Diversified Top-k Answering of Cypher Queries over Large Data Graphs

Houari Mahfoud
Abou-Bekr Belkaid University & LRIT Laboratory
Tlemcen, Algeria
houari.mahfoud@{univ-tlemcen.dz, gmail.com}

***Abstract*—Cypher is the most used language for querying data modeled as graphs. It is originally designed and implemented as part of the Neo4j graph database system, but it is currently used by other systems. Answering a Cypher query C over a data graph G consists in finding all subgraphs of G that are *isomorphic* to the structure of C and satisfy all its conditions. Despite the NP-Completeness of the *subgraph isomorphism* problem, Neo4j proposes several techniques for efficient answering of C over G. These techniques remain insufficient for the following raisons: 1) real-life data graphs are very large which increases dramatically the answering time; 2) there may exist an exponential number of matches, even though C is very simple, which makes inspection very difficult. On the other hand, users are often interested in only k *relevant* matches which are as *diverse* as possible. Existing solutions rely all on a formal class of queries which does not cover all features used in practice. Moreover, it is hard to see how they can be integrated within a commercial system like Neo4j. This paper proposes practical solutions for the *diversified top-k answering* of Cypher queries. We first investigate a solution that is based on *level-wise* strategy and aims to enhance the quality of the diversified k matches. As it examines the entire match set, this solution may be time-consuming in large data graphs. Indeed, we propose a second solution that rectifies the limit of the first one by applying the *early-termination* property. This solution is a trade-off between quality and efficiency as it allows to find high quality diversified k matches in a reasonable time. We show effectiveness and efficiency of our solutions using real-life and synthetic data. To our knowledge, this paper presents the first solutions for the *diversified top-k answering* of Cypher queries, which can be easily integrated within Neo4j.**

***Index Terms*—Neo4j, Cypher, Pattern Matching, Top-k, Diversity.**

I. Introduction

Over the past decade, we have witnessed widespread use of the Neo4j[1] graph database system in industry and academia. It has been used in multiple domains, such as knowledge management, recommendation engines, social networks analysis, fraud detection, data privacy, access control, and artificial intelligence. The most important module of Neo4j is the Cypher query language [8] which allows querying of property graph databases via *isomorphism semantics*. That is, given a Cypher query C and a property graph G, it is to find all subgraphs of G that are isomorphic to the structure of C and fulfill all constraints of this latter (e.g. node/edge attributes constraints). Neo4j provides many solutions to efficiently compute the match result of C over G (denoted by $\mathcal{M}(C, G)$) such as: parallel/distributed evaluation, data/query cache, data indexation, query planning.

[1]https://neo4j.com/.

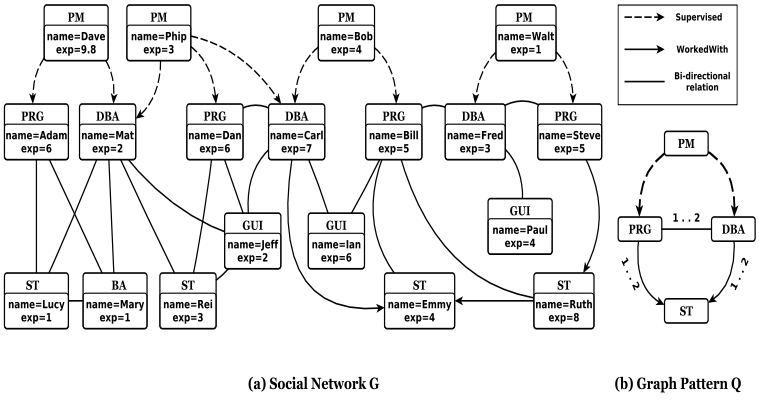


Fig. 1: Querying professional network.

However, the sheer size of social graphs makes these solutions insufficient. Indeed, with real-life graphs having millions of nodes and billions of edges, Neo4j takes a lot of time to compute and return $\mathcal{M}(C, G)$ to the users. Moreover, $\mathcal{M}(C, G)$ may contain exponentially many single results (called *match records*) with someones repeated several times. In addition to the decreased quality of $\mathcal{M}(C, G)$, it is a daunting task for the users to analyse it and find what they are looking for. The next example practically shows these limits and suggests a solution.

Example 1. An example of a collaboration network is depicted as graph G in Figure 1 (a). Each node in G represents a person and it is labeled with the professional profile of this later such as: *project manager* (PM), *database administrator* (DBA), *programmer* (PRG), *business analyst* (BA), *user interface developer* (GUI) and *software tester* (ST). Nodes have attributes specifying *name* and *experience* (i.e. number of years) of the person. Each edge indicates a professional relationship, e.g., (Dave,Adam) indicates that Dave has supervised Adam within a previous project; while (Adam,Mary) indicates that Mary has collaborated well with Adam in some tasks led by Adam.

Suppose that a company wants to build up a team for software development. The requirements are expressed by a graph pattern Q (depicted in Figure 1 (b)) as follows: (1) with exactly four persons whose profiles are PM, PRG, DBA and ST; (2) the PM must have supervisory experience with the PRG and the DBA; (3) PRG and DBA have mutually collaborated with each other either directly or via another person; (4) the ST has

collaborated well with both PRG and DBA via a collaboration path no longer than 2. One can express Q with a Cypher query C defined as follows:

```
Match (n2:PRG)<-[:Supervised]-(n1:PM),
   (n1)-[:Supervised]->(n3:DBA),
   (n2)-[:WorkedWith*1..2]->(n3),
   (n3)-[:WorkedWith*1..2]->(n2),
   (n2)-[:WorkedWith*1..2]->(n4:ST),
   (n3)-[:WorkedWith*1..2]->(n4)
Return *
```

Due to the isomorphism semantics, Neo4j returns a match result $\mathcal{M}(C, G)$ that contains 15 match records, while there are only 6 distinct match records as follows:

Match records of (PM,PRG,DBA,ST) resp.	Repetition	Average experience
t_1 = (Dave,Adam,Mat,Lucy)	3	4.7
t_2 = (Phip,Dan,Carl,Rei)	3	4.75
t_3 = (Phip,Dan,Mat,Rei)	3	3.5
t_4 = (Bob,Bill,Carl,Emmy)	2	5
t_5 = (Walt,Steve,Fred,Ruth)	2	4.25
t_6 = (Walt,Steve,Fred,Emmy)	2	3.25

