : par exemple, la spécification RDF.JS demande que l'on puisse chaîner les appels à la méthode `add` [10] mais notre implémentation de cette méthode ne peut pas renvoyer une référence à `self`.

Nous exposons dans la section 4.2.1 les mesures de performances des datasets exportés en Web Assembly, où nous comparons à la fois du code Rust compilé nativement avec le code exporté pour mesurer le coût de l'exportation, mais également le code exporté avec une autre bibliothèque, Graphy, dans la section 4.2.3, afin d'évaluer le bénéfice de cette démarche par rapport à une implémentation en Javascript uniquement.

3.2 Une nouvelle implémentation des datasets de Sophia : le dataset boisé

Lorsque nous avons mesuré les performances de notre première approche, nous avons remarqué un comportement sous-optimal des `FastDataset` pour traiter certaines requêtes que nous expérimentons dans la section 4.2.2. Nous commençons ici par exposer les défauts liés à la conception du `FastDataset`, et nous proposons une nouvelle implémentation du trait `Dataset`, se reposant sur des B-Arbres, pour combler ces manques.

---

8. wasm_bindgen ne permet pas d'exporter des types génériques (`Class<T>`) car contrairement à Java qui fait de l'effacement de type (`ArrayList<Integer>` et `ArrayList<Double>` sont un même type concret), tout comme C++, Rust compile une version distincte de la classe pour chaque type générique utilisé lorsqu'il rencontre une utilisation : `Vector<i32>` et `Vector<f64>` sont compilés en deux structures `Vector_i32` et `Vector_f64` distinctes lorsqu'elles sont utilisées.

9. Dans la version actuelle de wasm_bindgen, lorsqu'une fonction exportée contient une erreur de compilation, cette erreur est souvent mal localisée par le compilateur. Il est donc préférable de repousser un maximum de code dans d'autres fonctions.

10. Le chaînage de méthodes est une technique permettant d'écrire `monDataset.add(quad1).add(quad2)`.

3.2.1 Motivations pour une alternative au FastDataset

Nous avons présenté les `FastDataset` dans la section 2.5.1. Nous y exposions le fait que pour accélérer certaines requêtes, les quads étaient indexés dans différents ordres pour répondre de manière optimale aux requêtes suivant le schéma du dictionnaire. Les requêtes ne pouvant pas y répondre de manière optimale sont transmises au `Dataset` contenu.

Nous nous étions alors intéressé au traitement d'une requête cherchant tous les quads ayant pour sujet `https://www.bruy.at/julian` dans le graphe par défaut. Si maintenant, nous cherchons seulement tous les quads ayant pour sujet `https://www.bruy.at/julian`, sans préciser le graphe, la requête est relayée au `LightDataset`, qui va parcourir la totalité des quads et filtrer ceux n'ayant pas le bon sujet (c'est dans ce sens que la figure 2.5.1 explicite la fonction `quads_with_S` au niveau de la classe `LightDataset`). Or, une stratégie plus judicieuse dans le cadre d'un `FastDataset` (ou `OGSPWrapper<GSPOWrapper<Set<Quad>>>`), serait de parcourir tous les graphes au niveau de `GSPOWrapper`, et pour chaque graphe, récupérer les quads ayant le bon sujet.

Autrement dit, le manque de vision d'ensemble empêche aux *wrappers* de traiter des requêtes pour lesquelles leur implémentation n'est certes pas optimale, mais qui est la plus performante dans l'ensemble de la structure. Ceci amène aux mauvaises performances constatées dans la section 4.2.2.

3.2.2 Nouvelle implémentation de Dataset

Tout comme le `FastDataset`, nous cherchons à réaliser une structure de données qui permet de :

- stocker des quads dans une structure de données sous la forme d'index, et avoir séparément une table de correspondance entre index et termes ;
- garantir qu'un quad est présent au plus une fois dans la structure ;
- récupérer de la manière la plus efficace possible les quads ayant un certain motif.

Nous allons exposer les différents aspects qui ont été étudiés pour l'implémentation de notre nouveau dataset.

Structure d'arbre. La structure d'arbre offre des avantages en matière de maintenabilité du code. En effet, Rust possède des fonctions préétablies pour extraire d'un arbre tous les éléments compris entre deux bornes, ce qui correspond exactement à notre besoin de faire de la correspondance de motif si l'arbre est trié dans le bon ordre.

Cache du processeur. Les processeurs travaillant en suivant le principe de localité [11], Rust propose une implémentation se reposant sur les **B-Arbres** [7]. Contrairement aux arbres binaires qui à chaque noeud ne stockent qu'un élément, les B-Arbres stockent entre B et $2B-1$ valeurs, ce qui permet à cette implémen-

11. https://en.wikipedia.org/wiki/Locality_of_reference

tation d'être plus *cache friendly*.

Indexation multiple. Une autre remarque qui a été tirée du *benchmark* exposé dans la section 4.1.2 est que la mémoire consommée par Web Assembly est très inférieure à celle consommée par une bibliothèque native Javascript. De nombreux systèmes implémentent déjà une indexation multiple, comme RDF-3X [20] qui garde en mémoire 6 B-Arbres pour ses graphes (un par ordre possible sujet - prédicat - objet), RDF-4X [5] qui s'en inspire en implémentant un index par motif d'accès possible ou bien N3.js [12] qui maintient trois index différents.

Choix des index. L'*API* de Sophia nous permettant de demander les quads ayant un n'importe quel motif, mais ne permettant pas de choisir l'ordre de tri des quads retournés par rapport aux termes manquants, nous reprenons l'approche utilisée en RDF-4X. Par défaut dans notre implémentation, nous construisons jusqu'à six arbres où nous trions les quads respectivement dans l'ordre OGPS, GPSO, POGS, SPOG, GSPO et OSGP. Cela nous garantit d'avoir un arbre qui possède l'ordre optimal pour chaque motif SPOG possible.

Indexation paresseuse. Un autre choix qui a été fait est de faire de l'indexation paresseuse. En effet, la méthode `match` de RDF.JS doit retourner un nouveau dataset. Or, la plupart du temps, ce nouveau dataset ne servira qu'à itérer sur l'ensemble de ses quads, ce qui ne nécessite aucune indexation. Limiter au minimum l'effort d'indexation *a priori* est donc une stratégie qui semble cohérente. L'indexation paresseuse consiste à ne construire un index que lorsqu'on reçoit une requête où cet index serait optimal.

Le dataset boisé (implémenté sous le nom de `TreedDataset`) est une implémentation du trait `Dataset` de Sophia reprenant l'ensemble des propriétés que nous venons de décrire. Nous en mesurons les performances à l'exportation dans une plate-forme Web dans la section 4.2.3, aux côtés des implémentations de base proposées par Sophia.

