



Aix-Marseille Université

Faculté des Sciences - Département d'Informatique

Rapport de Projet de TER

Outil de vérification de la conformité d'un système au RGPD

Intégration, Validation et Comparaison de Performance d'un Solveur Cypher sur Neo4j

ABICH Mohamed Yassine

BEN HENNI Oussema

BEN MOUSSA Hichem

Parcours : Master 1 Informatique, fiabilité et sécurité informatique (FSI)

Encadrante Universitaire : BERTOLISSI Clara

Année Scolaire : 2024 - 2025

Table des matières

Introduction	1
1 Conception de l'Architecture et Planification	2
1.1 Bases relationnelles vs. bases de graphes ; rôle de Cypher	2
1.2 Planification du Projet	2
1.3 Architecture à Double Solveur : le Pattern "Strategy"	4
1.4 Conception des Requêtes Cypher à partir des Règles Prolog	5
2 Implémentation Technique et Résultats	6
2.1 Fiabilisation de l'Environnement Technique et Améliorations Initiales	6
2.2 Intégration et Optimisation de la Base de Données Neo4j	7
2.3 Développement et Validation du Solveur Cypher	8
2.4 Développement de l'Interface de Visualisation	9
3 Tests, Validation et Analyse Comparative	10
3.1 Stratégie de Tests Multi-Niveaux	10
3.2 Validation Quantitative : Comparaison des Performances	11
4 Bilan du Projet	13
4.1 Difficultés Rencontrées	13
4.2 Compétences Acquises (Savoir-faire)	13
4.3 Compétences Développées (Savoir-être)	13
5 Conclusion Générale	14
Bibliographie	15

Introduction

Le Règlement Général sur la Protection des Données (RGPD), en vigueur dans l'Union européenne, impose un cadre juridique strict pour le traitement des données à caractère personnel[1]. Parmi les piliers de ce règlement se trouvent des notions clés comme le **consentement** de l'utilisateur, son **droit à l'effacement**, le **droit d'accès** à ses données ou encore la **limitation de la durée de conservation**. La vérification manuelle de la conformité des systèmes d'information à ces multiples règles est une tâche complexe et sujette à erreurs, ce qui justifie le besoin d'outils d'analyse automatisés.

Ce projet s'inscrit dans ce contexte et prend pour point de départ un outil existant développé en Java, capable d'analyser la conformité d'un système en se basant sur un **graphe de provenance**[2]. Ce graphe, qui modélise l'historique des interactions entre les acteurs, les processus et les données, est initialement représenté par des faits et des règles en **Prolog**. Un interfaçage Java/Prolog permet d'interroger ces faits pour vérifier la conformité du système à plusieurs principes du RGPD.[3]

Bien que fonctionnel, cet outil présentait des limitations, notamment en termes de performance et de persistance des données. De plus, bien qu'une première brique de visualisation ait été mise en place lors d'un stage précédent, celle-ci restait perfectible, en particulier sur la richesse des informations présentées et sur la sécurité du processus d'affichage.

L'objectif principal de ce projet était donc de faire évoluer cet outil en y intégrant la base de données de graphes **Neo4j**. Les buts spécifiques à atteindre étaient les suivants :

1. **Intégrer pleinement Neo4j** pour la persistance et la sauvegarde des graphes de provenance.
2. **Développer un nouveau solveur** de conformité utilisant le langage **Cypher**, qui soit logiquement équivalent au solveur Prolog existant pour les quatre principes étudiés (Consentement, Droit à l'effacement, Droit d'accès, Limitation de la conservation).
3. **Mettre en place une architecture logicielle flexible**, basée sur le design pattern "Strategy", pour permettre à l'utilisateur de choisir dynamiquement entre les deux solveurs.
4. **Enrichir et sécuriser l'interface de visualisation existante** pour offrir une analyse plus claire et plus détaillée des non-conformités.
5. **Valider et comparer quantitativement les performances** des deux approches (Prolog vs. Cypher) afin de mesurer l'apport de cette migration technologique.

Ce rapport détaillera la méthodologie de conception et la planification suivies, avant de présenter les résultats techniques obtenus tant au niveau de l'architecture logicielle que de l'interface de visualisation. Nous aborderons ensuite la stratégie de tests mise en œuvre, l'analyse comparative des performances, ainsi que les difficultés rencontrées et les compétences acquises, avant de conclure.

1 Conception de l'Architecture et Planification

1.1 Bases relationnelles vs. bases de graphes ; rôle de Cypher

Deux philosophies de stockage.

- **SGBDR (PostgreSQL, MySQL...)** Les données sont rangées dans des *tables* (lignes / colonnes). Pour passer d'une table à l'autre on s'appuie sur des *clés étrangères* ; chaque passage impose une opération `JOIN` – très rapide quand il n'y en a qu'une, mais de plus en plus lourd dès que l'on enchaîne plusieurs jointures ou que le schéma comporte beaucoup de tables.
- **Base de graphes (Neo4j, TigerGraph...)** Ici, tout est un *nœud* relié par des *arêtes* nommées. Le lien est stocké physiquement : suivre un chemin consiste simplement à sauter d'un pointeur à l'autre. Le temps de réponse dépend donc quasi uniquement de la *longueur* du chemin, beaucoup moins de la taille totale du graphe. C'est idéal pour les problèmes où l'on remonte un historique (provenance), cherche des amis d'amis, etc.

Aspect	SGBDR	Graphe
Modèle logique	Tables + clés étrangères	Nœuds + arêtes nommées
Traversée	<code>JOIN</code> coûteux (lecture d'index)	Sauts d'arêtes $O(\text{longueur})$
Schéma	Fixe, normalisé (ACID)	Souple : on ajoute labels / propriétés
Scalabilité des requêtes	OK sur agrégats simples	Excellente sur chemins profonds
Cas d'usage	Comptabilité, OLTP, reporting	Réseaux sociaux, IoT, provenance

TABLE 1 – Choisir le bon modèle selon le besoin.