Repetition is due to the fact that the same match record can be computed in different manners (i.e. by traversing different paths). For instance, the relationship WorkedWith between Bill and Emmy is satisfied both directly and indirectly (via Ruth). That is why the match record t_4 is returned twice. In practice however, users are often interested in some k matches that are relevant and as diverse as possible. Indeed, the problem may consist in computing the *top-k diversified matches* of C over G. To this end, one can think about using the Limit and Order By clauses. Precisely, by fixing the team experience as *relevance criteria*, another version of C, $C^{'}$, may be investigated:

```
Match (n2:PRG)<-[:Supervised]-(n1:PM),
   (n1)-[:Supervised]->(n3:DBA),
   (n2)-[:WorkedWith*1..2]->(n3),
   (n3)-[:WorkedWith*1..2]->(n2),
   (n2)-[:WorkedWith*1..2]->(n4:ST),
   (n3)-[:WorkedWith*1..2]->(n4)
Return *
   Order By (n1.exp+n2.exp+n3.exp+n4.exp)/4 desc
   Limit k
```

$C^{'}$ looks for only k teams with the greatest average experience among their respective members. With $k = 3$, $\mathcal{M}(C^{'}, G)$ is given by $\{t_4, t_4, t_2\}$. The problem with this result is twofold: 1) the match record t_4 is repeated; and 2) t_4 and t_2 share some node between them. This makes the match result less diverse and then less interesting. The *top*-3 *diversified* match result of $C^{'}$ can be $\{t_4, t_1, t_5\}$ where the three teams are quite *dissimilar* as they do not have members in common. After choosing t_4, remark that t_2 is more relevant than t_1, however it shares a node with t_4. This tells that one would like to scarify relevance in order to diversify the match result. □

This example shows that the *top-k diversified answering* of Cypher queries allows users to get the most interesting match records without (computing and) inspecting the entire match result. However, neither Neo4j nor Cypher provide a practical solution for this problem.

Contributions. We make the following contributions:

1) Contrary to existing approaches that rely on formal patterns, we consider queries of the most used graph query language, the Cypher language (Section II-B), to make practical use of our results.
2) We revisit the diversified top-k answering problem by defining the *relevance criteria* as part of the Cypher query which gives more flexibility to the user (Section IV).
3) We prove that the problem is NP-Hard (Theorem 1).
4) Despite hardness of the problem, we propose two approximation algorithms: a) the first one increases the *quality* of the match result by (possibly) examining the entire match set (Section IV-B); while b) the second one finds the match result *efficiently* by applying the *level-wise* and *early-termination* principles (Section IV-C).
5) Based on real-life and synthetic data (Section V), we show that: (a) our first algorithm considerably increases quality of the match result but it may take a lot of time over large data graphs; (b) our second algorithm finds a trade-off between efficiency and quality as it provides high quality match result in a reasonable time over large data graph; (c) our approach requires less than $0.1s$ to identify top-k match records for a complex Cypher query over a data graph with 1 million nodes and 1.5 million edges.

Our results yield a promising approach to querying big data graphs via Cypher or any similar graph query language.

Related Work. We categorize previous work as follows:
Graph Pattern Matching (GPM). Given a data graph G and a user query Q expressed as graph, the GPM problem is typically defined in terms of *subgraph isomorphism* which aims to find all subgraphs of G that are isomorphic to the structure of Q. The approaches following this semantics (e.g. [1], [7], [12], [19]) conduct to exact solutions, but they become intractable in some cases since the subgraph isomorphism is an NP-Complete problem. Some works [3], [4], [11], [13]–[17] studied the GPM under *graph simulation* semantics which aims to conduct the GPM in polynomial time by loosing the structure of the query. Commercial products use only the isomorphism semantics. For instance, the *Neo4j* graph database system applies the isomorphism semantics only over edges of the Cypher queries, which it is called *edge-isomorphism* semantics. As the paper proposes solutions for Neo4j, thus we consider the *edge-isomorphism* semantics.
(Diversified) top-k GPM. The top-k GPM aims to find only the best k matches of Q in G, while the *diversified* aspect forces these k matches to be as diverse as possible. Some works have studied the problem without diversity concern: e.g. [5], [16] for simulation patterns, and [2] for isomorphism patterns. They differ in the manner the match relevance is defined: [5] ranks matches w.r.t number of their nodes, [16] uses a ranking function defined by the user, and [2] ranks matches w.r.t their compactness. The problem becomes more intriguing in case of diversify. A diversification function has been proposed [10], [18] as a bi-criteria optimization problem for balancing result *relevance* and *diversity*. Following the idea, diversified top-k GPM has been studied in, e.g., [5] for

simulation patterns, and [20] for isomorphism patterns. The problem is NP-Complete in [5] and NP-Hard in [20], and thus only approximate solutions are proposed. Close to our work, [20] uses isomorphism patterns where a bound is attached to each edge to specify its length (i.e. maximum length of the path mapping it). The matches are ranked according to their tightness, i.e., a match is relevant if paths mapping its edges have smaller lengths. The diversified top-k matches are found via an approximate solution that applies a *level-wise* strategy and has an *early-termination* property. Several limits are encountered with this solution: (1) The relevance criteria has no sense if all pattern edges have length equal to 1. (2) The solution picks one pattern node (the most relevant one), examines all its candidates on the data graph, and checks whether a new relevant match can be discovered based on each candidate. This may be time-consuming as in real-life graphs, there may be millions of candidates for a given pattern node.

Our work differs from prior work in the following: (1) rather to use formal queries, we consider a subclass of a real-life graph query language (i.e. Cypher) which makes our solution practical and easy to integrate in any graph system like Neo4j. (2) our relevance function is defined by the user as part of his/her Cypher query, which gives more flexibility according to user/application requirements. (3) previous works propose standalone-solutions only, while our solution involves the underlying graph system and leverages its optimization techniques, which makes the solution more efficient.

II. Preliminaries

We give definitions of data graphs and Cypher queries.

A. Data Graphs

A ***data graph***, called also *property graph*, is a directed multi graph G=$(V, E, \mathcal{L}, \mathcal{A})$ where: 1) V is a finite set of nodes; 2) $E \subseteq V \times V$ is a finite set of edges in which $(v, v^{'})$ denotes an edge from node v to $v^{'}$; 3) $\mathcal{L}$ is a function that assigns a label $\mathcal{L}(v)$ (resp. $\mathcal{L}(e)$) to each node $v \in V$ (resp. edge $e \in E$); and 4) for each node $v \in V$ (resp. edge $e \in E$), $\mathcal{A}(v)$ (resp. $\mathcal{A}(e)$) is a tuple $(A_1 = c_1, \ldots, A_n = c_n)$ where: A_i is an attribute of v (resp. e), c_i is a value, $n \geq 0$, and $A_i \neq A_j$ if $i \neq j$.

