Bases de données spécialisées - projet

Nicolas Graff & Steven Sailly

26 janvier 2024

1 Liminaire

La totalité de notre travail est disponible sur ce dépôt GitHub. Nous avons utilisé le driver Python Neo4j ainsi que Psycopg pour lancer les requêtes et afficher leurs résultats depuis notre code. L'installation des dépendances et le lancement des scripts sont expliqués dans le fichier `README.md`.

2 Données et modélisation

Les données proviennent de The Complete Pokemon Dataset, et sont fournies au format `csv`.

Le nettoyage des données s'est limité à séparer en plusieurs entrées les lignes dont certaines colonnes contenaient une liste d'entiers au lieu d'un seul entier, aussi ne sera-t-il pas détaillé.

Le fichier `csv` contient 41 colonnes :

- `pokedex_number` : un identifiant unique au Pokemon
- `classification` : une description textuelle du Pokemon
- `name` et `japanese_name` : le nom, en anglais et en japonais[1], du Pokemon
- `abilities` : les compétences d'un Pokemon
- `against_x` : 18 colonnes indiquant si le Pokemon est (très) résistant, (très) faible, insensible, ou sans résistance ni sensibilité contre le type `x`
- `type1` et `type2` : le ou les types du Pokemon
- `generation` : la génération dans laquelle le Pokemon est apparu[2]
- `is_legendary` : un booléen indiquant si un Pokemon est considéré comme "légendaire" ou non[3]
- 14 autres colonnes concernant les statistiques du Pokemon.

[1] À la fois en katakana et en rōmaji, ce qui ne pose de problème ni à Neo4j, ni à PostgreSQL, ni à nous.
[2] Voir la page Pokepedia dédiée.
[3] Voir la page Pokepedia dédiée.

2.1 Neo4j

2.1.1 Modélisation

Nous avons choisi comme types de nœuds ce qui pouvait ressembler à des clés primaires dans une base de données relationnelle — on verra d'ailleurs dans le schéma de la base de données PostgreSQL que les clés primaires choisies sont les mêmes que les types de nœuds.

- Les compétences (`Ability`) et les types (`Type`) ont un attribut `name`, le même que celui qui les identifiait dans le fichier csv.
- Les relations `HAS_ABILITY` représentent la colonne `ability`.
- Les relations `HAS_TYPE` représentent les colonnes `type1` et `type2`. Elles possèdent un attribut booléen `first` indiquant si la relation en question représente une colonne `type1` ou non.
- Les relations `AGAINST` représentent les 18 colonnes `against_x`. Cela signifie que chaque Pokemon est relié à chaque nœud `Type` par une relation de type `AGAINST`. Elles possèdent un attribut `value` indiquant la résistance d'un Pokemon face au type en question.
- Les autres colonnes sont représentées comme des attributs des nœuds `Pokemon`.
- Les attributs `name` et `pokedex_number` des nœuds `Pokemon` doivent être uniques.
- On ajoute un index sur l'attribut `name` des nœuds `Type`
- On définira plus tard une relation `STRONG_AGAINST` entre Pokemon (voir la définition donnée ci-après).

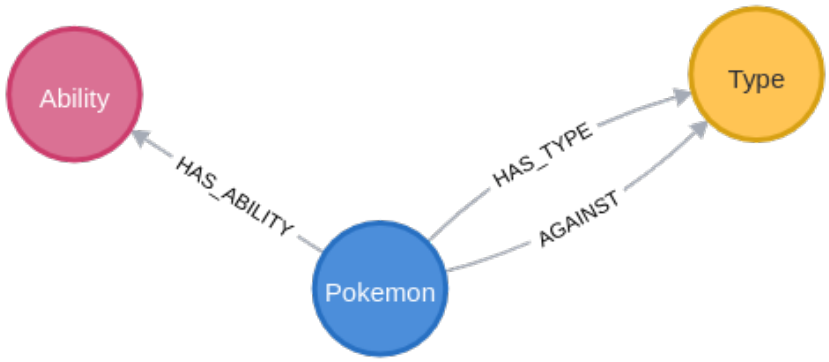


FIGURE 1 - Schéma de la base de données Neo4j.

2.1.2 Importation

Pour accéder aux valeurs des colonnes, on peut utiliser un accès par nom puisqu'on utilise la clause `LOAD CSV WITH HEADERS`. Les données étant suffisamment petites, on peut se passer d'une clause `USING PERIODIC COMMIT`.