3.3 Approches hybrides entre Web Assembly et Javascript

Les mesures effectuées sur les deux premières solutions exposées en Web Assembly révèlent un défaut de performances :

- en temps (similaire à des bibliothèques Javascript pour le requêtage, voir section 4.2.3, et moins bon pour le chargement, voir section 4.1.1) principalement lié au traitement des chaînes de caractères et aux nombreux échanges entre Javascript et Web Assembly ;
- en gestion de la mémoire, principalement à cause de l'absence de *Garbage Collector* [13].) qui force l'utilisateur à libérer manuellement les objets sous

12. N3.js (https://github.com/rdfjs/N3.js) est une implémentation des *stores* de RDF.JS proposée par Ruben Verborgh depuis 2011

13. Des travaux sont actuellement en cours pour intégrer un *Garbage Collector* à Web Assembly

peine de voir son application s'interrompre prématurément en atteignant la limite de RAM allouable par la plate-forme Web [14].

Afin de régler d'une part les problèmes de conformité à l'API RDF.JS, et d'autre part les problèmes de mémoires, nous proposons une nouvelle approche consistant à écrire une classe JAVASCRIPT qui utilise une des classes qui sont générées automatiquement. Nous commençons par mettre en œuvre cette approche en la mettant au service des datasets que nous générons déjà avec WASM_BINDGEN pour en corriger les manques. La seconde est de proposer une nouvelle approche où, plutôt que de garder une compatibilité avec SOPHIA, nous mettons de côté cet outil afin de nous libérer des contraintes qu'entraîne l'interopérabilité entre SOPHIA et RDF.JS. Notre objectif devient alors uniquement de proposer une bibliothèque performante implémentant RDF.JS se basant sur WEB ASSEMBLY.

3.3.1 Adaptateur en JAVASCRIPT des datasets exportés vers RDF.JS

Dans cette approche, nous écrivons une classe en JAVASCRIPT qui contient une instance de la classe JAVASCRIPT générée par WASM_BINDGEN qui fait le lien entre JAVASCRIPT et WEB ASSEMBLY. Son but est de modifier légèrement le comportement des fonctions afin de garantir qu'elles ne provoquent pas de fuite mémoire, et implémentent pleinement la sémantique de RDF.JS.

Nous préférons cette approche à celle consistant à modifier le code généré par WASM_BINDGEN afin d'assurer la compatibilité avec les différentes cibles de compilation (WASM_BINDGEN pouvant générer du code soit pour NODEJS, soit pour les *bundlers* comme WEBPACK) mais également avec les futures versions de WASM_BINDGEN qui pourraient changer la structure du code généré.

3.3.1.1 Correction des méthodes

Nous avons distingué deux classes de méthodes à corriger :

— les méthodes provoquant des fuites mémoires. Pour celles-ci, plutôt que de servir directement les données, nous fabriquons une nouvelle donnée exclusivement en JAVASCRIPT qui exhibe les mêmes fonctionnalités que l'objet se reposant sur WEB ASSEMBLY.

— les méthodes n'exposant pas strictement la bonne interface. C'était le cas de la méthode `add` qui ne renvoyait pas `this`, ce qui fait l'objet du listing 2. Nous donnons également la possibilité à notre dataset d'être itérable (implémentation de la fonction `[Symbol.iterator]()`), ce qui n'était pas possible uniquement avec WASM_BINDGEN.

```
class FixedDataset {
    // dataset est une instance d'un dataset
    // dont le code a ete genere par wasm_bindgen
    constructor(dataset) { this.base = dataset; }
```

14. L'introduction de `FinalizationRegistry` dans les versions très récentes de JAVASCRIPT, jouant le rôle de destructeur, a permis à WASM_BINDGEN de proposer une option pour régler ce problème dans la version 0.2.66 sortie le 28 juillet 2020.

```
    add(quad) {
        this.base.add(quad);
        // Ici, on peut retourner this pour
        // permettre le chainage
        return this;
    }

    // Implementation des autres fonctions
}
```

Listing 2. Exemple de réecriture de la fonction add

3.3.1.2 Transmission de données par chaînes au format N-Quads

Pour itérer sur les quads, nous proposons de changer de stratégie : plutôt que d'exporter un objet permettant d'itérer sur les quads, nous proposons d'exporter une représentation textuelle du dataset. Ce choix est motivé par le fait que lorsque l'on fait des recherches pour optimiser un programme en Web Assembly, un conseil récurrent est de limiter le nombre d'échanges entre lui et Javascript.

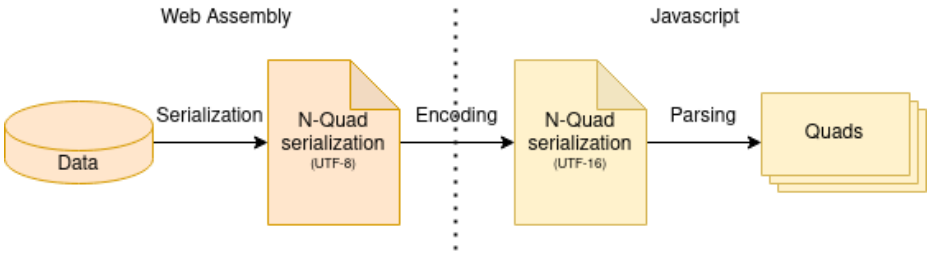

Figure 3. Stratégie pour extraire la liste des quads en utilisant une sérialisation au format N-Quads.

La figure 3 présente les différentes étapes : nous commençons par sérialiser en Web Assembly au format N-Quads [10] les quads du dataset ; Javascript récupère cette chaîne de texte par un appel à la fonction `toNQuads`, qui se charge également du transcodage de UTF-8 vers UTF-16 ; enfin, on utilise le lecteur N-Quads de N3.js pour convertir la sérialisation en une liste de quads.

3.3.2 Mettre Web Assembly uniquement au service de Javascript : WasmTree

L'implémentation qui a été faite des datasets boisés séparait d'un côté le stockage des quads et de l'autre une correspondance entre les index (entiers) et les termes (chaînes de caractères), selon le principe décrit en section 2.1.3. Dans cette nouvelle approche, nous laissons à Rust la gestion des arbres, que nous appelons *back-end* mais nous déléguons à Javascript la correspondance entre les index et leur sémantique, que nous nommons *front-end*.

3.3.2.1 Le back-end Rust

L'implémentation reprend la structure de `TreedDataset` présentée précédemment. La seule différence étant que nous ne gardons pas, dans le code Rust, la dernière étape qui permet de faire la correspondance entre les indices de termes et les termes (et donc les chaînes) qu'ils décrivent.

À la place d'exporter des termes, nous utilisons de nouvelles fonctions permettant d'importer et d'exporter un grand nombre de quads sous la forme de tableaux d'entiers. Le choix de réduire les échanges est motivé par les mêmes raisons que dans la section 3.3.1.2.

Dans le code exporté par wasm_bindgen les tableaux sont toujours importés par référence (afin que le compilateur sache qu'à la fin de l'appel, il peut libérer la mémoire allouée au tableau car le code Rust n'en a pas pris la possession) et exportés en donnant la propriété de l'objet tableau construit (garantissant que Rust n'utilisera plus jamais ce tableau). Ces choix font que contrairement à l'implémentation précédente, la seule structure de données que l'utilisateur devra libérer explicitement est le dataset en lui-même lorsqu'il se repose sur le *back-end* écrit en Rust (la section 3.3.2.3 présente des exemples où nous ne faisons pas toujours appel au *back-end* car un résultat intermédiaire précédent est suffisant).