Intuitively, the label of a node represents an entity (e.g. Person) while the label of an edge represents a relationship (e.g. Supervised, WorkedWith). Moreover, the function $\mathcal{A}$ defines properties over nodes and edges (e.g. attributes name and exp for persons, and attribute date for the relationship WorkedWith). Notice that there may be several edges between the same pair of nodes, which express different relationships.

A ***subgraph*** G_s=$(V_s, E_s, \mathcal{L}_s, \mathcal{A}_s)$ of a data graph G=$(V, E, \mathcal{L}, \mathcal{A})$ must satisfy: 1) $V_s \subseteq V$; 2) $E_s \subseteq E$; and 3) for each $x \in E_s \cup V_s$: $\mathcal{L}_s(x) = \mathcal{L}(x)$ and $\mathcal{A}_s(x) = \mathcal{A}(x)$.

A ***path*** p in some data graph G is a sequence $v_1, \cdots, v_n$ of nodes of G where there is an edge in G for each pair (v_i, v_{i+1}) $(i \in [1, n-1])$. We say that p is *directed* if all these edges have the same direction, and *undirected* otherwise.

B. Cypher Queries

For the sake of readability only, this paper considers a simple but useful class of Cypher queries that are largely used in practice. Our Cypher queries are composed by a Match statement, an optional Where statement, and a simplified Return statement with Limit and Order By clauses. Formally, a Cypher query C has the following syntax:

C ::= $\mathcal{M}$ $\mathcal{W}$ $\mathcal{R}$
$\mathcal{M}$ ::= Match α
α ::= α' | α', α
α' ::= n | $n - r \rightarrow \alpha'$ | $n \leftarrow r - \alpha'$ | $n - r - \alpha'$
n ::= (v) | (v:l) | (v:l{δ})
r ::= $[V\ L\ H\ A]$
V ::= ϵ | v
L ::= ϵ | :l
H ::= ϵ | *1..h | *
A ::= ϵ | {δ}
δ ::= a:val | δ, δ
$\mathcal{W}$ ::= ϵ | Where β
β ::= β and β | β or β | not(β) | (v.a op val) | v <> v′ | Exists{β'}
β' ::= $n - r \rightarrow \alpha'$ | $n \leftarrow r - \alpha'$ | $n - r - \alpha'$
$\mathcal{R}$ ::= Return * $\mathcal{O}$ $\mathcal{L}$
$\mathcal{O}$ ::= Order By $\mathcal{E}xp$ | Order By $\mathcal{E}xp$ desc
$\mathcal{E}xp$ ::= v.a | ($\mathcal{E}xp$) | $\mathcal{E}xp$ op′ $\mathcal{E}xp$ | $\mathcal{E}xp$ op′ val | fct(.)
$\mathcal{L}$ ::= Limit k

Where v (resp. v′) is a node/relationship variable, l is a node/relationship label, a is an attribute name, op $\in \{=, <>, <, >, \hat{}\}$[2] is a boolean operator, op′ $\in \{+, -, *, \div\}$ is an arithmetic operator, val is a value, k and h are integers. Notice that fct is any predefined Cypher function which can take one/many inputs such as: a) atomic values (e.g. sqrt(v1.exp+v2.exp)); b) a path pattern (e.g. shortestpath($(v1) - [*] \rightarrow (v2)$)); or c) nested functions (e.g. length(shortestpath($(v1) - [*] \rightarrow (v2)$))).

In a nutshell, the Match statement specifies the patterns we are looking for on the data graph, the Where statement defines conditions over these patterns, and the Return statement specifies the manner these patterns will be returned to the user. Precisely, the Match statement allows to define a sequence of patterns separated by ",". Each pattern (formalized by α') can be either a simple node or a concatenation of nodes and relationships. In Cypher language, (...) refers to a node while [...] refers to a relationship between two nodes. Each node is defined with a variable v, an optional label l, and an optional list $\{\delta\}$ of attribute conditions (e.g. {*age:'35',gender:'female'*}). The variable is used to refer to the node throughout the query. A relationship can be directed (i.e. (.)-[.]→(.) or (.)←[.]-(.)) or undirected (i.e. (.)-[.]-(.)). It can be defined shortly by [ ] with refers to a relationship with no constraints, while optional constraints can be attached to it such as: i) a variable v; ii) a label l; iii) a length specified either with *1..h (*bounded length*) or * (*unbounded length*); and iv) a list of attribute conditions. A relationship may specify: a direct edge between two nodes (e.g. $(v1{:}Person) - [r : Friend] \rightarrow (v2{:}Person)$); a path of bounded

[2]The symbol <> expresses inequality, while ^ expresses power.

length (e.g. (v1:Person)$-[r:\text{Friend}*1..2]\rightarrow$(v2:Person)); or a path of unbounded length (e.g. (v1:Person)$-[r:\text{Friend}*]\rightarrow$(v2:Person)). A relationship is called *simple* if its length is equal to 1, and it is called a *variable-length relationship* otherwise.

The Where statement allows either to define attribute conditions over the nodes/relationships introduced by the Match statement, or to check the existence of some patterns between these nodes (via the function Exists).

We call *match record* any subgraph of the data graph that satisfies the Match and Where requirements. By using "Return *", a *match record* will map any node (resp. relationship) variable of the Match statement into a node (resp. edge/path) in the data graph. However, other versions of Return are allowed by the Cypher language. For instance, with "Return v1, v2", each match record will give mappings only for the two mentioned variables. Remark that the Return statement allows to return only k match records, and moreover, to sort them (by default in ascending order) via the Order By clause. The optional *desc* allows to sort match records in descending order. The sorting is done based on an expression $\mathcal{E}$ that can involve arithmetic operations over node/relationship attributes and/or call some predefined functions over them.

The *match result* of C over some data graph G (i.e. $\mathcal{M}(C,G)$) is a set containing all *match records* of C.