On importe le fichier `csv` puis, pour chaque ligne, on crée le nœud `Pokemon` correspondant. Ici, on peut utiliser la clause `CREATE` car chaque ligne identifie un Pokemon différent. Pour les autres créations, on utilisera des clauses `MERGE`.

Les nœuds `Ability` sont initialement, pour chaque Pokemon, sous forme de tableau. On utilise une clause `UNWIND` sur la fonction `split`, on remplace les caracères inutiles, puis on crée si besoin le nœud `Ability` et la relation entre les nœuds `Ability` et `Pokemon`.

Chaque Pokemon a un type principal (colonne `type1`), donc on commence par créer ces nœuds et relations. Avant d'importer un éventuel type secondaire, il faut utiliser `WITH p, row WHERE row.type2 IS NOT NULL`. Comme cela signifie qu'on ne garde que les Pokemon qui ont un deuxième type, il faut garder cela pour la fin. La méthode est cependant la même que pour le type principal.

Comme on utilise le driver Python et qu'on connaît les noms des types, pour importer les 18 valeurs des colonnes `against_x`, on peut utiliser une boucle pour toutes les ajouter simplement. On notera que les variables utilisées à chaque itération doivent être différentes, et que le type "Combat" est appelé soit `fight` soit `fighting`, selon si on se trouve dans une colonne `type` ou dans une colonne `against_x`.

2.2 PostgreSQL

2.2.1 Modélisation

La base de données PostgreSQL comporte trois tables avec des clés primaires, identifiant respectivement un Pokemon, un type, ou une compétence.

On propose quelques dépendances fonctionnelles ayant permis de normaliser le schéma :

- `pokedex_id` $\rightarrow$ `pokemon.name`,`japanese_name`
- `pokedex_id` $\rightarrow$ `is_legendary`
- `type_id` $\rightarrow$ `type.name`
- `ability_id` $\rightarrow$ `ability.name`
- `pokemon_id` $\wedge$ `type_id` $\rightarrow$ `sensibility`

On ne vérifie pas *formellement* si le schéma suit une forme normale, mais on considère qu'il est *suffisamment normalisé* pour ce projet. Notons qu'il est possible que certaines requêtes aient été plus efficaces si elles avaient été évaluées en *dénormalisant* ce schéma, évitant ainsi des jointures.

La table `pokemon_strong` ne représente pas directement les données initiales, mais est nécessaire pour l'évaluation d'une requête que nous proposons. On définit la relation "plus fort que" $>$ sur l'ensemble des Pokemon telle que, si un Pokemon p_1 est résistant ou très résistant face à un type t, alors pour tout Pokemon p_2 ayant le type t : $p_1 > p_2$[4]. Ainsi, pour tout tuple `(pid_1, pid_2)` de la table `pokemon_strong`, `pid_1 > pid_2`.

[4]On remarquera qu'il ne s'agit pas d'une relation d'ordre.