3.3.2.2 Le front-end Javascript

L'implémentation en Javascript utilise trois éléments principaux pour implémenter l'interface `DatasetCore` de RDF.JS.

- un indexeur qui est une table de correspondance entre les index et les termes qu'ils désignent. Pour cela, des dictionnaires Javascript classiques sont utilisés. Ceux-ci ne pouvant accueillir que des chaînes de caractères comme clés, nous utilisons la notion de termes concis [15] issue de Graphy afin de convertir les termes en une unique chaîne non ambiguë, la difficulté principale provenant des littéraux qui peuvent être décris grâce a une valeur de type chaîne et soit par une IRI décrivant le type de données, soit par une langue ;
- optionnellement un dataset exporté de Rust. C'est le dataset utilisé pour répondre à certaines fonctions demandant des traînements complexes (nous reviendrons sur ce que nous entendons par traitements complexes plus bas) ;
- optionnellement, une liste de nombres, représentant les index des quads. Les listes d'index manipulées par Javascript sont celles qui ont été retournées par Web Assembly dans la section 3.3.2.1. Cette liste est construite lorsque l'on en a besoin, et conservée jusqu'à ce que le dataset soit modifié ;
- ne posséder ni dataset ni liste de nombres implique que le dataset est vide. Cet état est atteint lorsque l'utilisateur a appelé la méthode *free* qui libère également le segment de mémoire linéaire réservée au dataset.

15. https://graphy.link/concise

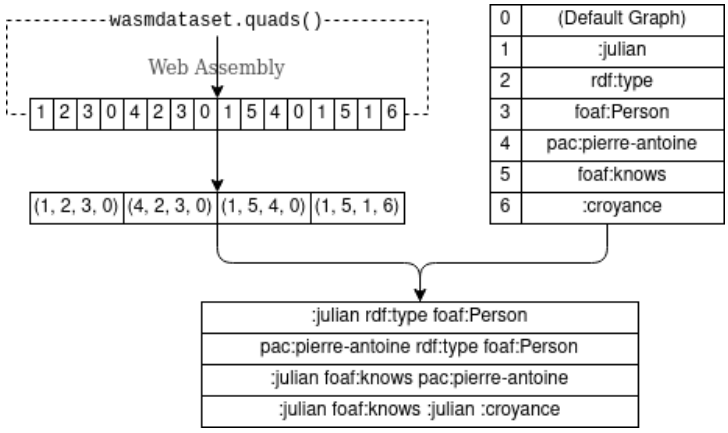

Figure 4. Reconstruction d'une liste de quads en Javascript à partir d'une liste de nombres retournée par Web Assembly

La figure 4 présente comment on convertit une liste de nombres représentant des quads en une liste de quads. Les nombres sont traités par groupes de quatre, où le premier nombre représente l'index du sujet, le second l'index du prédicat, le troisième l'index de l'objet et enfin le quatrième l'index du graphe.

3.3.2.3 Liens entre le back-end et le front-end

Le tableau 1 indique les idées principales derrière l'implémentation de quelques fonctions. Nous privilégions ici Web Assembly pour les opérations complexes comme la modification ou le calcul des données d'un nouveau `Dataset`. Tant que la liste d'index est valable (le dataset n'est pas modifié), nous la conservons pour les opérations d'itération et d'affichage.

4 Évaluation

Dans cette section, nous allons évaluer les performances de nos différentes approches dans trois contextes : le chargement des quads, la recherche de quads suivant un motif simple et enfin dans le contexte d'un moteur SPARQL [16].

Les tests ont été effectués sur un ordinateur équipé d'un processeur Intel(R) Core(TM) i5-1035G1 avec 16 GO de mémoire RAM en DDR4 exécutant le système d'exploitation Linux Ubuntu 20.04 LTS. Les requêtes SPARQL sont évaluées sur une machine virtuelle disposant de 4 CPU virtuels basés sur des Intel Xeon Skylake cadencés à 2600 MHz et 8 Go de mémoire RAM. On utilise pour compiler Rust Compiler 1.43.0, wasm_bindgen 0.2.63 et nous exécutons nos tests sur NodeJS 10.19.0.

16. SPARQL [17] est le langage de requête standard pour interroger des graphes et des datasets RDF. Il permet d'exprimer des requêtes complexes, par opposition aux motifs simples permis par les API Sophia ou RDF.JS.

Table 1. Description de l'algorithme utilisé pour quelques méthodes

Méthode	*Description*	*Dataset*	*Liste d'index*	*Algorithme*
add	Ajoute un quad	Utilisé	Détruite	Convertit le quad en quatre index. Appelle la fonction Web Assembly correspondant à l'ajout
addAll	Ajoute tous les quads d'une liste donnée	Utilisé	Détruite	Convertit en Javascript les n quads à ajouter en une liste de $4n$ index. Passe la liste à Web Assemby pour modifier le dataset
match	Construit un nouveau dataset dont les quads respectent un certain schéma	Utilisé		Construit une liste d'index avec seulement les quads respectant le schéma et l'utilise pour initialiser un nouveau dataset
Itération	Parcourt tous les quads		Utilisée	Décrit dans la figure 4

4.1 Chargement des quads

Nous commençons par présenter l'évaluation du chargement des quads dans un dataset implémentant l'interface `DatasetCore` proposée par RDF.JS. Pour cela, nous utilisons le *parser* de N3.js qui lit un fichier créé en extrayant une partie des quads du jeu de données *Persons* de DBpedia [6].

4.1.1 Temps de chargement

Dans la figure 5, nous affichons la vitesse de chargement de quads dans plusieurs implémentations de `Dataset` à destination des plates-formes Web : tableau de quads est l'exportation d'un `Vec<Quad>` depuis Rust en utilisant l'architecture proposée dans la section 3.1, `FastDataset` est l'exportation en utilisant le même outil, TreedDataset est le dataset implémentant Sophia présenté dans la section 3.2.2, tandis que WasmTree est l'implémentation avec des arbres sans recourir à Sophia (présenté en 3.3.2). Enfin, Graphy est une implémentation concurrente de `DatasetCore` écrite intégralement en Javascript. La lecture des fichiers au format TriG se fait en utilisant le lecteur de N3.js, qui utilise la fonction `add` des différents datasets. À titre indicatif, nous affichons également la vitesse de chargement d'un `FastDataset` compilé et exécuté en Rust natif, qui en conséquence n'utilise pas le lecteur de N3.js mais celui intégré à Sophia.

Tout d'abord, nous pouvons voir que le remplissage d'un `FastDataset` est deux fois plus rapide en Rust natif que dans les meilleures configurations que

nous avons essayées dans une plate-forme Web. Lorsque nous avons profilé le code, nous avons pu constater que le *Garbage Collector* représente une part importante (22%) du temps de chargement. Il n'est néanmoins pas surprenant qu'un langage compilé, ici Rust, soit plus performant qu'un langage interprété, ici Javascript.