Example 2. Consider the following Cypher query $C^{''}$:

```
Match (n2:PRG)<-[:Supervised]-(n1:PM),
   (n1)-[:Supervised]->(n3:DBA)
Where (n1.exp > 2) and
   Exists{(n2)-[]->(n4:ST{exp:1})<-[]-(n3)}
Return * Order By n1.exp limit 3
```

$C^{''}$ looks for any PM who has supervised a PRG and a DBA. The Where statement specifies that the PM must have more than 2 years experience, and in addition, the PRG and the DBA must have a direct relationship (of any type) with some ST who has/have 1 year experience. $C^{''}$ returns mappings of only variables n1, n2 and n3 as they make part of the Match statement, however, variable n4 is within the Where statement and then it does not make part of any match record. Finally, $C^{'}$ returns up to 3 match records sorted according to the experience of the PM. Consider the data graph G of Figure 1 (a), $\mathcal{M}(C^{''},G)$ has only one match record {Dave,Adam,Mat} specifying matches of n1, n2 and n3 respectively. □

III. The Diversified Top-k Answering Problem

As early mentioned, the users are often interested, not in the whole match result which can be excessively large, but in the best k matches that are as diverse as possible. We first show how to measure: a) the *relevance* of a match; and b) the *distance* between two matches which consists in determining their diversity. Based on these two notions, we give the definition of the *diversification* function which is a bi-criteria objective function combining both *relevance* and *distance*. Finally, we recall the definition of the *diversified top-k answering* problem as well as its complexity. Notice that, rather to use a static relevance function [5], [6], [20], our work allows the user to define his/her preferred relevance criterias as part of the query.

A. Relevance Criteria (Revised)

The relevance functions defined in previous works are user-independent and are not suitable for real-life queries. For instance, the relevance function defined in [5], [6] favors those matches that have more nodes. However, with subgraph isomorphism semantics, all matches have the same number of nodes which makes impossible their ranking. Moreover, authors of [20] designed a relevance function especially for bounded edges (i.e. variable-length relationships) in the sense that it favors those matches whose nodes are connected tightly. It is clear that, for queries with no variable-length relationships, all matches will have the same tightness degree.

To overcome these limits, we allow the users to define their relevance criteria via the Order By clause. This will make our solution suitable for any kind of query and situation. Notice that the ranking value of any match record must belong to $[0,1]$. Given a data graph G and a Cypher query C with "Order By $\mathcal{E}$". For any match record $t \in \mathcal{M}(C,G)$, we denote by $\mathcal{E}(t)$ the value obtained by executing the expression $\mathcal{E}$ over t. Moreover, $max(\mathcal{E}(.))$ refers to the maximum value obtained by all possible matches of C in G. Given the above, we define our relevance function $\mathcal{R}(.)$ as follows:

$$\mathcal{R}(t)_{t\in\mathcal{M}(C,G)} = \begin{cases} \frac{\mathcal{E}(t)}{max(\mathcal{E}(.))}, & \text{in case of } \textit{descending} \text{ order} \\ 1-\frac{\mathcal{E}(t)}{max(\mathcal{E}(.))}, & \text{in case of } \textit{ascending} \text{ order} \end{cases}$$

That is, matches with high $\mathcal{R}(.)$ are preferred for relevance.

Example 3. Consider the Cypher query C of Example 1 and its match records $\{t_1,...,t_6\}$. The *Order By* expression $\mathcal{E}$ computes the average experience of each team members, e.g. $\mathcal{E}(t_1)=4.7$, and the maximum value $max(\mathcal{E}(.))$ is given by 5. That is, the relevance values are given by: $\mathcal{R}(t_1)$=4.7/5=0.94, $\mathcal{R}(t_2)$=0.95, $\mathcal{R}(t_3)$=0.7, $\mathcal{R}(t_4)$=1, $\mathcal{R}(t_5)$=0.85 and $\mathcal{R}(t_6)$=0.65. □

B. Distance Criteria

The *dissimilarity* of matches is measured based on a *distance* function. Given two match records t_1 and t_2 with node sets V_1 and V_2 respectively. The *distance* between t_1 and t_2, denoted by $\mathcal{D}(t_1,t_2)$, is defined as follows:

$$\mathcal{D}(t_1,t_2) = 1 - \frac{|V_1 \cap V_2|}{|V_1 \cup V_2|}.$$

I.e, the larger $\mathcal{D}(t_1,t_2)$ is, the more dissimilar t_1 and t_2 are.

Example 4. We give some distances based on Example 3: $\mathcal{D}(t_4,t_2)=\frac{6}{7}$, $\mathcal{D}(t_4,t_1)=\mathcal{D}(t_4,t_5)=\mathcal{D}(t_5,t_1)$=1. Thus, the match result $\{t_4,t_1,t_5\}$ contains 03 dissimilar teams. □

C. Match Result Diversification

Consider a match record set $S=\{t_1,\cdots,t_k\}$ of a Cypher query C on a data graph G. The diversification function $\mathcal{F}(.)$ is defined as follows:

$$\mathcal{F}(S) = (1-\lambda)\sum_{t_i\in S}\mathcal{R}(t_i) + \frac{2.\lambda}{k-1}\sum_{t_i\in S, t_j\in S, i<j}\mathcal{D}(t_i,t_j)$$

where $\lambda \in [0,1]$ is a parameter set by users to determine the preference for diversifying match records based on specific

criteria. In other words, when λ is closer to 0 (resp. 1), it indicates a higher preference for *relevance* (resp. *distance*). The diversity metric is scaled down with $\frac{2.\lambda}{k-1}$, since there are $\frac{k.(k-1)}{2}$ numbers for the difference sum, while only k numbers for the relevance sum. The function $\mathcal{F}$ has been introduced first in [10] and slightly revised later in [5]. It is a bi-criteria objective function to capture both relevance and diversity. It strikes a balance between the two with a parameter λ that is controlled by users, as a trade-off between the two [18].

D. Problem Statement

Based on the diversification function $\mathcal{F}(.)$, the *diversified top-k answering* problem, denoted by DivTopK, is stated as follows. Given a Cypher query C, a data graph G, an integer k, and a parameter $\lambda \in [0,1]$, it is to find a set of k matches $S \subseteq \mathcal{M}(C,G)$ such that: $\mathcal{F}(S) = argmax_{S' \subseteq \mathcal{M}(C,G)} \mathcal{F}(S')$. That is, there is no $S' \subseteq \mathcal{M}(C,G)$ such that $\mathcal{F}(S') > \mathcal{F}(S)$.