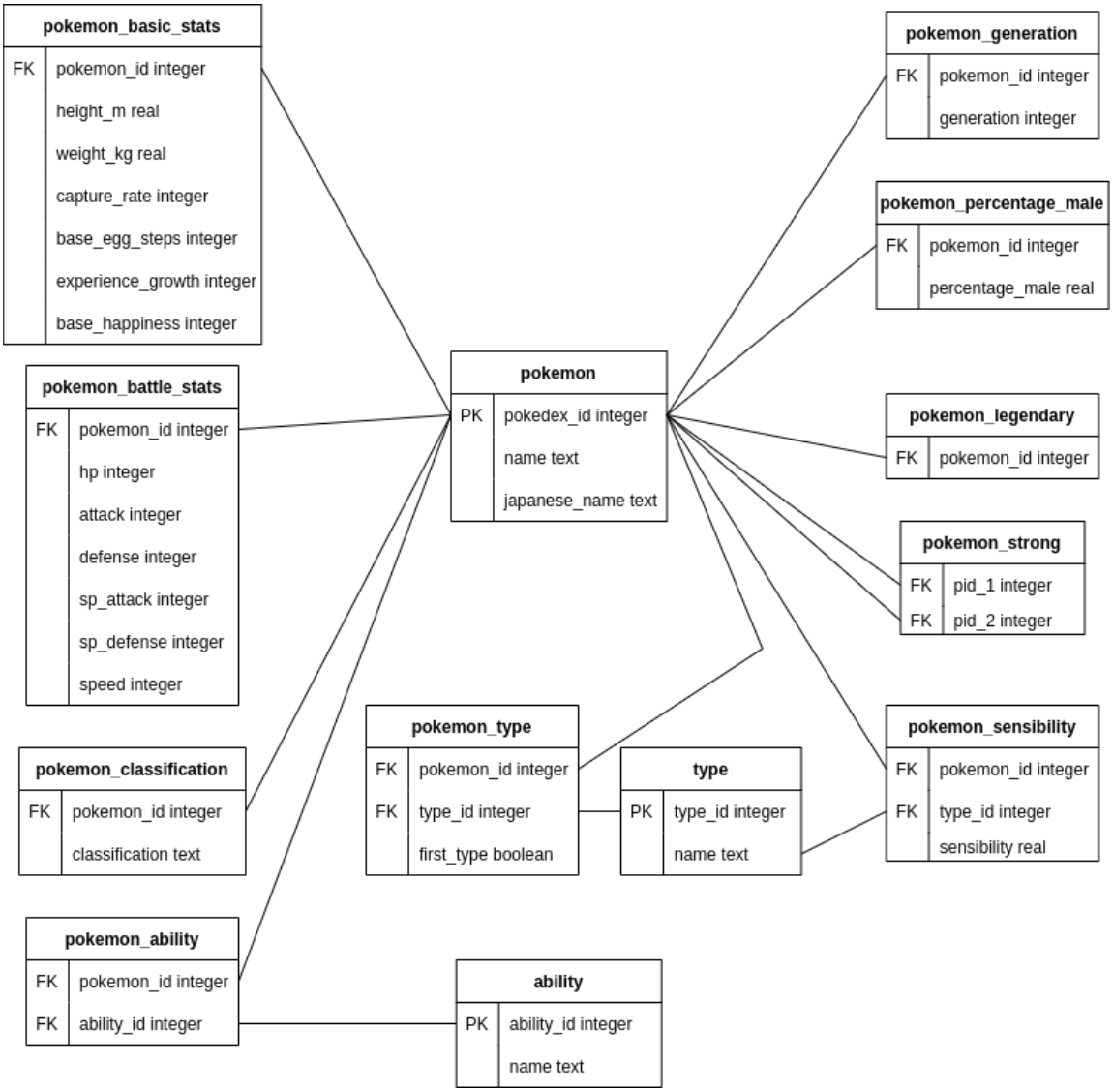


FIGURE 2 - Schéma de la base de données PostgreSQL.

2.2.2 Importation

On crée une table temporaire reprenant les mêmes colonnes que le fichier `csv`, puis on crée chaque table conformément au schéma. Comme la colonne `abilities` dans le fichier `csv` est un tableau de la forme `"['ability1', 'ability2']"`, on transforme les crochets en accolades pour les importer effectivement comme un tableau (type `TEXT[]`) dans la base de données. On peuple ensuite les tables à partir de la table temporaire (`INSERT INTO table(...) SELECT ... FROM tmp ...`).

Pour pouvoir réutiliser les clés, on crée d'abord les tables `pokemon`, `type`, et `ability`.

Pour créer un tuple dans la table `ability` pour chaque élément distinct des tableaux `tmp.abilities`, on utilise la fonction `unnest`.

Pour peupler la table `pokemon_sensibility`, on itère sur les noms des types, puis on fait une jointure entre les tables `tmp` et `ability` pour ne garder que l'*identifiant* de la compétence. On opère similairement pour peupler les tables `pokemon_type` et `pokemon_ability`.

3 Requêtes et analytique de graphe

La totalité des requêtes est disponible dans les fichiers Python. La description d'une requête est disponible en tant que documentation de la fonction l'utilisant. On écrit et exécute les requêtes Cypher depuis `neo4j-queries.py`, et les requêtes SQL depuis `postgres-queries.py`. Pour (presque) chaque requête Cypher dans une fonction `my_query1()` dans `neo4j-queries.py`, il existe la requête SQL équivalente dans la fonction `my_query1()` dans `postgres-queries.py`. On notera particulièrement que les fonctions `data_and_topo()` des deux fichiers peuvent être longues à évaluer.