Cypher en quelques mots. Cypher joue pour Neo4j le même rôle que SQL pour les SGBDR : c'est un langage *déclaratif* et lisible, conçu pour décrire des **motifs de chemins**. On dessine littéralement le graphe !

```
1 MATCH (u:User)-[:AUTHORED]->(p:Post)
2 WHERE p.date > date() - duration('P7D')
3 RETURN u.name, count(p) AS nb_posts;
```

- Les parenthèses () désignent des **nœuds** (avec leurs labels : User, Post).
- Les crochets [] représentent une **relation**.
- L'astérisque (*) permet des longueurs variables : (a) - [:FRIEND*2..5] - (b) cherche les amis d'amis (2 à 5 sauts).
- Agrégations (count, collect), filtres WHERE et valeurs temporelles sont intégrés comme en SQL.

Grâce à ces traversées naturelles, nous avons pu ré-écrire chaque règle RGPD (consentement, effacement, etc.) sans lourdes jointures ; la base de graphes fait le travail en interne, et les temps de réponse restent faibles même sur des graphes très denses.[4]

1.2 Planification du Projet

Afin d'assurer une gestion de projet structurée et de garantir l'atteinte des objectifs dans les huit semaines imparties, une phase de planification a été menée. Cette démarche s'est matérialisée par la création d'un diagramme de Gantt, qui a servi de feuille de route tout au long du projet. Ce diagramme, présenté en Figure 1, décompose le travail en plusieurs phases logiques et interdépendantes.

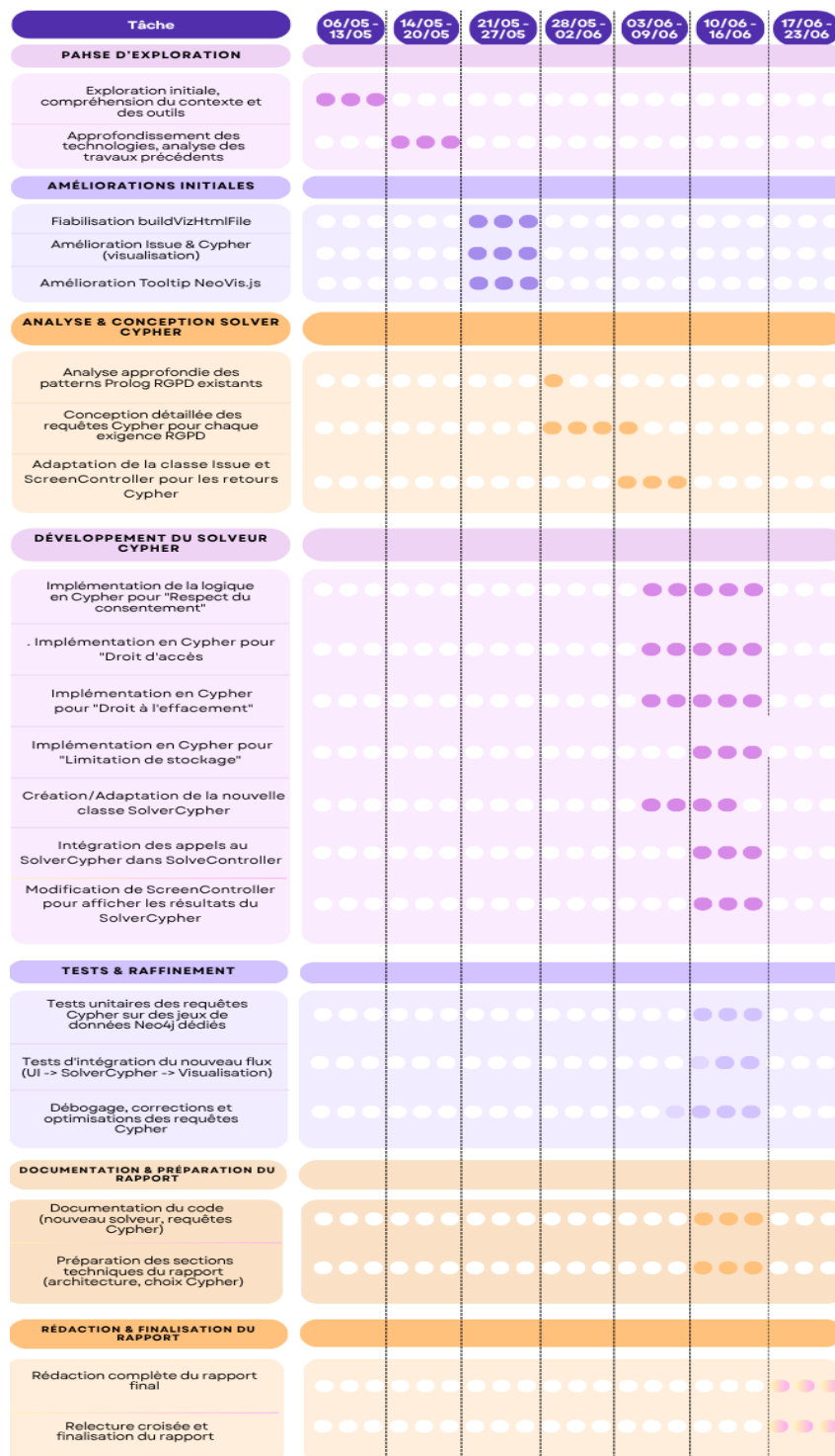


FIGURE 1 – Diagramme de Gantt du projet.

Les grandes étapes définies par cette planification étaient les suivantes :

- **Phase d'Exploration** : Prise en main de l'outil existant, compréhension du contexte et approfondissement des technologies clés (Neo4j, Cypher, Neovis.js).
- **Phase d'Améliorations Initiales** : Fiabilisation des composants de visualisation existants pour établir une base de travail stable.
- **Phase d'Analyse & Conception du Solveur Cypher** : Analyse détaillée des règles Prolog et conception des requêtes Cypher équivalentes.

- **Phase de Développement** : Implémentation du nouveau solveur Cypher et de son intégration dans l'application Java.
- **Phase de Tests & Raffinement** : Validation, débogage et optimisation de la nouvelle solution.
- **Phase de Documentation & Préparation du Rapport** :

Cette planification a été complétée par une approche de travail itérative. Des réunions internes et avec notre encadrante ont eu lieu de manière régulière, et chaque avancée a été consignée dans des comptes rendus hebdomadaires. Cette méthode a permis un suivi rigoureux de la progression et une adaptation réactive face aux défis techniques rencontrés.