Example 5. Recall the Cypher query C and data graph G of Example 1. Fixing $\lambda = 0.5$, one can verify that when $k = 2$ (resp. $k = 3$), $\{t_4, t_1\}$ (resp. $\{t_4, t_1, t_5\}$) makes a top-2 (resp. top-3) diversified match result since its diversification value 1.97 (resp. 2.89) is maximum among all 2-element (resp. 3-element) subsets of $\mathcal{M}(C,G)$. □

Theorem 1. The DivTopK problem is NP-HARD (*decision problem*). □

Proof Sketch. The decision problem of DivTopK is stated as follows. Given C, G, k, $\lambda \in [0,1]$, and a bound B, it is to decide whether there exists a k-element subset $S \subseteq \mathcal{M}(C,G)$ such that $\mathcal{F}(S) \geq B$.

Authors of [20] have studied the same decision problem for graph patterns formalized as an undirected graph $P = (N_p, E_p, f_n, f_e)$ where: N_p is the set of pattern nodes; E_p is the set of undirected pattern edges; f_n is a function giving a label to each node in N_p; and f_e is a function assigning a positive integer $h \geq 1$ or * to edges in E_p. For any edge $e \in E_p$, $f_e(e) = h$ (resp. *) specifies that e can be mapped by a data path with length up to h (resp. with unbounded length). They showed that the DivTopK problem is NP-HARD for these graph patterns by reduction from the NP-COMPLETE 3-*dimensional matching problem* [9].

Notice that one can transform any graph pattern P into a Cypher query C. This transformation must follows two stages: *syntax* and *semantics* transformation.

Syntaxical transformation. The syntax of P can be represented via the Cypher query C as follows:

- Create a statement Match$(v_1:l_1), \cdots, (v_k:l_k)$ where: each $n_{1 \leq i \leq k} \in N_p$ is represented via a variable v_i and $f_n(n_i) = l_i$.
- Create a statement Where c_{e_1} **and** $\cdots$ **and** c_{e_m} where: for $e_{1 \leq i \leq m}$ in E_p, c_{e_i} is a boolean formula checking the existing of this edge via Cypher syntax. That is, let $e_i = (n_1, n_2)$ and $f_e(e_i) = h$, then c_{e_i} is given by: Exists$\{(v_1) - [*1..h] - (v_2)\}$. Moreover, if $f_e(e_i) = *$ then *1..h is replaced by * to express an unbounded length relationship.
- Create a statement: Return * Order By $\mathcal{E}$ desc.

The expression $\mathcal{E}$ must be defined related to the semantics of relevance used for the graph pattern P. Recall that, for any match t of P in some data graph G, its relevance value (denoted by $w(t)$) is defined in [20] with:

$$w(t) := \frac{\sqrt{|E_p|}}{\sqrt{(\sum_{e=(n_1,n_2) \in E_p} dist(t(n_1), t(n_2))^2}}$$

Where $t(n_1)$ (resp. $t(n_2)$) is the match given for the pattern node n_1 (resp. n_2) in G by t, and $dist(t(n_1), t(n_2))$ is the length of the shortest path connecting the data nodes $t(n_1)$ and $t(n_2)$. Therefore, to complete the above transformation from P into C, one can easily define $w(t)$ via a Cypher expression $\mathcal{E}$ by using the predefined function sqrt and some operators (e.g. / for division and ^ for power). Precisely, $\mathcal{E}$ is given by:

$$\mathcal{E} := \text{sqrt(length\_Of\_}E_p\text{)} \ / \ \text{sqrt}(\textstyle\sum_{e \in E_p} d_e \ \hat{} \ 2\ )$$

Where for each edge $e = (n_1, n_2)$, d_e is given in Cypher by:

$$d_e := \text{length(shortestpath}((v_1) - [*] - (v_2)))$$

Semantical transformation. Authors of [20] studied the DivTopK problem under full subgraph-isomorphism semantics which means that each node (resp. edge) of P must be mapped to a different node (resp. edge) in the data graph. However, Cypher ensures isomorphism only over edges (i.e. edge isomorphism) which means that two nodes of C may be mapped to the same data node. To enforce the full isomorphism semantics of P via C, the Where statement created above must be extended with some conditions as follows. For any two nodes n_1 and n_2 in N_p, we add $v_1 <> v_2$ to the conjunctionnal conditions of the above Where statement. In this way, C will force the application of the isomorphism semantics between nodes of P, while it applies it by default over edges.

Summary. Given the above, the transformation from P to C is in PTIME, and it is indeed a reduction since there exists a solution for C (i.e. a k-element subset S with $\mathcal{F}(S) \geq B$) if and only if there exists a solution for P. As authors of [20] showed that the decision problem of DivTopK is NP-HARD for graph patterns, hence it is also NP-HARD for Cypher queries as graph patterns are special case of our Cypher queries.

Notice that authors of [5] showed that the DivTopK problem is NP-COMPLETE for graph patterns under graph simulation. The NP-COMPLETENESS is not ensured under isomorphism semantics since: given a k-element subset S as a certificate, checking whether $S \subseteq \mathcal{M}(C,G)$ cannot be done in PTIME.

IV. ALGORITHMS FOR DIVERSIFIED TOP-K ANSWERING

We first describe an approximation algorithm for the DivTopK problem [20] that considers graph patterns. Next, we introduce two approximation algorithms for Cypher queries.

A. A Conventional Algorithm

Authors of [20] proposed an approximation algorithm for the DivTopK problem which is based on: the *level-wise* strategy and the *early-termination* property. The algorithm, referred to

Algorithm SL(G, C, k)
Input: A data graph G, a Cypher query C with a node set V_C, and an integer k.
Output: A set S of diversified top-k match records of C in G.

1: Initialize $S := \emptyset$, $l := 0$, $sk := 0$, $b := 2 * k$;
2: **while** $l \leq |V_C|$ **do**
3: Generate $C^{'}$ by replacing "Limit k" in C with "Skip sk Limit b";
4: Compute $\mathcal{M}(C^{'}, G)$;
5: **if** $(\mathcal{M}(C^{'}, G) = \emptyset$ **and** $sk = 0)$ **then** /* *Case of empty match result* */
6: **return** $\emptyset$;
7: **else if** $(\mathcal{M}(C^{'}, G) = \emptyset$ **and** $sk \neq 0)$ **then** /* *Case of empty batch result* */
8: $l := l + 1$; $sk := 0$;
9: **continue**;
10: **for each** (match record $t \in \mathcal{M}(C^{'}, G)$ with $t \notin S$) **do**
11: **if** (t has at most l common nodes with S) **then**
12: $S := S \cup \{t\}$;
13: **if** $|S| = k$ **then return** S;
14: $sk := sk + b$;
15: **return** S;

Fig. 2: A brute-force algorithm for the DivTopK.

as TopkET, searches diversified match records *level by level* and terminates at certain *level* once k match records are found.