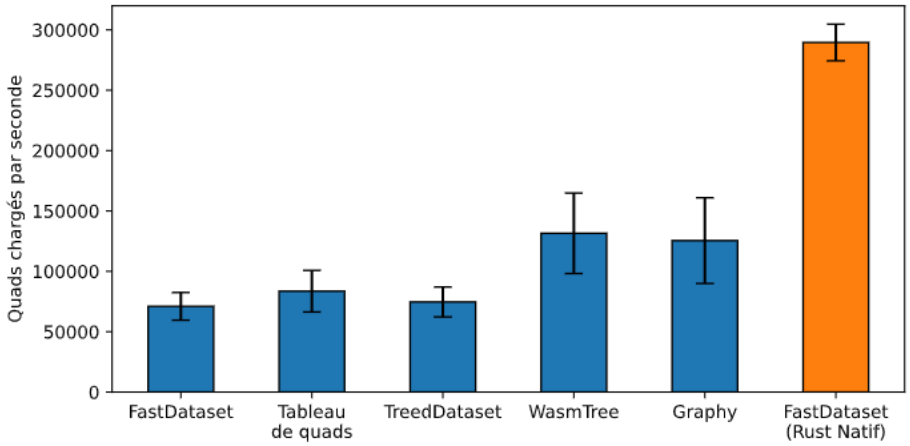

Figure 5. Vitesse de remplissage de différents datasets sur une plate-forme Web (Node.js) avec des quads issus d'un fichier au format TriG lu par N3.js. La vitesse de chargement par une application compilée en Rust natif utilisant le lecteur de Sophia et un FastDataset est indiquée avec la mention « FastDataset (Rust Natif) ».

Toutefois, bien qu'ils utilisent du code Web Assembly, qui est donc compilé, on peut voir que `FastDataset`, le tableau de quads et le `TreedDataset` exhibent tous les trois un temps de chargement plus de 4 fois plus lent que le `FastDataset` en Rust natif, et deux fois plus lent que Graphy, une librairie écrite intégralement en Javascript. WasmTree n'a pas cet écart, exhibant des performances similaires à Graphy. La différence entre WasmTree et TreedDataset est que WasmTree conserve la correspondance entre les termes et les index en Javascript, remplaçant les nombreux échanges (appels de fonctions et transmission de chaînes de caractères) entre Web Assembly et Javascript pour construire en interne le quad à ajouter par un quad représenté par quatre nombres dont la sémantique est opaque pour le code Web Assembly. Nous en déduisons que ce nombre important d'échanges de données non triviales [17] est une cause majeure de dégradation des performances au chargement.

17. Le passage d'un nombre flottant sur 64 bits (type `number` de Javascript), à des entiers non signés sur 32 bits (type `u32` de Rust compilé en utilisant le type `i32` en Web Assembly, dont la sémantique est nombre entier sur 32 bits) est une opération supportée par les processeurs. En revanche, les chaînes de texte en Javascript sont encodées en UTF-16, alors qu'en Rust elles sont encodées en UTF-8. Il faut donc transcoder les chaînes, ce qui est une opération complexe.

4.1.2 Consommation en mémoire

Table 2. Consommation en mémoire (en kB) de différents Dataset et Store dans Node.js pour un million de triplets. « Par défaut » correspond à un dataset que l'on a seulement rempli. « Pleinement initialisé » correspond à un dataset auquel on a fait une requête pour tous les motifs possibles SPOG. Entre parenthèses, le nombre de fois où les quads sont stockés dans la structure.

Dataset	*Par défaut*	*Pleinement initialisé*
Sophia Tree (section 3.2.2)	29 692 (1 index)	30 204 (6 index)
Wasm Tree (section 3.3.2)	427 708 (1 index)	429 244 (6 index)
Graphy	594 964 (1 index)	602 652 (1 index)
N3.js	1 276 956 (3 index)	1 279 712 (3 index)

Nous mesurons ici la consommation en mémoire de différentes structures de données à destination des plates-formes Web, dans lesquelles nous chargeons un million de triplets issus du jeu de données *Persons* de DBpedia. Nous reportons dans le tableau 2 la mesure obtenue lorsque les triplets sont tous chargés, ainsi que la mémoire consommée après avoir construit tous les index possibles (construction de tous les index paresseux).

Nous effectuons les mesures de mémoire en utilisant la donnée `VmPeak` du fichier `/proc/self/status` ce qui introduit des biais notamment parce que Javascript peut continuer à allouer de la mémoire sans avoir libéré de la mémoire dont il ne se sert plus (le *Garbage Collector* n'a pas encore effectué son travail). Cela pourrait être un des facteurs expliquant le faible écart mesuré entre un Sophia Tree avec un seul index et avec six.

Nous pouvons voir que les structures de données se reposant sur Web Assembly sont plus compactes que les structures de données se reposant sur une implémentation Javascript pure. L'implémentation en Sophia natif est de loin la plus compétitive, étant au moins un ordre de grandeur meilleure que toutes les autres. Néanmoins son temps de chargement observé dans la section 4.1.1 et les mesures qui sont présentées pour le requêtage dans la section 4.2 rendent son utilisation prohibitive.

Dans cette mesure, Wasm Tree occupe moins de mémoire que la meilleure approche Javascript pure étudiée ici (Graphy).

Notons également que l'écart mesuré entre avoir un seul arbre ou six est négligeable. Ce fait ainsi que le fait que Sophia Tree ne consomme que peu de mémoire nous permettent d'affirmer que le composant prenant le plus de mémoire est la table de correspondances entre index et termes en Javascript. La construction d'index supplémentaire en Javascript pur ne se faisant pas sans coût, comme le montre l'écart important entre Graphy et N3.js (un arbre GSPO contre trois arbres, GSPO, GPOS et GOSP).

4.2 Recherche de quads suivant un motif défini

Nous souhaitons désormais évaluer la performance de l'extraction de quads suivant un motif donné (pour répondre à une requête du type « trouve toutes les ressources de type personnes »).

L'algorithme de mesure est présenté dans le listing 3. Nous commençons par créer un dataset du type que l'on souhaite mesurer et nous appelons une première fois la fonction `match` afin de déclencher les éventuels mécanismes d'indexation paresseuse et de cache. Nous appelons ensuite une seconde fois la méthode `match`, dont nous mesurons cette fois-ci le temps d'exécution. Cette fonction nous renvoie un dataset, sur lequel nous itérons. Nous mesurons également le temps d'itération.

