Neosemantics (n10s) 5.20.0

n10s neosemantics	**Plugin Neo4J pour la compatibilité avec les outils sémantiques de RDF**	
	Système :	Multiplateforme
	Développé par :	Neo4J Labs
Open source, incompatible avec certaines versions de Neo4j	Personne de contact :	pierre.leleux@smals.be

Fonctionnalités

Neosemantics (souvent abrégé en n10s) est un plugin pour Neo4J ayant pour objectif de rendre ce dernier compatible avec les outils sémantiques offerts par les graphes de connaissances en RDF (*Resource Description Framework*).

RDF, initialement conçu comme un framework pour des opérations d'échange de données, met un fort accent sur l'interopérabilité, mais dispose aussi d'une grande richesse sémantique (ontologies en RDFS/OWL, raisonneurs OWL pour l'inférence automatique d'informations implicites, validation du contenu des graphes de connaissances via SHACL). Neo4J, étant conçu en tant que base de données orientée graphe basée sur une structure de type LPG (Labeled Property Graph), propose une solution plus simple à mettre en place et offre des performances de requêtage optimisées, mais au prix de la perte de la richesse sémantique offerte par RDF. L'objectif du plugin Neosemantics est donc de donner accès aux outils sémantiques de RDF directement en Neo4J.

Ce plugin permet entre autres :

- D'importer des données encodées en format RDF (triplets RDF) vers une base de données Neo4J.
- D'importer des ontologies RDF.
- De créer des règles d'inférence automatique en Neo4J à partir de fichiers (ou de lignes de code) en RDFS/OWL.
- De valider le contenu du graphe de connaissances sur base de contraintes exprimées en SHACL.

En tant que plugin, l'utilisation de cet outil se fait directement en passant par Neo4J, via l'utilisation de requêtes en Cypher. Les outils d'inférence automatique et de validation peuvent au choix être exécutés d'un coup sur l'intégralité du graphe, ou de façon transactionnelle pour ne travailler que sur les données ajoutées durant une transaction.

Conclusions & recommandations

Neosemantics permet de pallier plusieurs limites liées à l'utilisations de Neo4J en tant qu'outil pour gérer un graphe de connaissances. En contrepartie, apprendre à utiliser ce plugin est relativement compliqué, car il requiert non seulement des connaissances en Neo4J (et en Cypher), mais aussi une maîtrise suffisante des langages RDF, RDFS, OWL et SHACL que pouvoir y écrire l'ontologie, les règles d'inférence et/ou les contraintes de validation, afin de pouvoir les importer ensuite vers Neo4J. Dans la pratique, ce plugin est relativement niche : sauf si l'on dispose déjà de fichiers contenant l'ontologie en RDF et/ou des contraintes en SHACL que l'on souhaite utiliser directement en Neo4J, ce plugin n'apporte pas une grande valeur ajoutée par rapport à l'écriture de requêtes Cypher pour réaliser la validation/inférence.

Tests et résultats

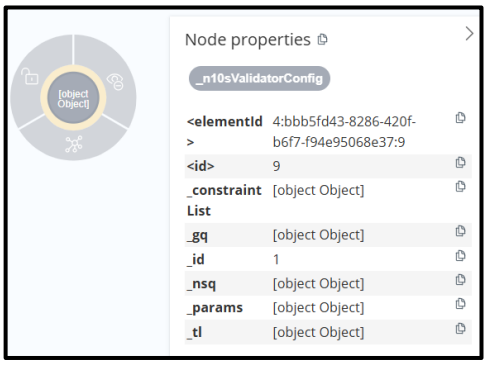
Nous avons testé Neosemantics sur une machine virtuelle sous Linux, en utilisant la distribution debian-based de Neo4J.

Le plugin est relativement facile à activer, il suffit de télécharger le plugin (.jar) depuis sa page github, de le placer dans le folder `plugins/` de Neo4J, puis de l'activer via le fichier config de Neo4J (`neo4j.conf`) et de redémarrer Neo4J (si nécessaire).

Une fois le plugin installé, les procédures de n10s peuvent s'appeler simplement depuis Neo4J en Cypher via les commandes `[call n10s]`. En guise d'exemple, importer des contraintes de validation SHACL vers Neo4J peut se faire, au choix, via la commande :

`call n10s.validation.shacl.import.inline('my_SHACL_constraints')` pour importer des contraintes SHACL directement sous forme de lignes de code, ou alors via

`call n10s.validation.shacl.import.fetch(my_SHACL_file)` pour importer les contraintes contenues dans un fichier.



Les contraintes sont ensuite sauvegardées sous la forme de méta-nœuds de manière à être interrogeables via des requêtes (voir image à droite : méta-nœud contenant des contraintes SHACL importées via n10s). Lorsqu'un appel est fait à n10s pour réaliser une validation, cette dernière sera traduite en requêtes Cypher sur base de l'information (*patterns* à détecter pour validation) contenue dans les méta-nœuds.

Par conséquent, puisque qu'une validation est obtenue via une série de requêtes sur un ensemble de *patterns*, effectuer une validation sur le graphe entier peut être est une opération lourde pouvant durer plusieurs minutes, voir heures, suivant la taille du graphe (par exemple des graphes contenant plusieurs de millions de nœuds) et le nombre de *patterns* à identifier (nombre de contraintes à valider). De façon alternative, comme discuté précédemment, ces règles de validation peuvent aussi être exécutées de manière transactionnelle, en les associant à un *trigger* APOC (un autre plugin Neo4J) afin que les contraintes de validation ne soient exécutées que lorsque du nouveau contenu est ajouté. Auquel cas, il est possible de faire en sorte que la validation ne soit exécutée que sur les données ajoutées durant la transaction, pour rendre le processus automatique et performant.

Bien que ce genre de validation puisse en pratique être remplacée par une série de requêtes Cypher écrites manuellement, on appréciera le fait de pouvoir facilement valider l'ensemble de contraintes d'un coup et d'obtenir un rapport récapitulatif qui liste les nœuds problématiques et la/les contrainte(s) qu'ils violent.

Concernant l'inférence automatique (rendre explicite des informations implicites) réalisée via n10s, celle-ci peut aussi être réalisée de plusieurs manières :

- Faire de l'inférence, soit globale, soit transactionnelle, afin d'ajouter l'information de façon explicite dans le graphe, grâce à l'utilisation de méta-nœuds et méta-relations pour stocker les règles d'inférence. L'idée est intéressante mais, en pratique, de simples requêtes Cypher pourraient être utilisées pour ce genre d'opérations, sans nécessiter l'ajout de méta-données dans le graphe.
- Faire de l'inférence à la volée (*on the fly*) de sorte que l'information implicite soit accessible via requêtes, mais sans que celle-ci ne soit pour autant explicitement ajoutée dans le graphe. Ce type d'inférence peut notamment être utile si les règles d'inférence sont dynamiques et changent fréquemment, et que, par conséquent, on ne souhaite pas que les résultats de l'inférence ne soient « matérialisés » dans le graphe. Conceptuellement intéressant, mais très niche dans la pratique. Difficile d'en trouver des applications concrètes.

Conditions d'utilisation & budget

Le package Neosemantics est gratuit, open source, téléchargeable depuis github, mais incompatible avec certaines versions de Neo4J.