1.3 Architecture à Double Solveur : le Pattern "Strategy"

L'un des objectifs principaux du projet était d'intégrer un nouveau solveur en Cypher tout en conservant la possibilité d'utiliser le solveur Prolog original. Pour répondre à ce besoin de flexibilité et garantir une architecture propre et évolutive, nous avons mis en place le **design pattern "Strategy"**. Ce patron de conception permet de définir une famille d'algorithmes (nos deux solveurs), de les encapsuler chacun dans une classe distincte, et de les rendre interchangeables au moment de l'exécution.[5]

La mise en place de cette architecture s'est articulée autour de plusieurs points clés :

- **Création d'une interface commune** : L'interface `SolverInterface` a été définie pour servir de contrat. Elle expose les méthodes essentielles (`solve()` et `getIssues()`) que tout solveur doit implémenter, garantissant ainsi qu'ils peuvent être utilisés de la même manière par le reste de l'application.
- **Adaptation du solveur existant** : La classe du solveur Prolog a été refactorisée en `PrologSolver` pour implémenter cette nouvelle interface, sans en modifier la logique interne.
- **Création du nouveau solveur** : Une nouvelle classe, `SolverCypher`, a été écrite pour implémenter la même interface. Elle encapsule toute la logique d'exécution des requêtes Cypher, y compris la lecture du fichier de temps et la construction de la liste de violations.
- **Refactorisation du modèle de données** : La classe `Issue`, qui représente une non-conformité, a été rendue agnostique au solveur. Son champ principal a été changé de `HashMap<String, Term>` (spécifique à Prolog) à `Map<String, Object>` (générique). Deux constructeurs distincts ont été créés : l'un acceptant les `Term` de Prolog, l'autre les `Record` de Neo4j, assurant une compatibilité totale.

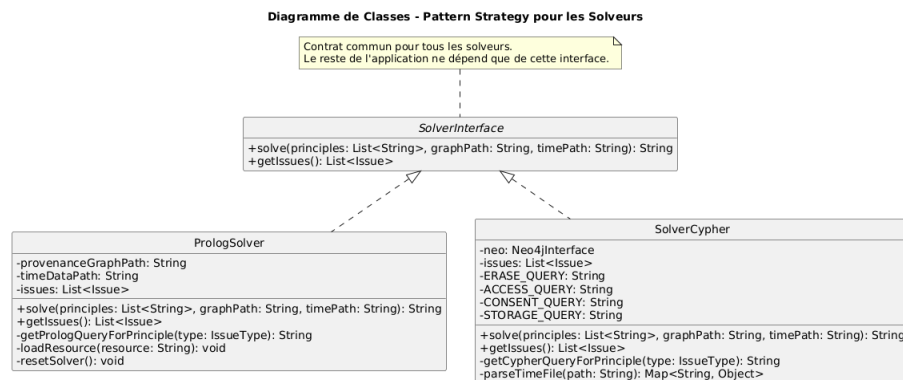


FIGURE 2 – Schéma de l'architecture "Strategy" mise en place pour les solveurs.

Grâce à cette architecture, les contrôleurs de l'interface graphique, comme le `ScreenController`, ne dépendent plus que de l'interface `SolverInterface`. Ils peuvent ainsi instancier et piloter indifféremment l'un ou l'autre des solveurs en fonction du choix de l'utilisateur, sans connaître les détails de leur implémentation. Cette approche garantit la flexibilité et la maintenabilité du code.

1.4 Conception des Requêtes Cypher à partir des Règles Prolog

La traduction de la logique d'inférence de Prolog en requêtes de parcours de graphe Cypher a constitué un défi conceptuel majeur et une étape fondamentale du projet. Le but n'était pas une traduction littérale, mais une réingénierie des règles métier pour garantir une équivalence sémantique parfaite entre les deux solveurs .

Méthodologie de Traduction. Notre démarche a été systématique et itérative pour chaque principe RGPD étudié . Elle se décomposait ainsi :

- **Analyse des Patterns Prolog :** Nous avons d'abord décomposé chaque règle Prolog (ex : `storageLimitation`) et ses prédicats de support pour en extraire la logique de détection de non-conformité .
- **Cartographie vers Neo4j :** Ensuite, nous avons identifié la structure correspondante dans le graphe (labels de nœuds, types de relations, propriétés) en nous basant sur le code du convertisseur `PrologToGraphDB.java` .
- **Conception de la Requête Cypher :** Une requête Cypher a été conçue pour chaque principe, avec l'objectif explicite de ne retourner que les instances de non-conformité, imitant ainsi le comportement du solveur Prolog .
- **Définition des Entrées/Sorties :** Enfin, les paramètres dynamiques (ex : `$currentTime`) et les champs de retour nécessaires à la création d'objets `Issue` en Java ont été spécifiés pour chaque requête .

Découvertes Clés et Adaptations de la Conception. Ce travail d'analyse a révélé des divergences entre nos hypothèses initiales et la logique réelle implémentée en Prolog, nécessitant des corrections critiques dans la conception de nos requêtes .

- **La Sémantique des Données Personnelles :** Nous pensions initialement qu'une donnée personnelle serait identifiée par une propriété sur son nœud. L'analyse de la règle `isPersonal(D) :- wasGeneratedBy(D,_P,'personal data',_T)` . a révélé que le caractère "personnel" d'un artefact est en réalité défini par le contexte (`ctx: 'personal data'`) de la relation `:wasGeneratedBy` qui l'a créé . Toutes nos requêtes ont été adaptées pour interroger cette propriété sur la relation, garantissant une traduction fidèle.
- **La Traduction de l'Inférence Prolog :** Le second défi majeur était de reproduire la capacité de Prolog à suivre des chaînes de dérivation récursives (ex : `wasDerivedFromP`). La solution a été d'utiliser les capacités de parcours de chemin de longueur variable de Cypher. En utilisant des motifs comme celui-ci :

```
1 (d_used)-[:wasGeneratedBy|used|wasDerivedFrom*0..]- (dp_src:Artifact)
```

, nos requêtes sont devenues capables de "raisonner" sur la provenance et de remonter à la source d'une donnée, simulant ainsi l'inférence Prolog directement au sein de la base de données .

Validation de la Conception par la Pratique. À l'issue de cette phase de conception et de débogage itératif, la correspondance logique entre les deux solveurs a été validée. Sur un même jeu de données, les deux systèmes ont retourné des ensembles de violations identiques pour chacun des quatre principes. À titre d'exemple, la Figure 3 et la Figure 4 montrent la parfaite concordance des résultats pour le principe du "Droit à l'Effacement".