Un des biais du jeu de données choisi, le dataset *Persons* issus de DBpedia, est qu'il ne mentionne pas de graphe : tous ses triplets sont donc dans le graphe par défaut.

```
fonction benchmark(dataset, quads, s?, p?, o?, g?) {
    // Creation du dataset et mise en place des index
    Pour chaque quad dans quads:
        dataset.add(quad);

    dataset.match(s, p, o, g);

    // Mesure du temps de requetage
    lancer_chronometre();
    resultat = dataset.match(s, p, o, g);
    temps_match = arreter_chronometre();

    // Mesure du temps d'iteration
    lancer_chronometre();
    compteur = 0;
    Pour chaque quad dans resultat:
        compteur += 1;
    temps_iteration = arreter_chronometre();

    retourner temps_match, temps_iteration;
}
```

Listing 3. Algorithme de mesure

4.2.1 FastDataset compilé pour une plate-forme Web

Nous cherchons ici à requêter toutes les personnes décrites dans le graphe par défaut d'un `FastDataset`. Le nombre de personnes décrites augmente linéairement avec la taille de l'extrait du dataset *Persons* que nous prenons. Pour cela, nous explorons différentes stratégies, en Rust natif et dans une plate-forme Web :

— lorsque nous appelons la méthode `match`, soit nous remplissons un nouveau `FastDataset`, soit nous remplissons un tableau de quads. Remplir un `FastDataset` est en effet une tâche longue car il faut ajouter chaque quad dans trois structures différentes (le `Set`, l'`OGPSWrapper` et le `SPOGWrapper`). Or, l'usage attendu du dataset renvoyé par la méthode `match` est de parcourir les quads qui ont été filtrés. Un tableau de quads est la structure à la fois la plus rapide à remplir et la plus rapide à parcourir ;

— l'utilisation, dans une plate-forme Web, de l'approche présentée dans la section 3.3.1.2, consistant à faire communiquer Web Assembly et Javascript avec une représentation textuelle au format N-Quads plutôt que de transférer les quads un par un.

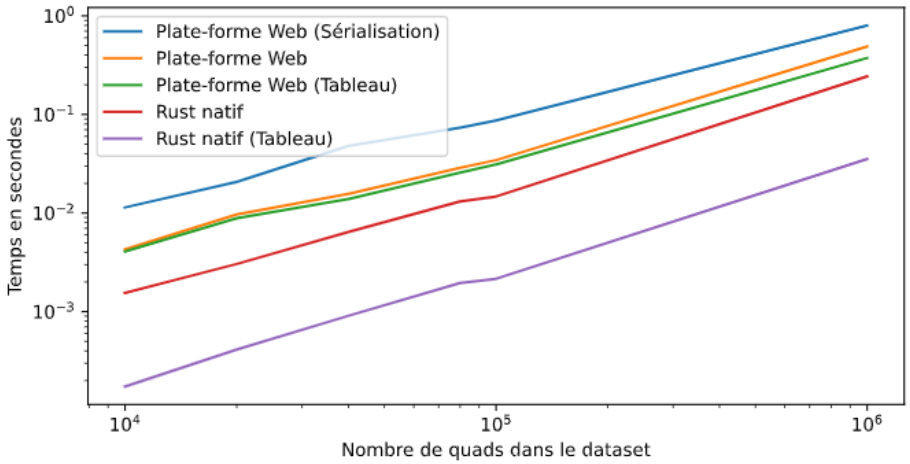

Figure 6. Temps mesurés pour l'extraction de toutes les personnes d'un FastDataset, la construction d'un nouveau dataset et son parcours, placés sur une échelle logarithmique (requête au motif POG).

La figure 6 expose les résultats obtenus. Nous pouvons voir tout d'abord que l'exportation d'un dataset de Rust vers Web Assembly pour être utilisé dans une plate-forme Web ne se fait pas sans coût : nous constatons un facteur multiplicateur d'environ 5.

L'utilisation d'un tableau comme dataset de destination (indiqué par la mention « Tableau » dans la légende) permet de gagner un ordre de grandeur en Rust natif, mais n'améliore que légèrement les performances au sein d'une plate-forme Web. En revanche, l'utilisation d'une représentation textuelle au format N-Quads pour communiquer entre Web Assembly et Javascript (comme décrit en 3.3.1.2) dégrade les performances : en moyenne utiliser cette stratégie est deux fois plus lent sur l'ensemble du traitement de la requête.

La figure 7 montre la répartition de la charge entre l'appel à `match` (la construction) et le parcours des quads. Nous pouvons voir qu'en Rust natif, le temps de parcours est négligeable, tandis que dans le cadre d'une plate-forme Web il représente un temps non négligeable. Il est supérieur lorsque nous utilisons l'approche N-Quads par rapport à une approche naïve où des itérateurs sont exportés de Web Assembly vers Javascript, ce qui fait penser que le temps pour construire la représentation textuelle, la transmettre et reconstruire les quads

est supérieur au temps cumulé des échanges individuels de données concernant un quad (ici plus de trois fois).

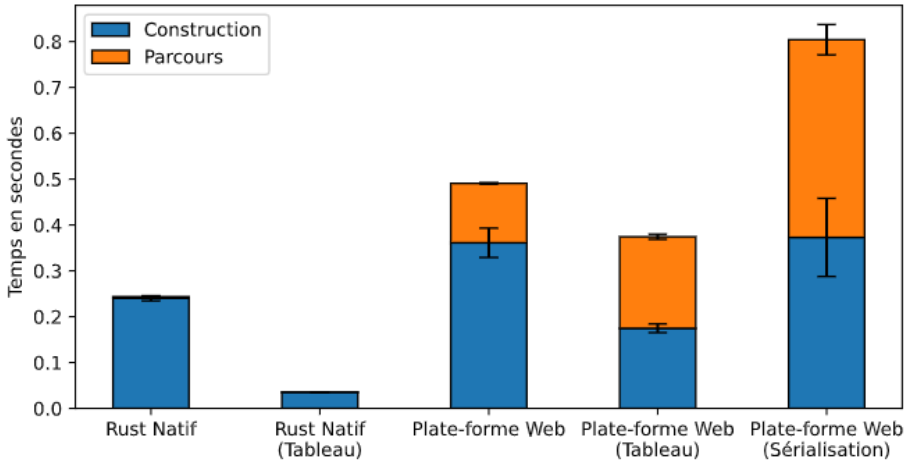

Figure 7. Répartition de la charge pour le requêtage de toutes les personnes d'un FastDataset de 1'000'000 de quads (requête au motif POG) avec différentes stratégies.

4.2.2 Impact du motif demandé pour la recherche dans un FastDataset

Nous souhaitons désormais étudier l'impact du motif demandé dans le requêtage d'un `FastDataset` Pour cela, nous allons faire une requête dont nous savons que la taille du résultat ne dépend pas de la taille de l'extrait du dataset *Person* choisi, à savoir tous les quads ayant pour sujet `Vincent_Descombes_Sevoie`.

Nous exposons nos mesures dans la figure 8. Nous pouvons observer que dans le cadre d'une requête spécifiant le graphe, le temps d'extraction ne dépend pas de la taille du jeu de données. La stratégie d'indexation est donc bien efficace.

Néanmoins, comme nous le mentionnions dans la section 3.2.1, bien que les structures de données soient présentes pour répondre à cette requête de manière efficace, les performances sont dégradées lorsqu'on ne spécifie pas le graphe dans le motif recherché – et ce, même malgré le fait que le dataset ne contient qu'un seul graphe. On constate que le temps pour trouver un petit nombre constant de quads devient proportionnel à la taille du dataset. En utilisant un `TreedDataset`, quel que soit le motif utilisé (S ou SG), nous obtenons des performances similaires au `FastDataset` pour un programme compilé en Rust natif et légèrement inférieures dans le cadre d'une plate-forme Web.

