BD-Spécialisées

Examen de Bases de données spécialisées

Partie 2/3 - *Property Graph model* et Neo4j

15 Décembre 2021

Documents autorisés : trois feuilles A4 simples manuscrites ou imprimées (en tout pour les 3 parties de l'examen), en plus de la refcard Cypher. Portables/Ordinateurs/Tablettes interdits.

Question 1

Présentez brièvement un cas d'usage dans lequel il serait avantageux d'utiliser une base de données orientée graphe basée sur du *stockage graphe natif*, plutôt qu'une base de données relationnelle. Argumentez bien votre réponse (e.g., efficacité de l'évaluation des requêtes).

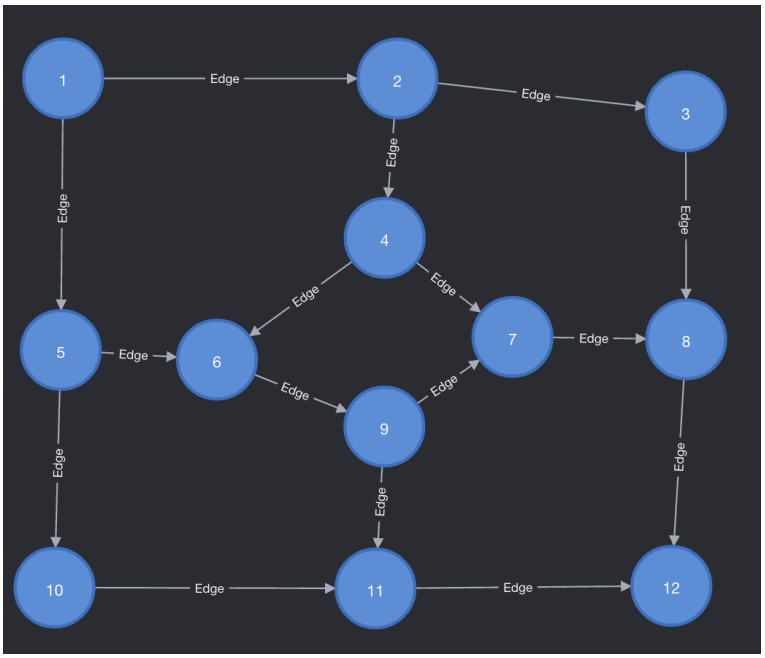
Écrire ici la réponse :

Question 2

Soit le graphe généré par le code Cypher suivant :

```
CREATE (v1:Vertice {name:1})
CREATE (v2:Vertice {name:2})
CREATE (v3:Vertice {name:3})
CREATE (v4:Vertice {name:4})
CREATE (v5:Vertice {name:5})
CREATE (v6:Vertice {name:6})
CREATE (v7:Vertice {name:7})
CREATE (v8:Vertice {name:8})
CREATE (v9:Vertice {name:9})
CREATE (v10:Vertice {name:10})
CREATE (v11:Vertice {name:11})
CREATE (v12:Vertice {name:12})

CREATE
  (v1)-[:Edge{name:1}]->(v2),
  (v2)-[:Edge{name:2}]->(v3),
  (v1)-[:Edge{name:3}]->(v5),
  (v2)-[:Edge{name:4}]->(v4),
  (v3)-[:Edge{name:5}]->(v8),
  (v5)-[:Edge{name:6}]->(v10),
  (v8)-[:Edge{name:7}]->(v12),
  (v10)-[:Edge{name:8}]->(v11),
  (v11)-[:Edge{name:9}]->(v12),
  (v5)-[:Edge{name:10}]->(v6),
  (v6)-[:Edge{name:11}]->(v9),
  (v9)-[:Edge{name:12}]->(v7),
  (v4)-[:Edge{name:13}]->(v6),
  (v4)-[:Edge{name:14}]->(v7),
  (v9)-[:Edge{name:15}]->(v11),
  (v7)-[:Edge{name:16}]->(v8)
```



Evaluation d'une première requête

La requête suivante retourne-t-elle des résultats ? Si oui, donnez un exemple de valeur possible pour `r1` et `r2`.

```
MATCH p1 = (s1 {name:1}) -[ :Edge*]- (t1 {name:12})
WITH [r in relationships(p1) | r.name] as r1
MATCH p2 = (s2 {name:1}) -[:Edge*]- (t2 {name:12})
WHERE none(r in relationships(p2) WHERE r.name in r1)
RETURN r1, [r in relationships(p2) | r.name] as r2
```

Réponse :
Pour chaque couple de chemins $p1, p2$ du noeud de nom 1 (propriété *name*) au noeud de nom 12, tel que les deux chemins n'empruntent aucune arrête en commun, la requête retourne deux ensembles : l'ensemble $r1$ des noms d'arrête sur le premier chemin $p1$ d'une part et l'ensemble des noms d'arrêtes $r2$ sur le second chemin $p2$ d'autre part. Par exemple $r1 = [1, 2, 5, 7]$ et $r2 = [3, 6, 8, 9]$.

