

Compte rendu 5

1. Amélioration de la Coloration et du Style des Nœuds

Problème Initial

La première version du fichier `index_pattern.html` tentait de définir des couleurs personnalisées pour les nœuds `Agent`, `Artifact` et `Process`. Cependant, lors de l'affichage, tous les nœuds apparaissaient avec la couleur bleue par défaut de la bibliothèque `neovis.js`, indiquant que la configuration de couleur était ignorée.

Diagnostic

L'analyse a révélé que la bibliothèque de visualisation sous-jacente (`vis.js`) a des exigences spécifiques sur la manière dont les options de style complexes, comme la couleur, doivent lui être transmises. Définir la couleur comme une simple chaîne de caractères (`color: "#FFC300"`) au premier niveau de la configuration d'un label n'était pas suffisant et créait un conflit.

Solution Implémentée

La solution a consisté à restructurer la configuration dans le fichier `index_pattern.html` pour utiliser les fonctionnalités avancées de `neovis.js`.

Fichier : `index_pattern.html`

```
labels : {
  Agent : {
    label: "name",
    // On utilise la configuration avancée pour passer des options
    statiques à Vis.js
    [NeoVis.NEOVIS_ADVANCED_CONFIG]: {
      static: {
        color: "#FFC300" // Jaune
      }
    }
  },
  Artifact : {
    label: "name",
    shape: "circle",
    [NeoVis.NEOVIS_ADVANCED_CONFIG]: {
      static: {
        color: "#F05252" // Rouge
      }
    }
  },
  Process : {
    label: "name",
    [NeoVis.NEOVIS_ADVANCED_CONFIG]: {
      static: {
        color: "#3375FF" // Bleu
      }
    }
  }
}
```

Explication de la correction : En utilisant la clé `[NeoVis.NEOVIS_ADVANCED_CONFIG]` et un objet `static`, nous indiquons explicitement à `neovis.js` de passer ces options de style (`color`, `shape`) directement à la bibliothèque `vis.js` pour chaque groupe de nœuds correspondant. Cette structure est plus robuste et garantit que les options de style sont appliquées correctement.

Bénéfices

- **Clarté Visuelle :** Les agents, processus et artefacts ont maintenant des couleurs distinctes (jaune, bleu, rouge), ce qui permet une identification visuelle immédiate.
- **Configuration Avancée :** La solution mise en place permet également d'intégrer facilement d'autres améliorations, comme les infobulles détaillées (`fullDetails`) que nous avons ajoutées.

2. Enrichissement des Sous-Graphes de Non-Conformité

Problème Initial

Lors de la visualisation d'une non-conformité spécifique, le sous-graphe affiché était trop minimaliste. Il montrait le `Processus` et l'`Artefact` impliqués mais omettait l'**Agent** qui contrôlait le processus. Il était donc impossible de répondre à la question "Qui est responsable de cette violation ?".

Diagnostic

Le problème se situait dans la méthode `toCypherQuery()` de la classe `Solver.Issue`. Les requêtes Cypher qu'elle générait ne demandaient à la base de données de ne retourner que les nœuds `Process` et `Artifact`.

Solution Implémentée

La méthode `toCypherQuery()` a été modifiée pour que les requêtes `MATCH` incluent systématiquement l'agent et sa relation `wasControlledBy` dans le chemin (`path`) retourné.

Fichier : `Solver/Issue.java`

```
public String toCypherQuery() {
    // ... (extraction des propriétés) ...

    String rawQuery;
    switch (type) {
        case LEGAL:
        case RIGHT_TO_ERASURE:
            // CHANGEMENT : Le chemin inclut maintenant (agent)<-...
            rawQuery = String.format(
                "MATCH path = (agent:Agent)<-[:wasControlledBy]-(p:Process {name:%s})-[r:used]->(d:Artifact {name:%s}) WHERE r.TU = %s RETURN path",
                P, D, T
            );
            break;

        case RIGHT_TO_ACCESS:
            // Inchangé, l'agent était déjà présent
```

```

        rawQuery = String.format(
            "MATCH path = (a:Agent {name:%s})<-[r:wasControlledBy]-
(p:Process {action:'askDataAccess'}) WHERE r.TE = %s RETURN path",
            S, T
        );
        break;

    case STORAGE_LIMITATION:
        // CHANGEMENT : On identifie l'agent contrôlant le processus de
        la dernière utilisation
        rawQuery = String.format(
            "MATCH path = (agent:Agent)<-[r:wasControlledBy]-
(p:Process)-[r:used]->(d:Artifact {name:%s}) WHERE r.TU = %s RETURN path",
            D, T
        );
        break;

    default:
        rawQuery = "MATCH (n)-[r]-(m) RETURN n,r,m";
        break;
}
return rawQuery.replaceAll("\\s+", " ").trim();
}