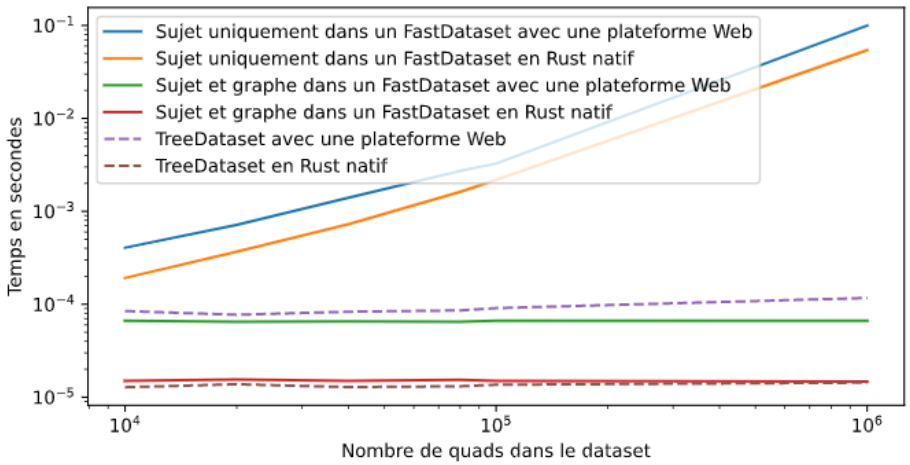

Figure 8. Temps mesurés pour l'extraction, la construction d'un nouveau dataset et son parcours dans le cadre de requêtes concernant une personne du jeu de données *Person*, avec et sans mention du graphe

4.2.3 Comparaisons de différentes implémentations sur une plate-forme Web

La figure 9 présente les résultats obtenus avec différentes solutions pour requêter tous les noeuds de type Personne (sortie de taille linéaire selon le nombre de quads). La meilleure stratégie utilisant Sophia est le vecteur de quads : toute stratégie pour accéder rapidement aux quads de type Personne est donc trop coûteuse par rapport au grand nombre de quads (environ 13%) à conserver. De plus, cette solution donne des performances similaires à Graphy ce qui signifie que pour cette requête, utiliser Sophia et Web Assembly n'apportent pas de gain de performances.

WasmTree est plus rapide de plus d'un ordre de grandeur que Graphy, ce que l'on peut attribuer à une répartition judicieuse des traitements entre Web Assemly et Javascript, en évitant les échanges coûteux entre les deux langages. Nous pouvons également mentionner le fait que WasmTree utilise toujours une indexation optimale, alors que Graphy n'utilise qu'un seul index GSPO. Rappelons, comme on l'a montré plus haut, que les six indexes de WasmTree occupent moins de place en mémoire que le seul index de Graphy.

La figure 10 expose les temps d'exécution d'une requête dont la sortie ne dépend pas de la taille du graphe. On peut voir que dans la pratique, toutes les approches répondent en un temps (presque) constant, à part le vecteur de quads qui doit parcourir tous les quads. Le `FastDataset` devient également légèrement plus rapide que Graphy sans atteindre les performances du WasmTree.

Dans la figure 9, nous pouvons voir que pour WasmTree, et dans une moindre mesure pour Graphy, il existe un coût fixe en temps d'appel assez élevé par

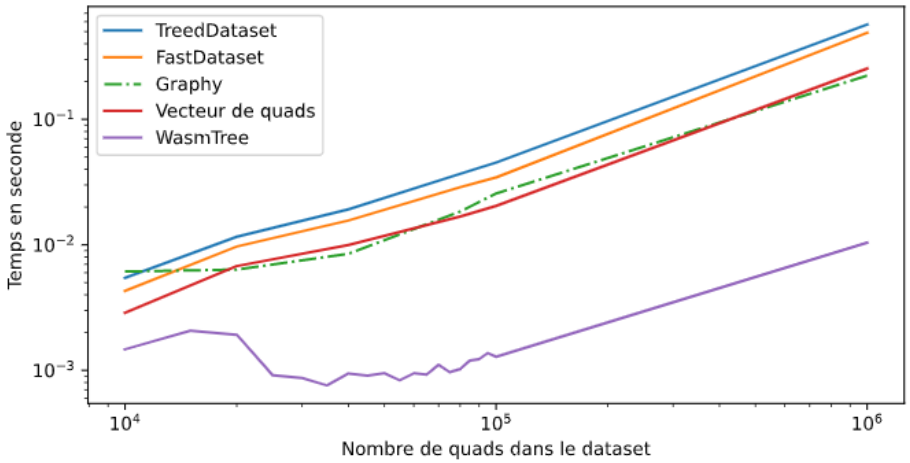

Figure 9. Temps pour chercher toutes les personnes mentionnées dans un dataset sur une plate-forme Web (requête au motif POG)

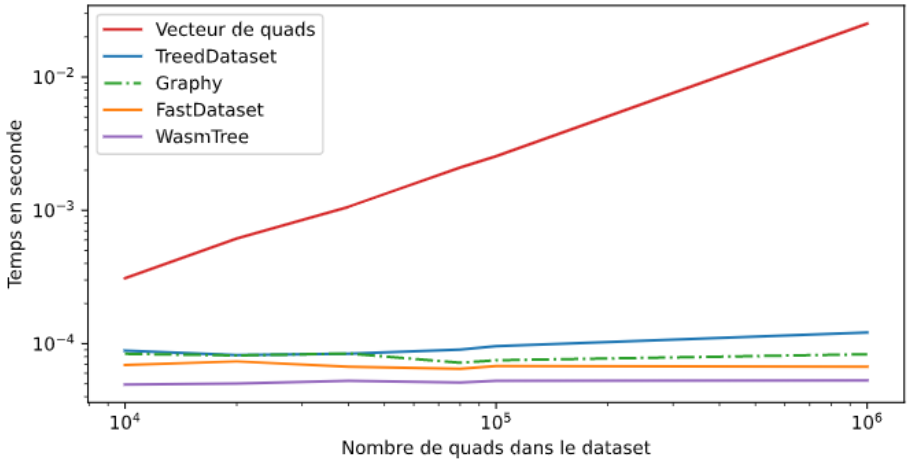

Figure 10. Temps pour chercher tous les quads dont le sujet est Vincent Descombes Sevoies sur une plate-forme Web (requête au motif SG)

rapport aux coûts dépendant de la taille de la sortie qui fait stagner les courbes pour de petits datasets, là où les autres courbes semblent toujours linéaires. Le fait que l'on ne retrouve pas ces coûts fixes dans la figure 10 nous fait penser que l'on mesure principalement des coûts d'allocations de pages de mémoires supplémentaires (une page mémoire mesurant 64 Ko).

4.3 SPARQL

SPARQL est un langage de requêtage permettant de poser des requêtes complexes, par exemple trouver les noms et dates de naissance de toutes le personnes nées à Lyon entre 1900 et 2000. Un *SPARQL endpoint* est un serveur accessible via le protocole HTTP permettant à un utilisateur d'interroger une base de données RDF grâce à une requête SPARQL.