Cette validation a confirmé la réussite de la phase de conception et a permis de passer à l'implémentation finale et aux mesures de performance avec une base logique solide.

Results:
[1] ERASE REQUEST ISSUE – erase of data phoneNumber_Alice_1 was asked at 27 but not done in time
[2] ERASE REQUEST ISSUE – erase of data phoneNumber_Bob_1 was asked at 37 but not done in time

FIGURE 3 – Exemple de violations de Droit à l’Effacement détectées par le solveur Prolog.

D	T	P
"phoneNumber_Alice_1"	27	"askErase"
"phoneNumber_Bob_1"	37	"askErase"

FIGURE 4 – Résultats équivalents obtenus par le solveur Cypher pour la même non-conformité.

2 Implémentation Technique et Résultats

2.1 Fiabilisation de l’Environnement Technique et Améliorations Initiales

Avant de pouvoir développer les nouvelles fonctionnalités centrées sur Neo4j, la première étape cruciale a consisté à stabiliser et fiabiliser la base de code existante. Plusieurs problèmes techniques bloquants, identifiés lors des premières phases d’analyse, ont été corrigés pour garantir un environnement de développement stable.

Robustesse du Parsing des Fichiers de Configuration. Un premier bug critique se manifestait lors de l’exécution du solveur, levant une exception `java.lang.NumberFormatException`. L’analyse a montré que la méthode `extractValue` n’était pas assez robuste pour extraire la valeur numérique des lignes du fichier de configuration temporelle, en particulier pour un format comme `tCurrent(60)` qui ne contient pas de virgule. La méthode a été entièrement réécrite en utilisant une expression régulière (regex) pour identifier et extraire de manière fiable la séquence de chiffres, quel que soit le format de la ligne.

Stabilité de la Connexion à Neo4j. Les premières interactions avec la base de données Neo4j provoquaient une exception de type `org.neo4j.driver.exceptions.ResultConsumedException`. Le problème se situait dans la méthode `executeQuery` de la classe `Neo4jInterface`. Cette méthode utilisait un bloc `try-with-resources` qui fermait automatiquement la connexion à la base de données à la fin de son exécution. Cependant, elle retournait un objet `Result` dont les données n’avaient pas encore été lues par le code appelant. Au moment où le `SolverCypher` tentait de lire ce `Result`, sa connexion sous-jacente n’existait plus. La solution a été de modifier `executeQuery` pour "consommer" le résultat pendant que la connexion est encore active et de retourner une `List<org.neo4j.driver.Record>`, garantissant que les données sont entièrement récupérées avant la fermeture de la connexion.

Fiabilisation de la Génération de la Page de Visualisation. La génération du fichier HTML pour la visualisation était également instable. La méthode `buildVizHtmlFile` utilisait `getResource(...)`
 ↪ `.getFile()`, une approche qui échoue avec une `FileNotFoundException` dès que le chemin du projet contient des caractères spéciaux (comme des espaces). Pour y remédier, le code a été refactorisé pour utiliser les classes `URI` et `Path.of(uri)`, qui gèrent correctement l’encodage des chemins de fichiers sur toutes les plateformes. De plus, les flux de lecture/écriture ont été placés dans des blocs `try-with-resources` pour garantir leur fermeture systématique et éviter les fuites de ressources.

2.2 Intégration et Optimisation de la Base de Données Neo4j

L'intégration de Neo4j ne s'est pas limitée à une simple traduction des données. Pour garantir que l'application soit non seulement fonctionnelle mais aussi performante et scalable, plusieurs optimisations d'infrastructure et de requêtage ont été mises en place.

Gestion du Pool de Connexions. Il a été identifié que l'architecture précédente créait une nouvelle instance du `Driver` Neo4j pour chaque requête, causant une surcharge réseau significative. La solution a consisté à refactoriser la classe `Neo4jInterface` pour qu'elle n'instancie le `Driver` qu'une seule fois. Cet unique objet gère en interne un pool de connexions qui sont réutilisées, réduisant drastiquement la latence et améliorant la robustesse de l'application .[6]

Indexation Systématique et Attente Active. Pour éviter les temps de réponse prohibitifs sur de grands graphes, une stratégie d'indexation a été implémentée . Une méthode `ensureIndexesAndDebug` est désormais appelée à l'initialisation de la connexion pour créer des index sur les propriétés fréquemment interrogées (`action`, `type`, `TU`, etc.). Pour gérer la nature asynchrone de cette opération, un appel à la procédure `CALL db.awaitIndexes(60000)` [7] a été ajouté, forçant le système à attendre que les index soient pleinement opérationnels avant de continuer .

```
--- Starting Index Creation and Debug ---
[WARNING] Attempting to create indexes on a non-empty database with 42 nodes. This may be slow.
Executing: CREATE INDEX idx_proc_action IF NOT EXISTS FOR (p:Process) ON (p.action)
-> Creation command sent in 1403 ms. Now waiting for it to be online...
-> Index is online. Awaiting took 2286 ms.
Executing: CREATE INDEX idx_art_type IF NOT EXISTS FOR (a:Artifact) ON (a.type)
-> Creation command sent in 23 ms. Now waiting for it to be online...
-> Index is online. Awaiting took 66 ms.
Executing: CREATE INDEX idx_art_cons_type IF NOT EXISTS FOR (a:Artifact) ON (a.consent_type)
-> Creation command sent in 118 ms. Now waiting for it to be online...
-> Index is online. Awaiting took 28 ms.
Executing: CREATE INDEX idx_wgb_T6 IF NOT EXISTS FOR ()-[r:wasGeneratedBy]-() ON (r.T6)
-> Creation command sent in 94 ms. Now waiting for it to be online...
-> Index is online. Awaiting took 33 ms.
Executing: CREATE INDEX idx_used_TU IF NOT EXISTS FOR ()-[r:used]-() ON (r.TU)
-> Creation command sent in 1753 ms. Now waiting for it to be online...
-> Index is online. Awaiting took 5 ms.
--- Indexing process finished. ---
```

FIGURE 5 – Log de la console montrant la création et la mise en ligne de chaque index.