Given a data graph G, a query Q and a value k, TopkET starts by initializing the level $l = 0$ and the set S of match records to empty. Next, it applies a *level-wise* strategy which consists to look first for diversified match records that share no node (i.e. $l = 0$) between them. If k match records are found, then the algorithm stops, otherwise, it repeats the same process with next level, and so on until the maximum level (i.e. $|V_Q|$) is reached. At each level, TopkET takes a query node u from Q and computes the set $cand(u)$ of its candidates in G (i.e. nodes in G having the same label of u). For each $v \in cand(u)$, TopkET finds incrementally all match records of Q in G that map u to v (i.e. a set S_u). Next, only one match record $t \in S_u$ is chosen and added to S such that: t is the most relevant one in S_u that shares exactly l node(s) with S. Once $|S| = k$, TopkET stops running by returning S. If all candidates of u are examined and S still contains less than k match records, then the whole process is repeated with level $l + 1$. An optimized version of this algorithm, referred to as TopkET$_{opt}$, consists in selecting the query node u that has less candidates in G. This helps decreasing significantly the answering time of TopkET as showed the experiments of [20].

When thoroughly analysing algorithm TopkET$_{opt}$, we have formally and experimentally discovered different drawbacks. The main limitation is related to efficiency. The candidates enumeration process makes TopkET$_{opt}$ very sensitive to large data graphs as there may be an important number of candidates for each query node. For instance, some real-life data graphs like *Amazon*, *YouTube*, and *Facebook* contain millions of nodes which increases the answering time of TopkET$_{opt}$, and especially, when different levels are considered. Furthermore, applying the algorithm for queries with complex features like aggregation functions (e.g. max, count) or unique nodes (e.g. a query with a condition Where v.name='*Adam*') may lead to incorrect/incomplete match result. We do not give details about these limitations due to space limit.

B. Our Brute-force Algorithm

Given a Cypher query C and a data graph G. One can think to approximately resolve the DivTopK problem in two steps as follows: 1) compute the match result $\mathcal{M}(C, G)$; and 2) extract approximately a k-element subset S of $\mathcal{M}(C, G)$ that contains the most relevant and diversified match records of $\mathcal{M}(C, G)$. This strategy may be time-consuming since, in case of large data graphs, there may be millions of match records even for a simple Cypher query. To avoid this limit, one can think to consider, not the whole set $\mathcal{M}(C, G)$, but only a subset of it having a limited size, and to proceed to step (2) with it. If S is not completely identified (i.e. $|S| < k$) then repeat this incremental process with another subset of $\mathcal{M}(C, G)$, otherwise, return S as a final result. We propose an approximation algorithm that follows this idea and implements a *level-wise* strategy.

Our algorithm, referred to as SL, is given in Figure 2. It takes a data graph G, a Cypher query Q, an integer k, and works as follows. Starting at level $l = 0$, it defines a new version of C, $C^{'}$, by replacing the Limit k clause with "Skip sk Limit b". This means that only b match records will be returned by $C^{'}$ by ignoring the first sk records. In other words, rather to compute the whole set $\mathcal{M}(C, G)$, only a subset of size b is considered, called a *batch result*. At the beginning, sk is given by 0 while the size of the batch result, b, is given by $2 * k$ (line 1). Notice that b can be initialized with any multiple of k. We have experimentally remarked that a high value of b (e.g. $5 * k$) allows Neo4j to execute less queries, which reduces the answering time of our algorithm. Once computing the batch result $\mathcal{M}(C^{'}, G)$ (line 4), three cases can be considered. 1) If $|\mathcal{M}(C^{'}, G)| = 0$ and $sk = 0$ (lines 5–6), then obviously the original query C has no match record and the algorithm stops running by returning $\emptyset$. 2) If $|\mathcal{M}(C^{'}, G)| = 0$ but $sk \neq 0$ (lines 7–9), then this means that all possible batch results have been analyzed. In this case, SL repeats the whole process with another level, i.e. $l = l + 1$ and $sk = 0$. 3) If $|\mathcal{M}(C^{'}, G)| \neq 0$, then the current batch result is examined (lines 10–14) by extracting all its match records that intersect with S in at most l nodes (0 nodes at the beginning). Match records that fulfill this condition are added to S (lines 11–12). The set S is returned once its size is equal to k (line 13).

If the current batch result is entirely examined and S still has less than k match records, then SL generates another query by adding b to the value of sk (line 14). This means that the same process (3–14) will be repeated for another batch result, i.e. the new value of sk aims to discard the match records that have been previously analysed. At the worse case, SL could consider all possible levels (i.e. $|V_C|$), and for each one, it could consider all possible batch results. Finally, it will return S as the diversified top-k match records of C.

Example 6. Consider $C^{'}$ of Example 1. With $k = 4$, SL initializes $S = \emptyset$, $l = 0$, and extracts the first batch result that is sorted based on the requirement of $C^{'}$ as follows: $\{t_4, t_2, t_1, t_5, t_3, t_6\}$. Next: t_4 intersects with S in 0 nodes and then it is added to S; t_2 is ignored as it has a common node

Algorithm SLET(G, C, k)
Input: A data graph G, a Cypher query C with a node set V_C, and an integer k.
Output: A set S of diversified top-k match records of C in G.

1: Initialize $S := \emptyset$, $l := 0$, $sk := 0$, $b := 2 * k$;
2: Extract from C: Match statement M, Where statement W, Return statement R, and Order By statement O;
3: **while** $l \leq |V_C|$ **do**
4: Let LW be the expression "With $*$ Skip sk Limit b";
5: Create a Cypher query $C^{'}$ by combining the statements: $M\ W\ LW\ R\ O$;
6: Compute $\mathcal{M}(C^{'}, G)$;
7: **if** ($\mathcal{M}(C^{'}, G) = \emptyset$ **and** $sk = 0$) **then** /* Case of empty match result */
8: **return** $\emptyset$;
9: **else if** ($\mathcal{M}(C^{'}, G) = \emptyset$ **and** $sk \neq 0$) **then** /* Case of empty batch result */
10: $l := l + 1$; $sk := 0$;
11: **continue**;
12: **for each** (match record $t \in \mathcal{M}(C^{'}, G)$ with $t \notin S$) **do**
13: **if** (t has at most l common nodes with S) **then**
14: $S := S \cup \{t\}$;
15: **if** $|S| = k$ **then return** S;
16: $sk := sk + b$;
17: **return** S;

Fig. 3: An early-termination algorithm for the DivTopK.

with S (i.e. Carl) while $l = 0$; t_1 and t_5 are added to S; while t_3 and t_6 are ignored. The current batch result is fully examined, there is no other batch result, and $|S| = 3 < 4$, thus, SL moves to next level and repeats the process. At level 1, t_2 intersects with S in only one node (i.e. Carl) and thus it is added to S. At this time, we have $|S| = 4$ and then SL returns $S = \{t_4, t_1, t_5, t_2\}$ as the diversified top-4 match result. □

Notice that the algorithm SL[3] does not preserve the *early-termination* property since, for any Cypher query $C^{'}$ where the Order by $\mathcal{E}$ clause is followed by the Limit k clause, Neo4j will first sort all possible match records of $C^{'}$ in G w.r.t $\mathcal{E}$, and then return the k first ones. However, the above property specifies the ability of an algorithm to respond without computing the entire match result.