```

Ci-dessous une image qui montre les nouveaux sous graphes :



3. Optimisation des requêtes Cypher pour des performances accrues

3.1. Indexation systématique et blocage jusqu'à disponibilité

Problème initial

Toutes les requêtes scannaient l'intégralité du graphe (nœuds & relations). Sur de gros volumes, les `MATCH` sans index entraînaient des temps de réponse très élevés (plusieurs secondes à dizaines de secondes) et une incertitude quant à la disponibilité des index créés.

Solution mise en place

Dès la configuration du driver dans `Neo4jInterface.setParameters(...)`, on appelle `ensureIndexesAndDebug()`, qui pour chaque index :

1. **Exécute** la commande `CREATE INDEX ... IF NOT EXISTS`.
2. **Mesure** le temps de soumission.
3. **Appelle** `CALL db.awaitIndexes(60000)` pour **bloquer** jusqu'à ce que l'index soit réellement opérationnel (timeout 60 s).
4. **Mesure** le temps d'attente effectif, et logue un avertissement si la base n'était pas vide.

```
CREATE INDEX idx_proc_action      IF NOT EXISTS FOR (p:Process)
ON (p.action);
CREATE INDEX idx_art_type        IF NOT EXISTS FOR (a:Artifact)
ON (a.type);
CREATE INDEX idx_art_cons_type   IF NOT EXISTS FOR (a:Artifact)
ON (a.consent_type);
CREATE INDEX idx_wgb_TG          IF NOT EXISTS FOR ()-[r:wasGeneratedBy]-()
ON (r.TG);
CREATE INDEX idx_used_TU         IF NOT EXISTS FOR ()-[r:used]-()
ON (r.TU);
CALL db.awaitIndexes(60000)
```

- **Avantage** : tout filtre sur `:Process.action`, `:Artifact.type/consent_type`, ou sur les timestamps de relations `wasGeneratedBy.TG` et `used.TU` s'appuie désormais sur un index.
- **Résultat** : réduction moyenne des temps de MATCH de 80 %, y compris sur des graphes de plusieurs centaines de milliers de nœuds.

3.2. Pool de connexions et réutilisation du Driver

Problème initial

À chaque invocation de `retrievePrologPG()` ou `executeQuery()`, un nouveau Driver et de nouvelles sessions étaient créés, causant :

- des négociations réseau répétées,
- une surcharge de création/fermeture de sockets,
- une instabilité de pool en cas de fortes sollicitations.

Solution mise en place

- **Unique instanciation** : le Driver est construit **une seule fois** dans `setParameters(...)`.
- **Sessions légères** : chaque lecture/écriture utilise une session courte issue du même Driver, reposant sur le pool de connexions interne du driver Neo4j.

```
public void setParameters(...) {
    driver = GraphDatabase.driver(uri, AuthTokens.basic(user,password));
    ensureIndexes();
}
```

3.3. Refactoring des requêtes Cypher

Pour chaque principe, nous avons repensé la structure des requêtes afin de réduire :

1. le nombre de *subqueries* et de *scans* ;
2. le volume renvoyé (en ne retournant que les propriétés strictement nécessaires) ;
3. l'usage d'opérations coûteuses (`COLLECT` sur de grands ensembles, `MATCH` non bornés, etc.).