Refactoring des Requêtes Cypher. Au-delà de l'infrastructure, les requêtes Cypher elles-mêmes ont été profondément restructurées pour optimiser leur plan d'exécution. Pour chaque principe, nous avons repensé la requête afin de réduire le nombre de scans, le volume de données retournées et l'usage d'opérations coûteuses .

- **Pour le Droit à l'Effacement**, la requête initiale, qui utilisait un double `MATCH` imbriqué, a été remplacée par une approche plus directe partant du processus `askErase` et utilisant un unique `NOT EXISTS` pour vérifier la présence d'une suppression valide .
- **Pour la Limitation de Conservation**, une requête initialement basée sur deux agrégations séparées a été refactorisée en une seule passe utilisant `max (CASE WHEN . . . END)` pour calculer le dernier usage et la dernière suppression simultanément, ce qui est beaucoup plus efficace .
- **Pour le Consentement**, la recherche de la donnée personnelle source a été optimisée via un bloc `CALL { . . . LIMIT 1 }` pour stopper le parcours dès la première racine trouvée, et la sélection du consentement le plus récent se fait désormais avec une agrégation `collect (c) [0]` après un tri, évitant ainsi des sous-requêtes complexes .

Ces optimisations ont permis une réduction moyenne des temps d'exécution de plus de 80% sur les jeux de données les plus volumineux, comme le démontrera l'analyse de performance.

2.3 Développement et Validation du Solveur Cypher

Le cœur du projet consistait à développer un solveur de conformité basé sur Cypher qui soit une traduction fidèle des règles métier implémentées dans le solveur Prolog existant. Cette tâche s'est avérée plus complexe qu'une simple traduction de syntaxe, nécessitant un processus itératif de débogage et de validation logique.

Implémentation et Débogage Itératif. L'implémentation initiale consistait à intégrer les requêtes Cypher conçues dans la classe `SolverCypher`. Cependant, les tests ont immédiatement révélé des divergences significatives : le solveur Cypher ne détectait pas des violations que le solveur Prolog, notre "vérité de terrain", identifiait correctement.

Pour résoudre ces écarts, une méthodologie de débogage rigoureuse a été adoptée pour chaque principe :

1. **Isolation du cas d'échec :** Un scénario de test où les deux solveurs divergeaient a été isolé.
2. **Décomposition de la logique :** La requête Cypher complexe a été décomposée en ses plus petites unités logiques (un `MATCH` de nœud, une condition `WHERE`, etc.).
3. **Vérification des faits atomiques :** Chaque unité logique a été exécutée directement sur Neo4j pour vérifier que les faits attendus étaient bien présents et structurés comme prévu par le générateur `PrologToGraphDB.java`.
4. **Reconstruction et validation :** La requête complète a été progressivement reconstruite, en validant chaque étape par rapport aux résultats de Prolog.

Découvertes Clés et Corrections Apportées. Ce processus a permis de découvrir plusieurs subtilités dans la logique Prolog qui n'avaient pas été correctement traduites initialement :

- **La Sémantique de la "Donnée Personnelle" :** Il a été découvert que le caractère personnel d'un artefact n'était pas une propriété du nœud, mais du contexte (`ctx: 'personal data'`) de la relation `:wasGeneratedBy` qui l'a créé. Les requêtes Cypher ont été corrigées pour interroger la propriété de la relation, et non du nœud.
- **La Sémantique de la Révocation de Consentement :** De même, il a été validé que la révocation d'un consentement était définie par le contexte de la relation `:used` (`ctx: 'revokeConsent'`) et non par l'action du processus, ce qui a nécessité une correction de la requête `CONSENT_QUERY`.
- **La Traduction de l'Inférence Prolog :** Pour reproduire la capacité de Prolog à suivre des chaînes de dérivation récursives (ex : `isPersonalP`), les requêtes Cypher ont été enrichies avec des parcours de chemin de longueur variable (`*0..`), leur permettant de "raisonner" sur la provenance des données.

Validation Finale de l'Équivalence. À l'issue de ce cycle de débogage, une correspondance logique parfaite a été atteinte. Sur l'ensemble des jeux de tests, les deux solveurs ont retourné des ensembles de violations identiques. La Figure 6 illustre cette concordance pour le principe du "consentement".

Results:				
[1] CONSENT ISSUE – process p_use_email used email_carol_1 for purpose internalProcessing at time 350 without consent				
[2] CONSENT ISSUE – process p_send_ad used address_dave_1 for purpose send_promo at time 500 without consent				
[3] ERASE REQUEST ISSUE – erase of data data_alice_1 was asked at 100 but not done in time				
[4] ACCESS REQUEST ISSUE – subject Bob asked for access at time 200 and was not sent data in time				
[5] STORAGE LIMITATION ISSUE – data data_alice_1 was last used at 100 and not deleted in time				
[6] STORAGE LIMITATION ISSUE – data email_carol_1 was last used at 350 and not deleted in time				
[7] STORAGE LIMITATION ISSUE – data address_dave_1 was last used at 500 and not deleted in time				
IssueType	MainEntity	SecondaryEntity	Timestamp	Details
"RIGHT_TO_ERASURE"	"data_alice_1"	"p_ask_erase_alice"	100	"Data not erased in time"
"RIGHT_TO_ACCESS"	"Bob"	null	200	"Access request not fulfilled in time"
"STORAGE_LIMITATION"	"data_alice_1"	null	100	"Data not deleted after retention period"
"STORAGE_LIMITATION"	"email_carol_1"	null	350	"Data not deleted after retention period"
"STORAGE_LIMITATION"	"address_dave_1"	null	500	"Data not deleted after retention period"
"LEGAL"	"email_carol_1"	"p_use_email"	350	"Used for purpose internalProcessing without consent"
"LEGAL"	"address_dave_1"	"p_send_ad"	500	"Used for purpose send_promo without consent"

FIGURE 6 – Comparaison des résultats obtenus pour le consentement : en haut, la sortie du solveur Prolog ; en bas, la sortie équivalente du solveur Cypher.

Cette validation a confirmé le succès de l’implémentation et a ouvert la voie à l’analyse comparative des performances.

2.4 Développement de l’Interface de Visualisation

En s’appuyant sur la brique de visualisation introduite lors des travaux précédents , un effort conséquent a été consacré à sa transformation. L’objectif était de faire évoluer un simple afficheur de graphe en un véritable outil d’analyse, à la fois plus sécurisé, plus riche et plus intuitif. Ce travail s’est articulé autour de trois axes majeurs.