C. Our Early-Termination Algorithm

As we should show later by experiments, algorithm SL provides a high quality match result since it first extracts the most relevant match records, across the entire data graph, and then filters them based on diversity. Even by using index of the graph database system, the sorting process may require a considerable time on large data graphs, which may make SL inefficient. To rectify this, we propose a revisited version of SL that preserves the *early-termination* property and guarantees a trade-off between quality and efficiency.

Our second algorithm, referred to as SLET, is given in Figure 3. The main difference between SL and SLET resides on the execution order between Order by and Limit k. When Limit k is placed after Order by (case of algorithm SL), then the system will sort all possible match records and then returns the first k ones. However, when Limit k is placed before Order by (idea of algorithm SLET), then the sorting will be applied only over k match records. Precisely, given a data graph G; an integer k; and a Cypher query C composed by four statements, Match (M), Where (W), Return (R) and Order By (O). Algorithm SLET adds the expression "With $*$ Skip sk Limit b" before the Order By clause of $C^{'}$ (lines 4–5). In this way, the system will look only for b match records and then the sorting is applied over them via O. This makes the sorting process less time-consuming since b is much smaller than the size of the entire match result. Each time a batch result is computed and sorted (lines 4–6), SLET behaves exactly like SL in terms of batch examination (lines 12–15), batch re-computation (line 16), level update (lines 9–11), and emptiness case (lines 7–8).

Therefore, SLET is a *level-wise* strategy (as it is a revisited version of SL) and it preserves the *early-termination* property (as it does not need to analyse the entire match result).

Example 7. We continue with Example 6. With $k = 2$, SLET extracts 4 match records, i.e. $\{t_1, t_2, t_3, t_4\}$; sorts them w.r.t the requirement of $C^{'}$, which gives the batch result $\{t_4, t_2, t_1, t_3\}$ (remark that the sorting is done over only 4 match records and not all records). Next, SLET behaves like SL to examine the above batch result, which yield to $S = \{t_4, t_1\}$ as the diversified top-2 match result. □

Notice that the Where statement of a Cypher query C may require all match records of C to have a common node (e.g. Where v1.id=1). We call this node a *unique node*, and we should show that *unique nodes* force the algorithms SL and SLET to analyse more levels, and thus, to take more time.

V. Experimental Evaluation

We next conduct several experimental sets in order to verify the effectiveness, efficiency and scalability of our algorithms.

A. Experimental Setup

We used the *Amazon* real-life network dataset[4], a product co-purchasing graph with *548K* nodes and *1.78M* edges. In addition, synthetic data graphs are used for scalability checking.

Using Java language, we implemented our algorithms SL and SLET; as well as the conventional algorithms TopkET and TopkET$_{opt}$ [20]. We implemented a generator that produces a synthetic data graph by controlling the number of nodes, edges, labels, and attributes. All the experiments were run on a Ubuntu desktop machine with an Intel Core i5-8250u CPU, 8GB memory, and 256GB SSD storage.

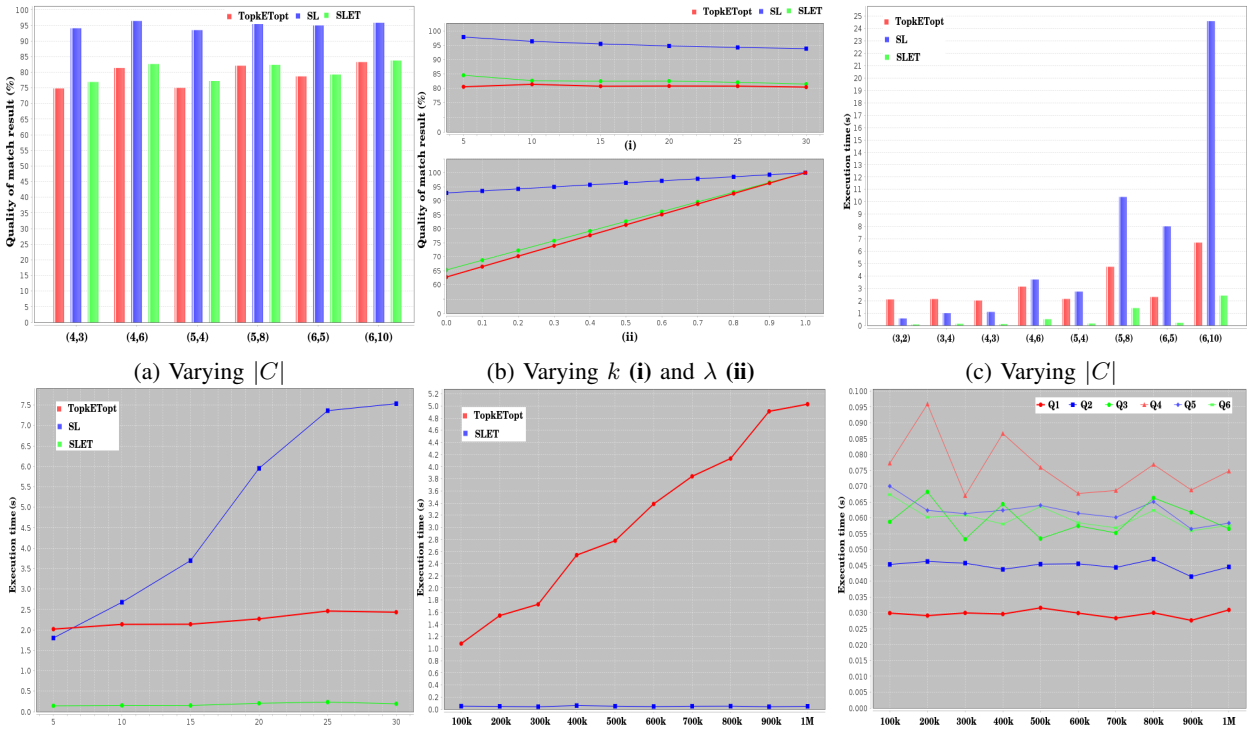
B. Experimental Results

Exp-1: Effectiveness. We measured the effectiveness by computing the quality of the match result returned by each algorithm. Given a k-element match result S, its quality is given by: $(\mathcal{F}(S) * 100)/k$ %.

