MAC - Méthode d'accès aux données

Bases de données orientées graphe

09 November 2025

Table des matières

1 Concepts fondamentaux

1.1 Les graphes sont partout

Le monde est un graphe connecté: personnes, lieux, événements, etc. Les graphes sont présents dans tous les domaines (sciences, technologies, éducation, art).

1.2 Structure de données de graphe

Un **graphe** est un type de donnée abstrait composé de:

- **Sommets** (nœuds): les entités du graphe
- **Arêtes** (relations): les connexions entre paires de sommets

1.3 Bases de données orientées graphe

Une base de données orientée graphe utilise des structures de graphes pour représenter et interroger les données via:

- **Nœuds**: entités avec propriétés (paires clé-valeur)
- **Arêtes**: relations dirigées et nommées entre nœuds
- **Propriétés**: attributs des nœuds et relations

Avantages clés:

- Les relations sont des éléments de première classe
- Traversée très rapide des relations
- Relations stockées (non calculées à la requête)

Cas d'usage: réseaux sociaux, moteurs de recommandation, détection de fraudes

1.4 Graphes vs Bases relationnelles

1.4.1 Différences fondamentales

Bases relationnelles:

- Relations calculées au moment de la requête via JOIN coûteux
- Performance dégradée avec données hautement connectées

Bases de données graphes:

- Relations stockées directement avec les données
- Performance constante indépendante de la taille des données
- Excellentes pour données hautement connectées et requêtes complexes

1.4.2 Exemple: réseau social

Expérience sur 1′000′000 de personnes avec 50 amis chacune, recherche d'amis à profondeur 5:

Profondeur	RDBMS (s)	Neo4j (s)	Résultats
2	0.016	0.01	2500
3	30.267	0.168	110′000
4	1543.505	1.359	600′000
5	Unfinished	2.132	800′000

1.5 Modèle Property Graph

1.5.1 Composants

Nœuds:

- Entités du graphe
- Contiennent des propriétés (paires clé-valeur)
- Peuvent avoir 0 à plusieurs labels (catégories)

Relations:

- Associations dirigées et nommées entre deux nœuds
- Ont un type, une direction, un nœud source et destination

- Peuvent avoir des propriétés
- Une relation doit toujours relier deux nœuds

1.5.2 Stockage natif

Bases de données graphes natives:

- Stockage natif de graphes dès la conception
- **Index-free adjacency**: relations stockées, pas calculées
- Avantage de performance significatif

Bases de données graphes non-natives:

- Sérialisent les données dans des BD relationnelles ou autres

Attention: Les index sont toujours nécessaires pour accélérer l'accès aux nœuds de départ d'une traversée.

2 Neo4j et le langage Cypher

2.1 Introduction à Neo4j

2.1.1 Qu'est-ce que Neo4j?

- Base de données graphe native open source NoSQL
- Développement initial: 2003, public depuis 2007
- Code source en Java et Scala
- Éditions communautaire et entreprise

2.1.2 Caractéristiques de Neo4j

- **Cypher**: langage de requête déclaratif optimisé pour les graphes (projet openCypher)
- **Performance**: parcours à temps constant dans les grands graphes
- **Flexibilité**: schéma de graphe flexible évolutif
- **Pilotes**: Java, JavaScript, .NET, Python, etc.
- **Scalabilité**: jusqu'à des milliards de nœuds

2.1.3 Convention de nommage Cypher

- **Labels de nœuds**: CamelCase, majuscule initiale (`:VehicleOwner`)
- **Types de relations**: MAJUSCULES, tiret bas (`:OWNS_VEHICLE`)
- **Autres** (propriétés, variables, paramètres): camelCase, minuscule initiale (`title` , `businessAddress`)

2.2 Syntaxe de base

2.2.1 Représentation des nœuds

```
()                    // nœud anonyme
(p:Person)            // nœud avec label Person, variable p
(work:Company)        // nœud avec label Company, variable work
(:Technology)         // label Technology, sans variable
```

2.2.2 Représentation des relations

```
(a)-->(b)                    // relation dirigée
(a)-[r]->(b)                 // avec variable r
(a)-[r:REL_TYPE]->(b)        // avec type de relation
(a)-[r {prop: value}]->(b)   // avec propriété
```

Important: Direction obligatoire à la création, mais peut être ignorée lors des recherches.

2.2.3 Propriétés

```
(p:Person {name: 'Jennifer'})  // propriété de nœud
(p1)-[rel:FRIENDS {since: 2018}]->(p2)  // propriété de relation
```

2.3 Opérations CRUD

2.3.0.1 CREATE - Création de données

```
// Créer un nœud
CREATE (movie:Movie {title: 'The Matrix', released: 1999})

// Créer une relation
CREATE (actor:Person {name: 'Keanu Reeves'})
CREATE (actor)-[:ACTED_IN {roles: ['Neo']}]->(movie)
```

2.3.1 MATCH et RETURN - Interrogation

```
// Tous les titres de films
MATCH (movie:Movie)
RETURN movie.title

// Acteurs d'un film spécifique
MATCH (:Movie {title: 'The Matrix'})<-[:ACTED_IN]-(actor)
```

```
RETURN actor.name

// Retourner tout
MATCH p = (person)-[r]->(m)
RETURN *
```

2.3.2 WHERE - Filtrage

```
// Filtrage simple
MATCH (movie:Movie)
WHERE movie.released >= 1990 AND movie.released < 2000
RETURN movie.title

// WHERE dans le motif de nœud
MATCH (movie:Movie WHERE movie.released >= 1990)
RETURN movie.title

// Utilisation de NOT
MATCH (p:Person)
WHERE NOT p.name = 'Jennifer'
RETURN p

// Test d'existence de propriété
WHERE p.birthdate IS NOT NULL
WHERE rel.startYear IS NULL
```