Nous proposons un court descriptif de ces requêtes :

- Nombre de Pokemon qui ne sont pas faibles contre le feu et pas forts contre l'eau
- Résistances des Pokemon de type Psy
- Nombre de Pokemon (très) faibles face au type Psy qui ont une compétence donnée
- Somme des attributs "attaque" des Pokemon (très) faibles contre les types Feu, Eau, ou Plante, et dont le nom commence par A
- Pokemon qui sont immunisés contre plus d'un seul type
- Paires de Pokemon ayant un type commun, immunisé contre un type (qui peut ne pas être le même), et dont un et seulement un de leurs noms ou de celui de leur type commun commence par F ou G
- 10 Pokemon les plus lourds, et 10 Pokemon les plus légers
- Cycles de quatre ou cinq Pokemon forts les uns contre les autres tels qu'aucun Pokemon dans le cycle ne soit plus fort que le premier

3.1 Plans d'exécution

3.1.1 Plans d'exécution avec et sans index

Les plans pour la requête "Nombre de Pokemon qui ne sont pas faibles contre le feu et pas forts contre l'eau" peuvent être trouvés dans les annexes A.1 et A.2. Ces deux plans diffèrent par la présence d'un index qui a été ajouté à la propriété `value` de la relation `AGAINST`. La première requête est nettement plus efficace, avec moins d'accès à la base de données et un temps d'exécution plus rapide :

- Sans index : 900352 total db hits in 306 ms;
- Avec index : 52450 total db hits in 72 ms.

En effet, certaines opérateurs de requête avec un index nécessitent beaucoup moins de db hits : par exemple, NodeIndexSeek - la recherche de nœuds à l'aide des indices - utilise environ 20 fois

Operator	DB hits avec index	DB hits sans index	Différence	Estimated rows avec index	Estimated rows sans index
NodeIndexSeek	1604	35250	21,976	802	0
NodeIndexSeek	1230	27030	21,976	748	0
Expand	20031	440932	22,012	802	0
Expand	15360	338128	22,014	748	0
Filter	802	17625	21,976	40	0
Filter	615	13515	21,976	37	0

moins d'opérations et crée des "estimated rows"; les opérateurs Expand et Filter ont un comportement similaire :

3.1.2 Plans d'exécution des requêtes équivalentes mais avec des syntaxes différentes

Pour comparer les requêtes avec différentes syntaxes qui donnent le même résultat, nous utilisons la requête "Nombre de Pokemon (très) faibles face au type Psy qui ont une compétence donnée" avec ou sans l'utilisation de `collect unwind`. Les deux peuvent être trouvés dans les annexes A.3 et A.4.

Une requête avec `collect unwind` a un plus grand temps d'exécution - 98ms contre 50ms sans `collect unwind`. Le nombre de db hits est commun pour les deux requêtes - 4654.

On peut conclure que l'utilisation de list n'est pas pratique pour cette requête et ne fait qu'augmenter le temps d'execution.

L'exemple de plan d'exécution suivant utilise la requête "10 Pokemon les plus lourds, et 10 Pokemon les plus légers". Les résultats sont dans les annexes A.5 et A.6. Une requête avec `collect` supplémentaire est plus rapide (110 ms contre 144 ms) et moins coûteuse en mémoire (14,056 total memory (bytes) contre 264,232 total memory (bytes)). Les types des Pokémon sont collectés une fois après l'`UNION`, tandis que dans la deuxième requête, les types sont collectés individuellement pour chaque partie de l'`UNION`.

3.2 Comparaison de l'efficacité Cypher - SQL

On désignera les requêtes par le nom des fonctions qui les utilisent dans les scripts.

Les requêtes n'ont pas été exécutées sur la même machine que précédemment, d'où les différences de temps d'exécution pour une même requête.[5] Entre chaque exécution de requête Cypher, on exécute `CALL db.clearQueryCaches`.

3.2.1 `negative_filter()`

- Cypher : Exécution en 1ms
- SQL : non implémenté

[5] En revanche, les comparaisons sont effectuées entre requêtes exécutées sur la même machine.

3.2.2 `optional_match()`

- Cypher : Exécution en moins d'1ms
- SQL : Exécution en 3ms

3.2.3 `collect_unwind()`

- Cypher : Exécution en 1 (`collect_unwind_variant()`) à 70ms
- SQL : Exécution en 6ms

3.2.4 `reduce()`

- Cypher : Exécution en moins d'1ms
- SQL : Exécution en 16ms

3.2.5 `with_filter_aggregate()`

- Cypher : Exécution en 1ms
- SQL : Exécution en 7ms

3.2.6 `predicate_function()`

- Cypher : Exécution en 100ms
- SQL : Exécution en 90ms

3.2.7 `post_union_processing()`

La requête SQL est ***bien plus efficace*** que la requête Cypher.