The Berlin SPARQL Benchmark [8] (BSBM) est une infrastructure de mesure de performances pour SPARQL. Cette infrastructure permet de générer des datasets concernant des produits dans le contexte d'une boutique et propose un pilote permettant de mesurer les performances d'un *SPARQL endpoint*. Nous nous intéressons ici à la recherche d'informations. La mesure se fait en envoyant un ensemble de 25 requêtes, générées à partir de 12 schémas de requêtes types issus de l'*Explore Use Case* de BSBM : la requête type 1 « trouver le nom de tous les produits qui ont deux caractéristiques précises », le pilote de test choisissant aléatoirement à chaque requête les caractéristiques voulues.

RDF.JS ne proposant pas d'interface permettant de faire des requêtes SPARQL, pour faire nos mesures nous utilisons Comunica, que nous avons présenté dans la section 2.6, qui permet de créer un moteur SPARQL à partir de n'importe quelle implémentation de RDF.JS `Store`. Nous comparons ici le réglage par défaut, utilisant N3.js, avec une version utilisant WasmTree à la place.

Oxigraph [18] est une implémentation de RDF en Rust qui permet de faire des requêtes SPARQL et possédant déjà sa propre API en Javascript (implémentant partiellement RDF.JS) via Web Assembly. Nous exploitons la méthode `.sparql` qu'elle expose pour créer un SPARQL endpoint avec Node.js dont le moteur se repose intégralement sur Web Assembly.

4.3.1 Mesures de temps

Le tableau 3 montre que dans le contexte d'exécution de requêtes SPARQL, remplacer N3.js par WasmTree comme *store* utilisé par Comunica nous permet d'exécuter 6, 7 fois plus de requêtes en un temps donné. Web Assembly permet donc déjà de gagner en performances en étant utilisé que pour effectuer certaines opérations (la recherche de quads) d'une opération plus complexe (devant en plus réaliser le plan d'exécution et les jointures). Oxigraph atteint des performances 20 fois supérieures à notre configuration de Comunica, ce qui nous confirme l'efficacité de Web Assembly à effectuer rapidement des calculs en interne.

18. https://github.com/oxigraph/oxigraph

Table 3. Nombre d'ensembles de requêtes exécutés par heure par le pilote de test BSBM sur un SPARQL end point utilisant pour source un dataset de 2000 produits (725305 quads). Mesuré à partir de 100 exécutions après une chauffe de 20 exécutions.

Moteur SPARQL	***Mix de requêtes par heure***	**Accélération par rapport à**		
		Oxigraph	**Comunica avec WasmTree**	**Comunica avec N3.js**
Oxigraph	8877,66		x19.90	x135,25
Comunica avec WasmTree	446,22	x0,05		x6,80
***Comunica avec* N3.js**	65,64	x0,007	x0,15	

4.3.2 Réflexions liées à l'usage de la mémoire

À l'origine, nous souhaitions faire notre comparaison avec un *scale factor* de 10'000 (paramétrage de BSBM correspondant à environ 4 millions de quads). Nous avons néanmoins été confrontés à des problèmes de manque de mémoire. Pour les approches basées sur Comunica, nous avons pu régler ces problèmes en augmentant la limite de mémoire allouée à Javascript. En revanche, pour Oxigraph, la limite vient du fait que les adresses mémoires en Web Assembly sont codées sur 32 bits, avec une limite théorique à 4Go, limite vite atteinte avec de gros datasets. WasmTree est moins impacté par cette limite car il ne stocke que ses index (dont on a vu plus haut qu'ils ne représentent qu'une petite fraction de la mémoire totale) dans la mémoire de Web Assembly.

5 Conclusion

Dans ce travail, nous avons exploré diverses manières d'implémenter, en utilisant Rust et Web Assembly, des bibliothèques RDF utilisables depuis Javascript et interopérables avec l'existant :

— une méthode permettant d'exporter des `Dataset` implémentant les traits de Sophia en respectant les interfaces de RDF.JS. Cette méthode, bien que permettant l'interopérabilité entre les deux outils, ne permet pas de bénéficier de meilleures performances ;

— une nouvelle bibliothèque, implémentant uniquement RDF.JS et donc indépendante de Sophia, donnant de bons résultats par rapport aux bibliothèques existantes, et pouvant servir de base pour accélérer les performances de bibliothèques exposant du SPARQL comme Comunica ;

Ces différentes expérimentations montrent que Web Assembly, un langage compilé exécutable à la fois par les navigateurs et le serveur avec Node.js, est un bon candidat pour l'amélioration des performances des bibliothèques Javascript pour le Web sémantique. Elles tendent à montrer que les performances s'améliorent en limitant les échanges, ainsi que les coûts associés au *Garbage Collector*, la transmission de textes et sa ré-interprétation. Le maintien de structures et l'application d'opérations complexes en Web Assembly (comme une

structure d'arbre ou un moteur SPARQL) tout en laissant à JAVASCRIPT la charge d'opérations simples et la gestion de structures simples également (comme la correspondance entre termes et index) ont été particulièrement fructueux.

Les différences des API SOPHIA et RDF.JS, bien que subtiles, comme la manière de retourner la liste de quads (un dataset ou un flux de quad) ou l'encodage des chaînes de caractères, amènent à des dégradations des performances. Pour éviter les coûts liés aux chaînes, une standardisation, grâce à un trait dans SOPHIA, de l'interface de dataset travaillant sur des index (et non plus sur des termes) est une piste intéressante. Ainsi, n'importe quel dataset implémentant ce trait pourrait être exporté de manière efficace à destination des plates-formes Web, à la manière de WasmTree. Ce trait serait également bénéfique au monde RUST, permettant par exemple de réaliser des opérations ensemblistes sur les quads représentés sous forme d'index issus de deux datasets partageant un dictionnaire commun.

Néanmoins, la manipulation exclusive d'index fait perdre la sémantique des termes manipulés, et donc la possibilité de raisonner sur eux (alors qu'avec la première architecture, proposée à la section 3.1, il était envisageable d'exporter directement des datasets faisant de l'inférence, comme Inferrust [9]). Dans cette optique, il faudrait alors développer un moyen de coordonner les deux parties de l'implémentation, y compris si elles sont écrites dans des langages différents. Le *back-end* aurait alors pour rôle de stocker les quads sous forme d'index et pourrait mettre en œuvre les mécanismes de raisonnement, Le *front-end* garderait la correspondance entre chaque index et sa sémantique pourrait ainsi piloter ces mécanismes.