2.3.3 SET et DELETE - Mise à jour et suppression

```
// Mise à jour
MATCH (p:Person {name: 'Jennifer'})
SET p.birthdate = date('1980-01-01')

// Suppression de relation
MATCH (j)-[r:IS_FRIENDS_WITH]->(m)
DELETE r

// Suppression de nœud avec relations
MATCH (m:Person {name: 'Mark'})
DETACH DELETE m

// Suppression de propriété
REMOVE n.birthdate
// ou
SET n.birthdate = null
```

2.3.4 MERGE - Éviter les doublons

Opération « sélection ou insertion »:

```
// Fusion de nœud
MERGE (mark:Person {name: 'Mark'})

// Fusion de relation
MATCH (j:Person {name: 'Jennifer'})
MATCH (m:Person {name: 'Mark'})
MERGE (j)-[r:IS_FRIENDS_WITH]->(m)

// Avec ON CREATE / ON MATCH
MERGE (keanu:Person {name: 'Keanu Reeves'})
ON CREATE SET keanu.created = timestamp()
ON MATCH SET keanu.lastSeen = timestamp()
```

2.4 Index et contraintes

2.4.1 Création d'index

```
// Index simple
CREATE INDEX FOR (a:Actor) ON (a.name)

// Index composite
```

```
CREATE INDEX FOR (a:Actor) ON (a.name, a.born)

// Inspection des index
SHOW indexes YIELD labelsOrTypes, properties, entityType
```

Utilité: Accélérer la recherche des points de départ d'un parcours de graphe.

2.4.2 Contraintes

```
// Contrainte d'unicité
CREATE CONSTRAINT FOR (movie:Movie)
REQUIRE movie.title IS UNIQUE

// Inspection des contraintes
SHOW constraints YIELD type, labelsOrTypes, properties
```

Note: Une contrainte d'unicité crée implicitement un index.

3 Requêtes avancées en Cypher

3.1 Fonctions d'agrégation

3.1.0.1 Fonctions de base

```
// Moyenne, max, min, sum
MATCH (p:Person)
RETURN avg(p.age), max(p.age), min(p.age), sum(p.age)

// Comptage
MATCH (p:Person {name: 'Keanu'})-[r]->()
RETURN type(r), count(*)

// Collection
MATCH (p1:Person)-[:KNOWS]->(p2:Person)
RETURN p1.name, collect(p2.name) AS acquaintances
```

3.1.1 DISTINCT et regroupement

```
// Comptage avec DISTINCT
MATCH (p:Person)-->(friend)-->(friendOfFriend)
WHERE p.name = 'Keanu'
RETURN friendOfFriend.name,
       count(DISTINCT friendOfFriend),
       count(friendOfFriend)

// Avec regroupement
MATCH (p1:Person)-[:KNOWS]->(p2:Person)
RETURN p1.name, max(p2.age) as ageOfOldestFriend
```

3.2 Clauses de transformation

3.2.0.1 WITH - Calculs intermédiaires

```
// Reporter toutes les variables + nouvelles
MATCH (person)-[r]->(otherPerson)
WITH *, type(r) AS connectionType
RETURN person.name, otherPerson.name, connectionType

// Filtrage sur agrégation
MATCH (david {name: 'David'})--(other)-->()
WITH other, count(*) AS connections
WHERE connections > 1
RETURN other.name
```

3.2.1 UNWIND - Transformer liste en lignes

```
// Recherche sur liste de technologies
WITH ['Graphs', 'Query Languages'] AS techReqs
UNWIND techReqs AS technology
MATCH (p:Person)-[:LIKES]-(t:Technology {type: technology})
RETURN t.type, collect(p.name) AS candidates

// Créer liste distincte
WITH [1, 1, 2, 2] AS coll
UNWIND coll AS x
WITH DISTINCT x
RETURN collect(x) AS setOfVals
```

3.2.2 ORDER BY et DISTINCT

```
// Tri des résultats
WITH [4, 5, 6, 7] AS experienceRange
UNWIND experienceRange AS number
MATCH (p:Person)
WHERE p.yearsExp = number
RETURN p.name, p.yearsExp
ORDER BY p.yearsExp DESC
```

```
// Résultats uniques
MATCH (user:Person)
WHERE user.twitter IS NOT null
WITH user
MATCH (user)-[:LIKES]-(t:Technology)
WHERE t.type IN ['Graphs', 'Query Languages']
RETURN DISTINCT user.name
```

3.3 Motifs de chemins

3.3.0.1 Longueur variable

```
// Un ou plusieurs sauts
MATCH (a)-[:KNOWS*]->(b)

// Exactement 3 sauts
MATCH (a)-[:KNOWS*3]->(b)

// Entre 3 et 5 sauts
MATCH (a)-[:KNOWS*3..5]->(b)

// 3 sauts ou plus
MATCH (a)-[:KNOWS*3..]->(b)

// 5 sauts ou moins
MATCH (a)-[:KNOWS*..5]->(b)

// Nommer un chemin
MATCH p = (a)-[:KNOWS*3..5]->(b)
```

3.3.1 Chemins les plus courts

```
// Films et acteurs jusqu'à 3 sauts
MATCH p = (bacon:Person {name: "Kevin Bacon"})-[*1..3]-(hollywood)
RETURN p

// Plus court chemin (Bacon number)
MATCH p = shortestPath(
  (bacon:Person {name: "Kevin Bacon"})-[*]-
  (meg:Person {name: "Meg Ryan"})
)
RETURN p
```