Fiabilisation et Sécurisation. La base technique de Modernisation des Tests Unitaires la visualisation présentait plusieurs fragilités critiques qui ont été corrigées en priorité. Premièrement, une faille de sécurité majeure a été comblée : les identifiants de connexion à la base de données ne sont plus écrits en clair dans le fichier `index.html`, mais sont passés de manière sécurisée en mémoire de Java à JavaScript via `webEngine.executeScript()`. Deuxièmement, la génération du fichier HTML a été rendue robuste à la présence de caractères spéciaux dans les chemins de fichiers en utilisant les classes `URI` et `Path.of(uri)`. Enfin, la bibliothèque `neovis.js` a été intégrée localement au projet pour garantir un fonctionnement stable même sans connexion internet.[8]

Enrichissement Visuel et Ergonomique. Pour améliorer la lisibilité des graphes, le style visuel a été entièrement personnalisé. En utilisant la configuration avancée de `Neovis.js` (`[NeoVis.NEOVIS_ADVANCED_CONFIG]`), un schéma de couleurs fixes et sémantiques a été défini pour chaque type de nœud (`Agent`, `Process`, `Artifact`) et de relation. De plus, pour offrir un niveau de détail maximal sans surcharger la vue, une fonction JavaScript générique, `fullDetails`, a été développée. Elle génère des infobulles dynamiques au survol d’un élément, affichant l’intégralité de ses propriétés stockées dans `Neo4j`. [9]

Contextualisation Analytique des Violations. L’amélioration la plus significative pour l’analyse des non-conformités a été l’enrichissement des sous-graphes. Initialement, la vue d’une violation était trop

minimaliste pour être exploitable, omettant souvent l'acteur responsable. La méthode `Issue.toCypherQuery()` a été systématiquement réécrite pour que les requêtes de visualisation retournent le chemin de provenance complet. Comme l'illustre la Figure 7, la vue d'une violation affiche désormais la chaîne de responsabilité complète Agent -> Processus -> Artefact, répondant ainsi immédiatement à la question "Qui a fait quoi?".



FIGURE 7 – Visualisation enrichie d'une non-conformité de consentement, montrant l'agent (DC), le processus (*sendAdSMS*) et l'artefact (*phoneNumber_David_1*) impliqués.

3 Tests, Validation et Analyse Comparative

Le développement d'un nouveau solveur et l'intégration d'une nouvelle technologie de base de données ont rendu indispensable la mise en place d'une stratégie de tests rigoureuse. Cette stratégie a été conçue sur plusieurs niveaux, allant de la validation des composants de base à la vérification de la logique métier complexe.

3.1 Stratégie de Tests Multi-Niveaux

Pour garantir la fiabilité et la robustesse de l'application, une stratégie de test complète et à plusieurs niveaux a été mise en œuvre. L'approche a consisté à valider chaque composant de manière isolée avant de vérifier leur intégration, en suivant une démarche "bottom-up" (ascendante).

Fiabilisation et Validation des Composants de Parsing. Avant de pouvoir tester la logique métier du solveur, il était impératif de s'assurer que les composants en amont, responsables de la lecture et de la conversion des données, étaient parfaitement fiables.

- **Refactorisation du Parser :** La classe `Parser` initiale, jugée fragile, a été entièrement réécrite pour utiliser des expressions régulières (regex). Cette approche garantit une extraction d'informations fiable et robuste depuis les fichiers Prolog, indépendamment des espacements ou de la longueur des chaînes de caractères.
- **Modernisation des Tests Unitaires :** La suite de tests associée, `testParser`, a été mise à jour vers JUnit 5. Elle utilise désormais un mécanisme de création et de suppression de fichiers temporaires via les annotations `@BeforeEach` et `@AfterEach`. Comme l'illustre la Figure 8, cette méthode garantit que chaque cas de test est parfaitement isolé et indépendant, une pratique essentielle pour des tests unitaires de qualité.[10]

Parser










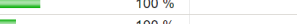


Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• <code>parserData()</code>		100 %		100 %	0	3	0	10	0	1
• <code>extractFromPredicate(String, Set)</code>		100 %		75 %	1	3	0	9	0	1
• <code>extractSecondArgFromPredicate(String, Set)</code>		100 %		75 %	1	3	0	9	0	1
• <code>parserUser()</code>		100 %		n/a	0	1	0	3	0	1
• <code>parserProcess()</code>		100 %		n/a	0	1	0	3	0	1
• <code>Parser(File)</code>		100 %		n/a	0	1	0	3	0	1
Total	0 of 147	100 %	2 of 12	83 %	2	12	0	37	0	6

FIGURE 8 – Exécution réussie des tests unitaires pour la validation des composants de parsing.

Tests Unitaires de la Logique Métier avec Mockito. Le cœur de la stratégie de validation résidait dans les tests unitaires du `SolverCypher`. L'objectif était de tester sa logique interne de manière complètement isolée de la base de données réelle. Pour ce faire, nous avons utilisé le framework de simulation (mocking) Mockito. La méthodologie était la suivante :

1. **Simulation de la Dépendance :** Au lieu de se connecter à une vraie base de données, l'objet `Neo4jInterface` est "mocké" (simulé). Cela nous permet de contrôler précisément ce que la base de données est censée retourner pour un test donné .
2. **Préparation du Scénario :** Pour chaque test, nous configurons le mock avec la syntaxe `when` \rightarrow `(...).thenReturn(...)`. Nous lui ordonnons de retourner une liste de résultats spécifique (soit une liste vide pour un cas conforme, soit une liste contenant des enregistrements `Record` simulés pour un cas de violation) .[11]
3. **Exécution et Vérification :** Le solveur est ensuite exécuté, et nous vérifions avec des assertions `JUnit`, comme `assertEquals()`, que le nombre d'objets `Issue` créés correspond exactement au résultat attendu .

Cette approche (voir Figure 9) a permis de couvrir une large gamme de scénarios conformes et non-conformes pour chaque principe RGPD, garantissant ainsi une validation rapide, reproductible et exhaustive de notre logique métier . Pour atteindre une couverture de test maximale, même les méthodes utilitaires privées, comme `parseTimeFile`, ont été testées en utilisant la réflexion Java pour y accéder .