Références

1. RDF/JS : Data model specification. Tech. rep. (Sep 2019), http://rdf.js.org/data-model-spec, [Online ; accessed 23. Jul. 2020]
2. RDF/JS : Stream interfaces. Tech. rep. (May 2019), http://rdf.js.org/stream-spec, [Online ; accessed 23. Jul. 2020]
3. RDF/JS : Dataset specification 1.0. Tech. rep. (Jun 2020), https://rdf.js.org/dataset-spec, [Online ; accessed 23. Jul. 2020]
4. The wasm-bindgen Guide (Aug 2020), https://rustwasm.github.io/docs/wasm-bindgen, [Online ; accessed 25. Aug. 2020]
5. Abbassi, S., Faiz, R. : Rdf-4x : a scalable solution for rdf quads store in the cloud. In : Proceedings of the 8th International Conference on Management of Digital EcoSystems. pp. 231–236 (2016)
6. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z. : Dbpedia : A nucleus for a web of open data. In : The semantic web, pp. 722–735. Springer (2007)
7. Bayer, R., McCreight, E. : Organization and maintenance of large ordered indices. In : Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control. p. 107–141. SIGFIDET '70, Association for Computing Machinery, New York, NY, USA (1970). https://doi.org/10.1145/1734663.1734671

8. Bizer, C., Schultz, A. : The berlin sparql benchmark. International Journal on Semantic Web and Information Systems (IJSWIS) **5**(2), 1–24 (2009)
9. Bourg, T., Champin, P.A. : Raisonnement efficace pour le web sémantique. Tech. rep. (2020)
10. Carothers, G. : RDF 1.1 n-quads. W3C recommendation, W3C (Feb 2014), https ://www.w3.org/TR/2014/REC-n-quads-20140225/
11. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P. : Named graphs, provenance and trust. In : Proceedings of the 14th international conference on World Wide Web. pp. 613–622 (2005)
12. Champin, P.A. : Sophia : A Linked Data and Semantic Web toolkit for Rust. The Web Conference 2020 : Developers Track (Apr 2020), https://www2020devtrack.github.io/site/schedule
13. Cyganiak, R., Wood, D., Lanthaler, M. : RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, W3C (Feb 2014), https://www.w3.org/TR/rdf11-concepts/
14. Dürst, M., Suignard, M. : Internationalized Resource Identifiers (IRIs). RFC 3987, IETF (Jan 2005), https://tools.ietf.org/html/rfc3987
15. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M. : Binary rdf representation for publication and exchange (hdt). Journal of Web Semantics **19**, 22–41 (2013)
16. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J. : Bringing the web up to speed with webassembly. SIGPLAN Not. **52**(6), 185–200 (Jun 2017). https://doi.org/10.1145/3140587.3062363
17. Harris, S., Seaborne, A. : SPARQL 1.1 Query Language. W3C Recommendation, W3C (Mar 2013), http://www.w3.org/TR/sparql11-query/
18. Jena, A. : A free and open source java framework for building semantic web and linked data applications. Available online : jena.apache.org/(accessed on 28 April 2015) (2015)
19. Matsakis, N.D., Klock, F.S. : The rust language. ACM SIGAda Ada Letters **34**(3), 103–104 (2014)
20. Neumann, T., Weikum, G. : Rdf-3x : a risc-style engine for rdf. Proceedings of the VLDB Endowment **1**(1), 647–659 (2008)
21. Patel-Schneider, P., Hayes, P. : RDF 1.1 semantics. W3C recommendation, W3C (Feb 2014), https ://www.w3.org/TR/2014/REC-rdf11-mt-20140225/
22. Schreiber, G., Raimond, Y. : RDF 1.1 Primer. W3C Working Group Note, W3C (Jun 2014), http://www.w3.org/TR/rdf-primer/
23. Seaborne, A., Carothers, G. : RDF 1.1 trig. W3C recommendation, W3C (Feb 2014), https ://www.w3.org/TR/2014/REC-trig-20140225/
24. Subercaze, J., Gravier, C., Chevalier, J., Laforest, F. : Inferray : fast in-memory RDF inference. PVLDB **9**(6), 468–479 (2016), http://www.vldb.org/pvldb/vol9/p468-subercaze.pdf
25. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R. : Comunica : a modular sparql query engine for the web. In : International Semantic Web Conference. pp. 239–255. Springer (2018)
26. Terdjimi, M., Médini, L., Mrissa, M. : Hylar+ : Improving hybrid location-agnostic reasoning with incremental rule-based update (04 2016). https://doi.org/10.1145/2872518.2890542

Annexes

A De Javascript au Web Assembly

A.1 Optimisation de Javascript

Javascript est un langage qui s'est imposé sur les navigateurs Web face à des concurrents tels que Flash ou Silverlight. Là où des plugins comme Flash sont connus pour poser des problèmes de sécurité, les permissions de Javascript sont très restreintes. De plus, la conception du langage, très simple et très permissive, permet de développer facilement en Javascript.

Néanmoins, avec l'augmentation du nombre de programmes en Javascript, et étant le seul langage viable sur un navigateur, les concepteurs de navigateur sont dû recourir à des méthodes pour améliorer les performances du Javascript.

Une première méthode a été l'introduction d'un JIT (Just In Time Compiler : l'idée principale est de repérer les fonctions appelées fréquemment, et de les optimiser de plus en plus à chaque appel pour qu'elles soient de plus en plus rapides.

A.2 asm.js

Le JIT permet d'améliorer les performances de n'importe quelles applications, mais certains développeurs peuvent vouloir développer des applications ayant explicitement des bonnes performances.

asm.js est un sous-ensemble du langage Javascript. Lorsqu'il interprète un bloc de code déclaré compatible, l'interpréteur peut déployer des stratégies plus efficaces pour accélérer la vitesse d'exécution.

Par exemple, Javascript travaille nativement sur des nombres flottants, qui ont un temps d'utilisation plus lent que des entiers. Pour manipuler des entiers sur 32 bits, une astuce consiste à appliquer au nombre une opération « ou binaire » avec 0 (`valeur | 0`). De plus, les tableaux dont la taille n'est pas changée ne sont pas réalloués. Ainsi, on peut allouer un grand tableau, et ne jamais créer d'objet et plutôt stocker les valeurs dans ce tableau qui sert alors de pseudo mémoire RAM.

Ainsi, asm.js a permis à certains développeurs d'exporter des jeux en 3D (demandant donc une puissance de calcul pour le traitement de l'image) sur navigateur.

A.3 L'aboutissement de la compilation vers asm.js : Web Assembly

Le format utilisé par asm.js reste le format Javascript, qui est un format textuel qu'il faut donc lire caractère par caractère. Afin de réduire le temps de transfert du code ainsi que le temps de lecture du format textuel, l'idée de transmettre des fichiers binaires a émergé. C'est ainsi que l'on arrive au Web Assembly, un langage exécutant des fichiers binaires qui permettent de modifier un tableau d'entiers pré-alloué (nommé mémoire linéaire).

B Le système d'emprunt de Rust

La notion qui domine Rust, et le différencie des autres langages de programmation, est son système d'emprunt.

Rust ne possède ni *Garbage Collector* (tâche de fond qui libère pendant l'exécution la mémoire allouée aux objets qui ne sont plus utilisés), ni libération explicite de la mémoire par le programmeur. À la place, Rust considère qu'un objet n'est possédé à tout moment que par une seule variable. Un objet peut être prêté en lecture seule en plusieurs exemplaires, mais pendant que l'objet est emprunté en lecture, il ne peut pas être modifié, pas même en passant par la variable propriétaire. Il est également possible de prêter une version mutable de l'objet, faisant que l'objet ne pourra être lu ou modifié par personne d'autre. Enfin, il est possible de transférer la propriété de l'objet.

Ces vérifications sont généralement faites à la compilation. Toutefois, la vérification de la mémoire étant un problème complexe, et certaines opérations systèmes n'étant pas sûres par nature, il est possible pour l'utilisateur de n'activer cette vérification qu'à l'exécution via les objets `RefCell` ou `Mutex`, voire de totalement supprimer cette vérification avec un bloc de code `unsafe`.