- Cypher : Exécution en 90 (`post_union_processing_variant()`) à 110ms
- SQL : Exécution en 5ms

3.2.8 `data_and_topo()`

La requête SQL est une ***requête récursive***.

- Cypher : Il est plus raisonnable de l'évaluer avec une clause `LIMIT`. En fixant la limite à 20 000 résultats, la requête est évaluée en environ 10s.
- SQL : Trop gourmande en mémoire, comme l'indique l'intervention de l'OOM killer si l'on n'utilise pas de clause `LIMIT`, avec une limite bien plus faible que celle imposée au-dessus. On ne peut pas l'évaluer complètement.

3.3 Analytique de graphe

Pour chacune des deux sous-sections, on propose l'utilisation de deux algorithmes.

3.3.1 Détection de communautés

On choisit des algorithmes de détection de communautés afin de savoir *quels Pokemon se ressemblent*. Ils sont exécutés sur un graphe contenant tous les nœuds et toutes les relations, excepté celles de type `STRONG_AGAINST`. On inclut dans ces communautés les compétences et les types, qui permettent de connaître les points communs entre les Pokemon d'une même communauté.

Notons que les résultats présentés sont ceux d'une exécution *particulière* de chacun de ces algorithmes, bien que jugée suffisamment représentative de l'ensemble des exécutions menées.

- Louvain : exécuté sur un graphe *orienté*, 142 communautés (taille minimale : 1 ; taille maximale : 508 ; taille moyenne : 7,4 ; écart-type : 46,3)
- Louvain bis : exécuté sur un graphe *non-orienté*, 14 communautés (taille minimale : 46 ; taille maximale : 134 ; taille moyenne : 74,8 ; écart-type : 29,3)
- Leiden : exécuté sur un graphe *non-orienté*, 13 communautés (taille minimale : 53 ; taille maximale : 125 ; taille moyenne : 80,5 ; écart-type : 26,1)

On remarque que les communautés sont formées plutôt en suivant les *types* dans le cas du graphe non-orienté. Dans le cas du graphe orienté, même si les communautés sont encore formées en suivant les types, les *capacités* ont cette fois un rôle bien plus important dans leur formation.

3.3.2 Plus courts chemins

On exécute les algorithmes de plus courts chemins sur un graphe contenant uniquement les nœuds de type `Pokemon` et les relations de type `STRONG_AGAINST`. On cherche à connaître la *distance moyenne* et la *plus grande distance minimale* entre deux Pokemon.