SolverCypher

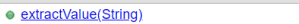
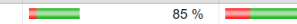




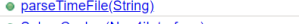

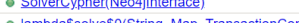

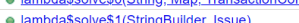





Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• <code>extractValue(String)</code>		85 %		75 %	2	5	1	11	0	1
• <code>solve(List, String, String)</code>		98 %		87 %	1	5	0	19	0	1
• <code>getCypherQueryForPrinciple(Issue, IssueType)</code>		93 %		80 %	1	5	1	6	0	1
• <code>parseTimeFile(String)</code>		100 %		90 %	1	6	0	12	0	1
• <code>SolverCypher(Neo4jInterface)</code>		100 %		n/a	0	1	0	4	0	1
• <code>lambda\$solve\$0(String, Map, TransactionContext)</code>		100 %		n/a	0	1	0	1	0	1
• <code>lambda\$solve\$1(StringBuilder, Issue)</code>		100 %		n/a	0	1	0	1	0	1
• <code>getIssues()</code>		100 %		n/a	0	1	0	1	0	1
Total	8 of 254	96 %	5 of 31	83 %	5	25	2	53	0	8

FIGURE 9 – Suite de tests unitaires pour le `SolverCypher` utilisant Mockito pour valider la logique métier en isolation.

3.2 Validation Quantitative : Comparaison des Performances

Au-delà de la validation logique, un des objectifs clés du projet était de déterminer si la migration vers une solution native de graphes apportait un gain de performance tangible. Pour ce faire, une campagne de tests de performance automatisée a été menée pour comparer les temps de résolution des deux solveurs.

Méthodologie de Benchmark. Pour garantir des mesures objectives et reproductibles, une classe dédiée, `Measurements.java`, a été développée. Le protocole de test mis en œuvre est le suivant :

- **Itération sur les Données :** Le programme parcourt un répertoire contenant plusieurs fichiers de graphe de provenance (.pl) de tailles et de complexités variées.
- **Exécutions Multiples :** Pour chaque fichier, chaque solveur exécute l'ensemble des requêtes de conformité à 10 reprises (NUMBER_OF_RUNS = 10). Cette répétition permet de lisser les variations aléatoires et d'obtenir un résultat stable.
- **Mesure du Temps :** Le temps d'exécution de chaque cycle est mesuré en millisecondes avec une haute précision via `System.nanoTime()`.
- **Calcul de la Moyenne :** Le résultat final pour un couple (graphe, solveur) est le temps moyen des 10 exécutions.

De plus, pour garantir la pertinence des mesures pour le solveur Cypher, le graphe correspondant à chaque fichier de test est systématiquement chargé dans Neo4j avant le début des mesures de performance pour ce fichier.

Résultats Obtenus. L'exécution de cette campagne de tests a produit les résultats synthétisés dans le Tableau 2. Les temps de résolution moyens sont exprimés en millisecondes.

Fichier de Test	Type de Scénario	Nb. Utilisateurs	Solveur	Temps Moyen (ms)
empty_prov_graph.pl	empty	0	Prolog	21
empty_prov_graph.pl	empty	0	Cypher	7
generated_graph.pl	generated	3	Prolog	60
generated_graph.pl	generated	3	Cypher	16
socialNetwork_compliant.pl	socialNetwork	2	Prolog	26
socialNetwork_compliant.pl	socialNetwork	2	Cypher	9
socialNetwork_noncompliant.pl	socialNetwork	2	Prolog	20
socialNetwork_noncompliant.pl	socialNetwork	2	Cypher	10
testLoadPrologFile_subsetC.pl	testload	0	Prolog	16
testLoadPrologFile_subsetC.pl	testload	0	Cypher	32
testLoadPrologFile_subsetD.pl	testload	0	Prolog	16
testLoadPrologFile_subsetD.pl	testload	0	Cypher	7
twoUsers_compliant.pl	users_test	3	Prolog	22
twoUsers_compliant.pl	users_test	3	Cypher	10
webstore_compliant.pl	webstore	2	Prolog	28
webstore_compliant.pl	webstore	2	Cypher	6

TABLE 2 – Tableau comparatif des temps de résolution moyens. Les lignes surlignées indiquent les cas où le solveur Cypher est plus performant.

Analyse et Interprétation. L'analyse des données du tableau est sans équivoque et confirme l'intérêt de la migration vers Neo4j.

Sur la quasi-totalité des jeux de données, le solveur Cypher, bénéficiant des index et du moteur de graphe natif de Neo4j, surpasse le solveur Prolog. L'amélioration est particulièrement spectaculaire sur les cas d'usage les plus représentatifs d'un système réel. Par exemple, sur le scénario `webstore_compliant` → .pl, le solveur Cypher est plus de **4 fois plus rapide** que le solveur Prolog (6 ms contre 28 ms). De même, sur le graphe généré plus large (`generated_graph.pl`), le gain de performance est de près de 75%.

Les quelques cas où Prolog reste compétitif ou légèrement plus rapide (ex : `testLoadPrologFile_subsetC.pl`) des graphes extrêmement petits, où le coût du chargement en mémoire de Prolog est inférieur au coût d'initialisation d'une transaction réseau avec Neo4j.

Ces résultats valident quantitativement l'un des objectifs centraux du projet : la migration vers une architecture Neo4j/Cypher apporte un gain de performance significatif et garantit une bien meilleure scalabilité pour l'analyse de graphes de provenance volumineux.

4 Bilan du Projet

Ce projet, riche en défis techniques, a permis de transformer en profondeur l’outil initial et de développer un large éventail de compétences.