Evaluation d'une seconde requête

La requête suivante retourne-t-elle des résultats ? Si oui, donnez un exemple de valeur possible pour `r1` et `r2`.

```
MATCH p1 = (s1 {name:1}) -[ :Edge*]- (t1 {name:12})
WITH [r in relationships(p1) | r.name] as r1
MATCH p2 = (s2 {name:3}) -[:Edge*]- (t2 {name:10})
WHERE none(r in relationships(p2) WHERE r.name in r1)
RETURN r1, [r in relationships(p2) | r.name] as r2
```

Réponse :
La requête ne retourne aucun résultat. Il n'y a aucun couple de chemins reliant ces deux paires de noeuds et n'empruntant aucune arrête en commun.

Ecriture de requête : comptage

Ecrivez une requête Cypher comptant le nombre de chemins du noeud 1 au noeud 12 (abstraction faite de l'orientation du graphe). Pouvez-vous commenter le choix de conception sous-jacent derrière le langage Cypher (sémantique des chemins) ?

Réponse :

```
MATCH p = (s1 {name:1}) -[ :Edge*]- (t1 {name:12})
RETURN COUNT(p)
```

La sémantique des chemins implémentée dans Cypher restreint les chemins considérés aux *trails* : il est impossible de passer deux fois par la même arrête, ce qui évite de considérer les chemins comportant des cycles (et donc d'obtenir des ensembles de chemins infini) et permet donc de compter les chemins.

Modifiez votre requête pour qu'elle retourne le nombre de chemins *de longueur paire* du noeud 1 au noeud 12 (abstraction faite de l'orientation du graphe). Rappel : en Cypher la division modulo s'écrit avec %.

Réponse :

```
MATCH p = (s1 {name:1}) -[ :Edge*]- (t1 {name:12})
WHERE length(p)%2=0
RETURN COUNT(p)
```

Ecriture de requête : ajout d'une arrête au graphe

Ecrivez une requête Cypher ajoutant une arrête de type `Edge` allant du noeud 1 au noeud 6.

Réponse :

```
MATCH (s {name:1}), (t {name:6})
CREATE (s)-[:Edge]->(t)
```

ou bien (dans le cas où il serait possible que l'arrête existe déjà) :

```
MATCH (s {name:1}), (t {name:6})
MERGE (s)-[:Edge]->(t)
```

Ecriture de requête : intégrité des données

On souhaite que la propriété de noeud `name` identifie de manière unique chaque noeud du graphe. Comment faire ?

Écrire ici la réponse :

```
CREATE CONSTRAINT name_key FOR (v:Vertice)
REQUIRE p.name IS NODE KEY
```

Complexité de l'évaluation des requêtes Cypher

Le problème de décision suivant (*k edge disjoint path*) est NP-complet :
Entrée : un entier k, un graphe dirigé G et deux noeuds s, t de G.
Question : existe-t-il k chemins différents de s à t n'ayant aucune arrête en commun ?
Est-il possible de conclure quelque chose de cette information quant à la complexité de l'évaluation du langage Cypher ? Argumentez votre réponse.

Réponse :

Le problème est NP difficile. Tout instance du problème de décision ci-dessus peut en effet être réduit à un problème de validité d'une requête Cypher sur un graphe de propriété. Il suffit de prendre le graphe G et de décorer ses noeuds avec des valeurs de données disjointes deux à deux pour une propriété *name*, idem pour les arrêtes. Soit G_{name} le graphe ainsi obtenu. Soit la requête vue précédemment :

```
MATCH p1 = (s1 {name:1}) -[ :Edge*]- (t1 {name:12})
WITH [r in relationships(p1) | r.name] as r1
MATCH p2 = (s2 {name:1}) -[:Edge*]- (t2 {name:12})
WHERE none(r in relationships(p2) WHERE r.name in r1)
RETURN r1, [r in relationships(p2) | r.name] as r2
```

Cette requête retourne un résultat si et seulement s'il existe 2 chemins différents de $s1$ à $t1$ n'ayant aucune arrête en commun. Elle permet donc de décider *2 edge disjoint path* sur G_{name} (il suffit juste de faire varier la valeur des propriété de noms sur les noeuds $s1$ et $s2$). L'adapter au cas k chemins est trivial. (Cette adaptation n'est d'ailleurs même pas nécessaire, car il est connu que sur les graphes dirigés, *2 edge disjoint path* est NP-complet).