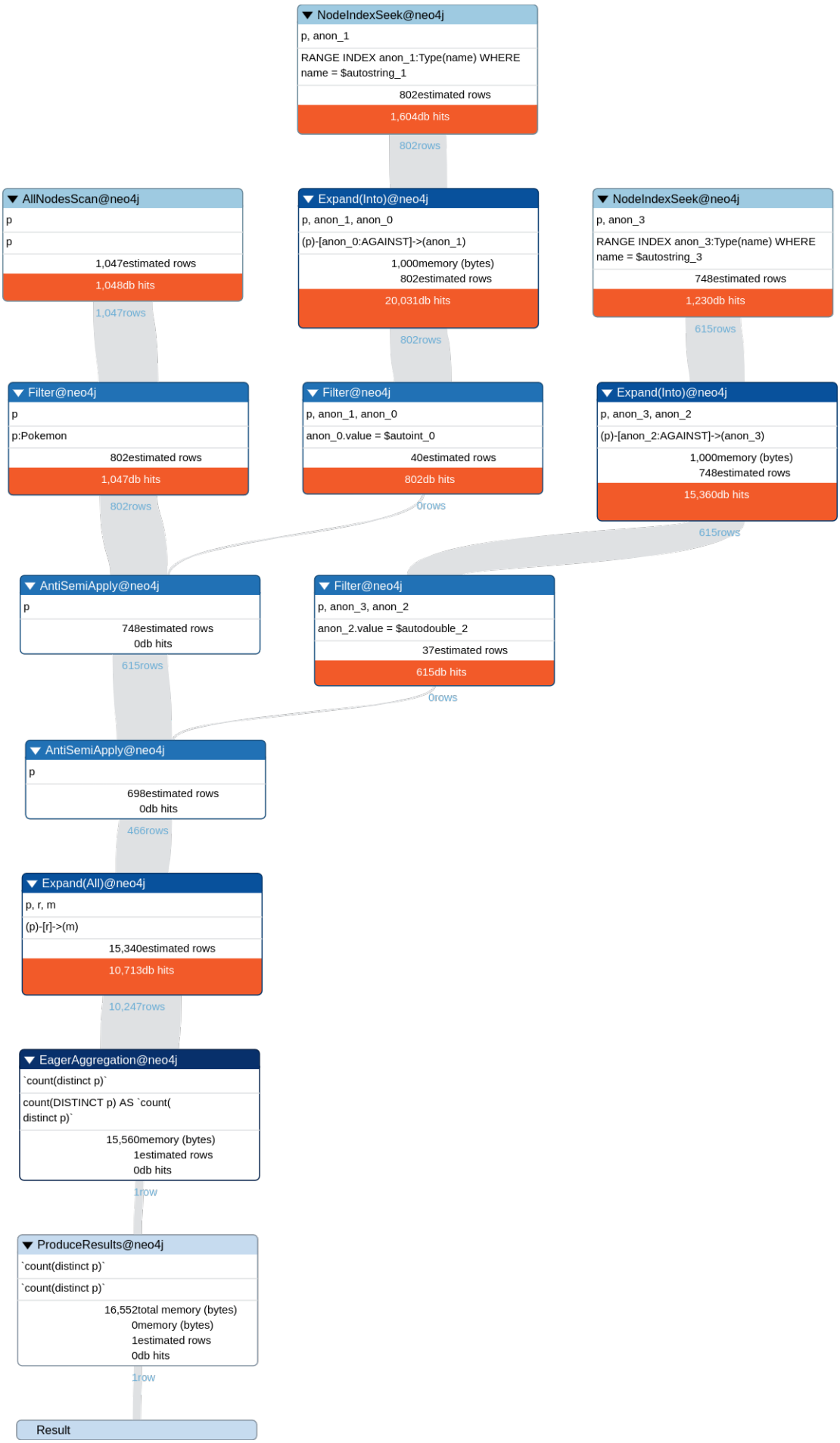
- `allShortestPaths` : exécuté en 2578ms
- `Dijkstra` : exécuté en plus de trois minutes

La différence des temps d'exécution s'explique par le fait que `Dijkstra` ne soit pas optimisé pour calculer sur toutes