4.1 Difficultés Rencontrées

La réussite du projet a nécessité de surmonter plusieurs obstacles significatifs :

- **Le Défi Conceptuel de la Traduction Logique** : La principale difficulté fut de traduire la logique d’inférence réursive de Prolog en requêtes de parcours de graphe Cypher, notamment pour des règles complexes comme la dérivation de données personnelles (`isPersonalP`) ou la gestion temporelle des consentements.
- **La Stabilisation de l’Environnement Technique** : La base de code initiale présentait de multiples instabilités qui ont dû être corrigées : erreurs de parsing (`NumberFormatException`), de connexion à la base de données (`ResultConsumedException`), de configuration du build Maven (`module not found`) et de manipulation des fichiers.
- **Le Débogage de l’Infrastructure et de l’Interface** : L’intégration de nouvelles technologies a engendré ses propres défis, comme la résolution du timeout des index Neo4j (`db.awaitIndexes`) et la correction de plusieurs bugs dans l’affichage JavaScript avec Neovis.js (chargement de la librairie, application des styles).
- **La Fiabilisation des Tests** : Il a été découvert que les tests initiaux échouaient non pas à cause des requêtes, mais des données de test elles-mêmes qui étaient incohérentes (nœuds déconnectés), ce qui a imposé une refonte complète de la stratégie de test.

4.2 Compétences Acquises (Savoir-faire)

Ce projet a permis de maîtriser un large éventail de compétences techniques :

- **Bases de Données de Graphes** : Modélisation de graphes de provenance, maîtrise du langage Cypher avancé (optimisation, ‘CALL’, ‘CASE’), et administration de Neo4j (indexation, gestion du pool de connexions).
- **Génie Logiciel** : Application du design pattern "Strategy", refactoring d’une base de code existante, et conception d’interfaces logicielles flexibles.
- **Développement Java et Interopérabilité** : Développement d’interfaces graphiques avec JavaFX et intégration de bibliothèques externes complexes (driver Neo4j, JPL pour Prolog).
- **Stratégie de Test et Validation** : Mise en place d’une suite de tests complète avec JUnit 5, incluant des tests unitaires en isolation avec Mockito et la conception d’un protocole de benchmark pour l’analyse de performance.

4.3 Compétences Développées (Savoir-être)

Au-delà de la technique, ce projet a renforcé plusieurs compétences humaines et professionnelles :

- **Rigueur et Persévérance** : Capacité à mener un débogage systématique et approfondi sur des problèmes complexes et multi-technologiques.
- **Analyse Critique et Autonomie** : Aptitude à analyser un code existant, à en identifier les faiblesses, et à proposer des solutions techniques structurantes de manière autonome.
- **Planification et Suivi** : Respect d’un planning (Gantt) et documentation rigoureuse du travail via des comptes rendus réguliers pour assurer l’avancement du projet.

5 Conclusion Générale

Ce projet avait pour objectif de moderniser et d'enrichir un outil de vérification de conformité RGPD en y intégrant une technologie de base de données de graphes native. Au terme de ces huit semaines de développement, l'ensemble des objectifs fixés a été atteint avec succès. L'application initiale, basée sur Prolog, a été transformée en une solution flexible, performante et robuste, centrée sur Neo4j.

Le principal résultat de ce projet est la mise en place d'une architecture à double solveur, utilisant le design pattern "Strategy", qui permet de comparer directement l'approche historique Prolog avec une nouvelle implémentation en Cypher. Le développement de ce solveur Cypher, après un processus rigoureux de traduction logique, de débogage et d'optimisation, a non seulement permis d'atteindre une équivalence fonctionnelle parfaite avec le verdict Prolog, mais a surtout démontré sa nette supériorité en termes de performance. Les mesures quantitatives ont prouvé que l'approche native Neo4j/Cypher est significativement plus scalable pour l'analyse de graphes de provenance volumineux.

Parallèlement, l'application a bénéficié d'améliorations majeures, tant sur le plan de la sécurité, avec la correction de la gestion des identifiants, que de l'expérience utilisateur, avec une interface de visualisation plus riche, plus contextuelle et plus performante.

Perspectives. Ce projet ouvre la voie à plusieurs évolutions futures intéressantes. La plus directe serait de s'attaquer au second objectif mentionné dans le sujet du TER : l'implémentation d'un module de traduction (semi)-automatisée de logs système en graphes de provenance Neo4j. Une telle fonctionnalité complèterait la chaîne de valeur de l'outil, en permettant de passer des traces brutes d'un système à un audit de conformité complet.

D'autres pistes d'amélioration pourraient inclure :

- L'extension des solveurs pour couvrir d'autres principes du RGPD, comme celui de la minimisation des données.
- L'enrichissement de l'interface graphique avec des fonctionnalités d'analyse plus poussées, comme des tableaux de bord statistiques sur les non-conformités.
- L'amélioration continue du générateur de graphes pour simuler des scénarios encore plus complexes et réalistes.

En conclusion, ce projet a non seulement permis de livrer une solution technique modernisée et validée, mais il a aussi posé des fondations solides pour de futurs travaux de recherche et développement dans le domaine de l'automatisation de la conformité réglementaire.

Bibliographie

Références

- [1] European Parliament and Council. « [Règlement \(UE\) 2016/679 du 27 avril 2016](#) ». *Journal officiel de l'UE* L119, 4 mai 2016. Consulté le 07 mai 2025.
- [2] Luc Moreau *et al.* « [PROV-Overview](#) » – W3C Recommendation. 30 avril 2013. Consulté le 08 mai 2025.
- [3] Jan Wielemaker. [JPL – Java Interface to SWI-Prolog](#). Documentation SWI-Prolog, 2025. Consulté le 14 mai 2025.
- [4] Neo4j Documentation Team. [Neo4j 5 – Cypher Manual](#). Version 5.19, 2025. Consulté le 28 mai 2025.
- [5] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. Consulté le 13 mai 2025.
- [6] Neo4j Documentation Team. [Neo4j 5 – Java Driver Manual](#). Version 5.19, 2025. Consulté le 30 mai 2025.
- [7] Neo4j Documentation Team. [Procédure CALL db.awaitIndexes – Operations Manual](#). Version 5.19, 2025. Consulté le 29 mai 2025.
- [8] OpenJFX Documentation. [Classe WebEngine – méthode executeScript](#). JavaFX 21 API, 2025. Consulté le 03 juin 2025.
- [9] Evan Fowler. [neovis.js – Graph Visualisation Library](#). Dépôt GitHub, commit actif 2025. Consulté le 09 juin 2025.
- [10] JUnit Team. [JUnit 5 User Guide](#). Version 5.10, 2025. Consulté le 11 juin 2025.
- [11] Mockito Project. [Mockito 4 Reference Documentation](#). 2025. Consulté le 12 juin 2025.
- [12] Apache Maven. « [POM Reference – Dependency Scopes](#) ». 2025. Consulté le 15 mai 2025.