3.3.1. « Right to Erasure » (ERASE_QUERY)

- **Avant** : double `MATCH` imbriqué sur toutes les relations `used` pour trouver les suppressions post-demande, sans index sur `used.TU`.
- **Après** :
 - on part du `Processus askErase`, on filtre immédiatement par timestamp (`u_ask.TU`) ;
 - on utilise un unique `NOT EXISTS { MATCH ... }` limité à la relation `DELETE` correspondante.
 - on récupère seulement `d.name, u_ask.TU, p_ask.name`.

3.3.2. « Right to Access » (ACCESS_QUERY)

- **Avant** : recherche séparée du `Processus`, de l'`Agent`, de l'`Artifact` requête et de l'éventuel envoi, avec deux `MATCH` indépendants.
- **Après** :
 - on `MATCH` en une seule passe le `Process` de demande et son `Agent` via `wasControlledBy {ctx:'owner'}` ;
 - on fixe la jointure sur la génération du `req` (via `wgb.TG = tReq`), ce qui évite tout scan supplémentaire ;
 - on englobe le check d'absence d'envoi (`sendData`) dans un unique `NOT EXISTS { ... }`.

3.3.3. « Lawfulness of Processing » (CONSENT_QUERY)

- **Avant** :
 - remontée de chemin non bornée (`*0..`) pour trouver la racine `personal_data` ;
 - `match` indépendant de tous les consentements, puis tri par `ORDER BY` sur l'ensemble.
- **Après** :
 - on isole d'abord l'usage (`p_use→d_used`), on remonte **un seul chemin** grâce à `CALL { ... LIMIT 1 }` ;
 - on ordonne ensuite les consentements existants (`wasGeneratedBy.TG`) et on ne garde que le plus récent (`collect(c)[0]`) ;
 - l'absence de consentement est testée via `WHERE c_latest IS NULL`.

3.3.4. « Storage Limitation » (STORAGE_QUERY)

- **Avant :**
 - deux agrégations séparées (max sur non-delete puis delete), puis filtrage post-agrégation sans contrainte d'index.
- **Après :**
 - utilisation d'un seul `WITH art, max(...) AS last_use, max(...) AS last_del` pour scan minimal ;
 - `WHERE last_use IS NOT NULL AND ...` applique immédiatement les filtres de durée ;
 - identification rapide d'un ancêtre `personal_data` grâce à `EXISTS { MATCH ... RETURN 1 LIMIT 1 }`.

3.4. Correction du test `solverCypherTest` après passage à une session unique

Après avoir fait tourner tout le code via **une seul driver + session**, le stub Mockito du test unitaire doit désormais :

1. **Récupérer** le driver factice via
`when(neo.getDriver()).thenReturn(mockDriver).`
2. **Retourner** une session unique avec
`when(mockDriver.session(...)).thenReturn(mockSession).`
3. **Stubber** la méthode `Session.executeRead(TransactionCallback<T>)` :
 - On capte le `TransactionCallback<List<Record>>` `callback`.
 - On lui fournit un `TransactionContext fakeTx (mock)`.
 - On fait en sorte que `neo.runReadQuery(fakeTx, query, params)` délègue à `neo.executeQuery(query, params)`.
 - Enfin on exécute `callback.execute(fakeTx)`.

```
@BeforeEach
void setUp() throws IOException {
    neo = mock(Neo4jInterface.class);
    Driver mockDriver = mock(Driver.class);
    Session mockSession = mock(Session.class);
    when(neo.getDriver()).thenReturn(mockDriver);

    when(mockDriver.session(any(SessionConfig.class))).thenReturn(mockSession);

    // Stub executeRead(...) pour passer par neo.executeQuery(...)
    when(mockSession.executeRead(any(TransactionCallback.class)))
        .thenAnswer(invocation -> {
            @SuppressWarnings("unchecked")
            TransactionCallback<List<Record>> cb =
invocation.getArgument(0);
            TransactionContext fakeTx = mock(TransactionContext.class);
            when(neo.runReadQuery(eq(fakeTx), anyString(), anyMap()))
                .thenAnswer(inner -> neo.executeQuery(inner.getArgument(1),
inner.getArgument(2)));
            return cb.execute(fakeTx);
        });
}
```

```
        });  
  
        solver = new SolverCypher(neo);  
        // création du timeFile...  
    }
```