les paires en même temps, et par le fait qu'on utilise le mode `stream` dans le premier cas alors qu'on utilise le mode `write` dans le second.

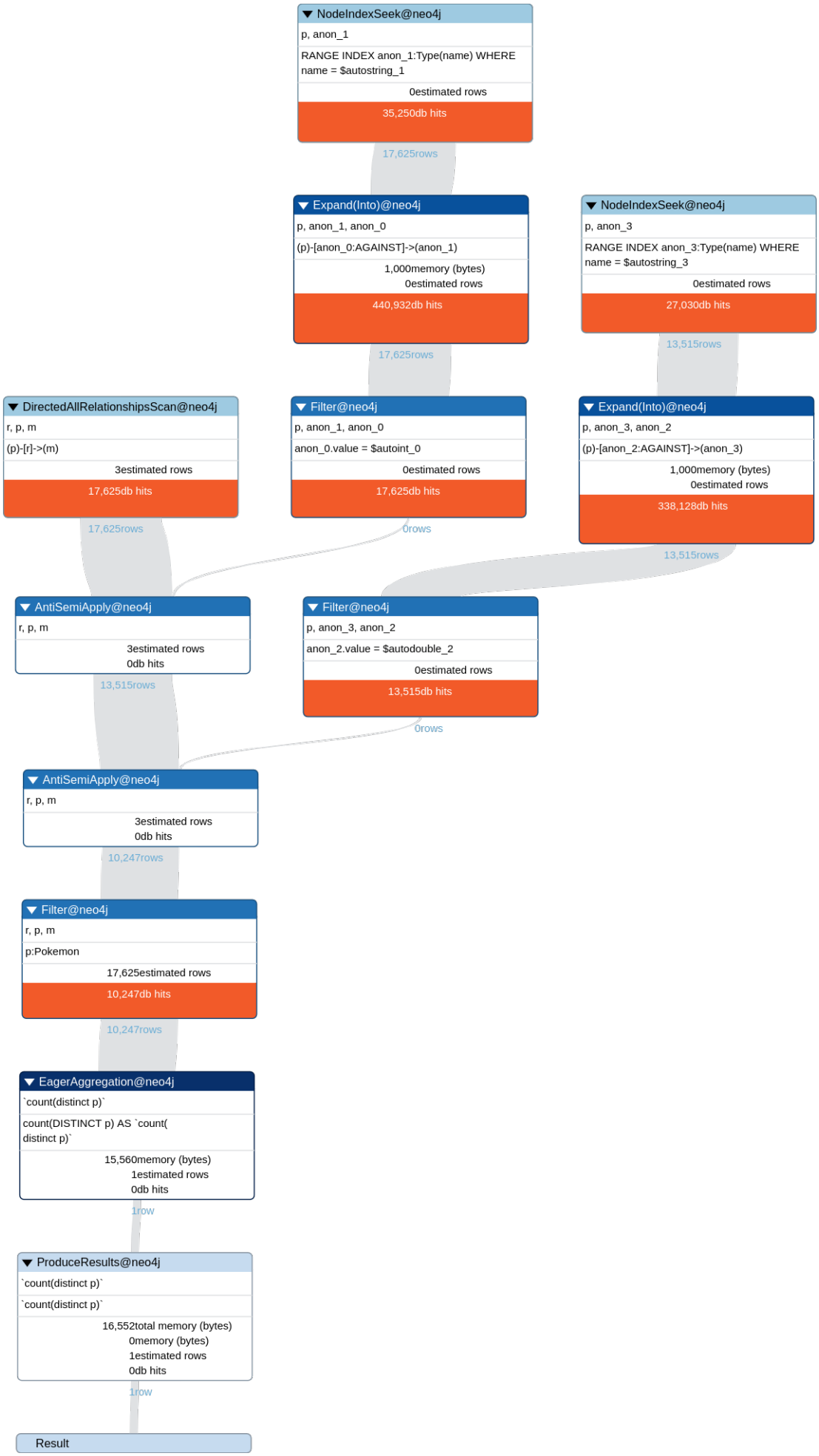
- Longueur maximale d'un PCC : 3
- Longueur moyenne d'un PCC : 1,66

Annexe A

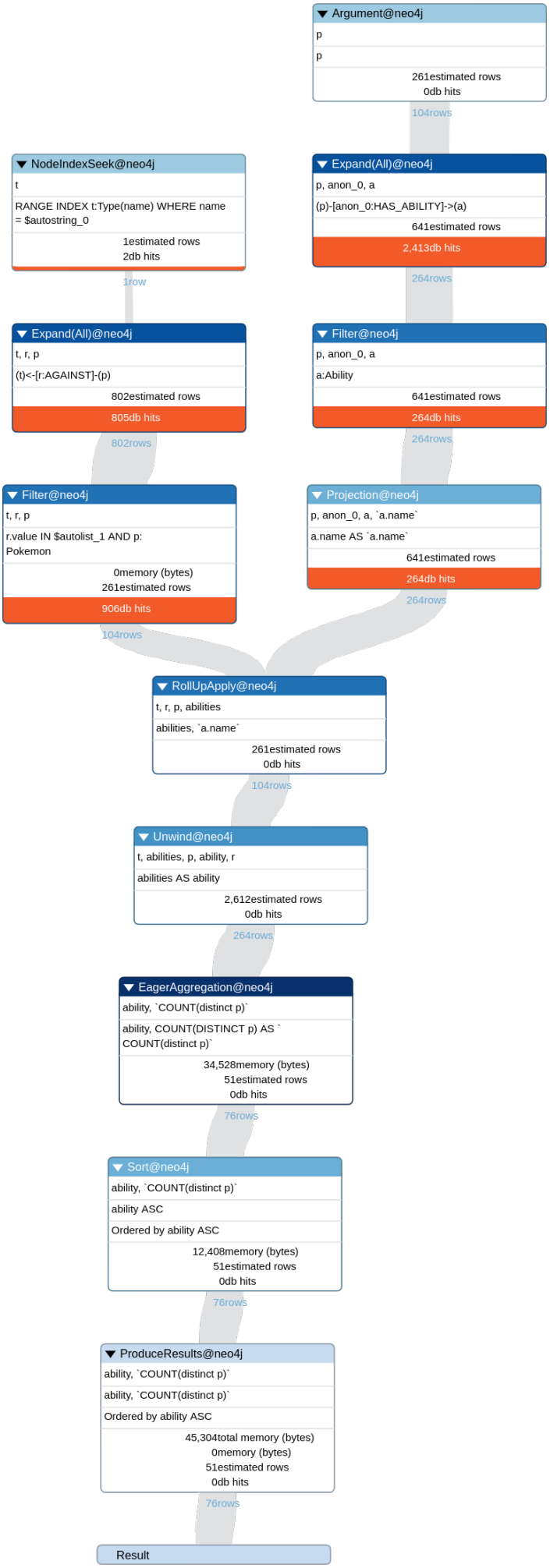
A.1 : Plan d'exécution avec index



A.2 : Plan d'exécution sans index

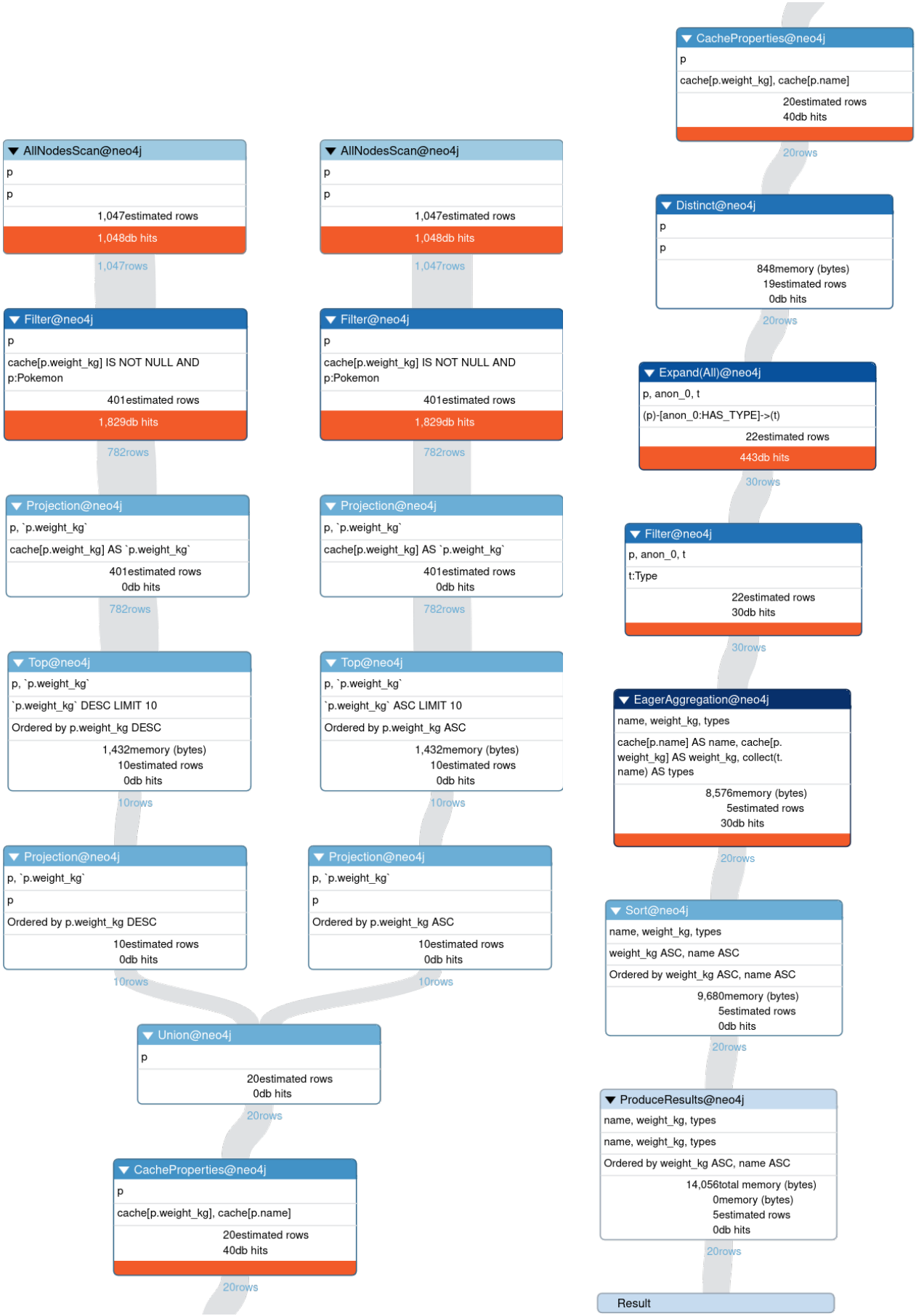


A.3 : Plan d'exécution de requête avec collect unwind



A.4 : Plan d'exécution de requête sans collect unwind

A.5 : Plan d'exécution de requête avec union et match



A.6 : Plan d'exécution de requête avec union sans match