Varying $|C|$: $|C|$ refers to the number of nodes and relationships of the Cypher query C. We fixed $k = 15$ and $\lambda = 0.5$ and we varied $|C|$ from $(4, 3)$ to $(6, 10)$. The results reported in Figure 4(a) show that SL is more effective than SLET and TopkET$_{opt}$. This is due to the fact that SL applies the Order By defined by the user over all the match records of C and

[3]It gets its name from the use of Skip and Limit clauses.

[4]Available at: http://snap.stanford.edu/data/index.html.



(a) Varying $|C|$ (b) Varying k **(i)** and λ **(ii)** (c) Varying $|C|$

(d) Varying k (e) Varying $|G|$ (f) Varying $|C|$ and $|G|$

Fig. 4: Experimental results.

then examines the sorted match records in order to find the top-k diversified ones. However, SLET extracts repeatedly few match records of C and applies the Order By over them, which may not lead to the most relevant match result. Furthermore, remark that the quality of SLET is slightly better than that of TopkET$_{opt}$. For instance, consider a query with size $(5, 4)$, the qualities ensured by SL, SLET and TopkET$_{opt}$ are 93.5%, 77% and 75% respectively.

Varying k and λ: we fixed $\lambda = 0.5$ and $|C| = (4, 6)$, and we varied k from 5 to 30. The results reported in Figure 4(b−i) are consistent with those of Figure 4(a). That is, SL is the most effective algorithm, followed by SLET and then TopkET$_{opt}$. Next, we fixed $k = 10$ and $|C| = (4, 6)$, and we varied λ from 0.0 to 1.0 in steps of 0.1. The results reported in Figure 4(b−ii) indicate that all algorithms effectively achieve diversification, with their match result quality reaching 100% when the diversification criterion is emphasized (i.e. big value of λ). However, when focusing on the relevance criterion (i.e. less value of λ), SL is still more effective, followed by SLET.

Exp-2: Efficiency. We evaluated the efficiency of the algorithms by comparing their answering times. All experiments were run 5 times and the average times are reported hereafter.

Varying $|C|$: we fixed $k = 10$ and $\lambda = 0.5$, and we varied $|C|$ from $(3, 3)$ to $(6, 10)$. The answering times of the algorithms are given in Figure 4(c). The results show that: 1) SL takes a lot of time to respond in case of large queries due to its brute force strategy; 2) SLET and TopkET$_{opt}$ are more efficient than SL thanks to their early-termination property; 3) for small queries, SL is faster than TopkET$_{opt}$ because the latter involves a candidate enumeration process, however, the time required for this process remains constant for larger queries, while the answering time of SL increases significantly; 4) SLET outperforms significantly TopkET$_{opt}$ by an improvement of 67% to 95% which is due to the fact that SLET computes the query and sends it to Neo4j to be executed, this involves the query planner and indexes of the Neo4j system and allows to find quickly a batch result to examine, however, TopkET$_{opt}$ runs the query on each candidate of some query node and many of these candidates may not produce any match records, which makes TopkET$_{opt}$ very slow compared to SLET.

We conducted the same experiment with different kinds of queries (e.g. with *descending/ascending* order, order defined over one/multiple node attributes, relationships of multiple hopes, Where statements, Exists function, negation) and the results were consistent to those of Figure 4(c). We observed that, contrary to SLET and TopkET$_{opt}$, SL is very sensitive to queries involving multiple hops and aggregations. To be more precise, when we considered a query of size $(6, 10)$ that included a relationship with two hops, the response time of TopkET$_{opt}$ and SLET was 6.87 and 3.42 seconds respectively, which is consistent with the values shown in Figure 4(c). For SL however, the response time increased significantly from 24.5 seconds (with 0 hops) to 90.35 seconds (with 2 hops).

Varying k: we fixed $\lambda = 0.5$ and $|C| = (5, 4)$, and we varied k from 5 to 30. The outcomes presented in Figure 4(d) demonstrate that the performance of SL deteriorates

significantly as k increases, whereas the response times of SLET and TopkET$_{opt}$ increase slowly in relation to the value of k. This is due to the fact that, when LIMIT k is placed after ORDER BY, i.e. case of of SL, Neo4j applies a *partial sort* over the entire data graph, which means it considers all the match records of C but it does not sort them all, it only sorts the necessary records up to the specified value of k. In other words, Neo4j stops sorting once it has found the first ordered k match records of C. This optimization improves query performance, however, in case of large value of k, the sorting process has to examine more records, which increases the answering time of SL as observed above. On the other hand, SLET looks for the first k match records of C and then it sorts them (i.e. LIMIT k is placed before ORDER BY). As a result, increasing the value of k has less impact on the answering time since the sorting process is not applied over the entire data graph.

Exp-3: Scalability. We finally checked the scalability of our algorithm SLET compared to TopkET$_{opt}$. SL is omitted since it takes a lot of time to respond. We fixed $k = 10$, $\lambda = 0.5$, $|C| = (6, 5)$, and we generated 10 synthetic data graphs whose sizes varied from $(100k, 150k)$ to $(1M, 1.5M)$. For each data graph, the answering time of each algorithm is measured. The results, reported in Figure 4(e), tell us that: 1) SLET scales with $|G|$ much better than TopkET$_{opt}$; 2) TopkET$_{opt}$ takes more time which is due to the candidates enumeration process that becomes time-consuming with large data graphs; 3) the time of SLET is a little bit constant and is not sensitive to large data graphs. We repeated this experiment with different kind of queries and the results remained consistent.

The last experiment was dedicated to SLET only. We first defined different kind of queries by varying the number of nodes and edges ($Q1$ and $Q2$), number of variable-length relationships ($Q3$ and $Q4$), and number of unique nodes ($Q5$ and $Q6$). For each query, we computed the answering time of algorithm SLET using the same synthetic data sets of the previous experiment. The answering times, reported in Figure 4(f), tell that: 1) the time of SLET grows by increasing the size of the query (see time of $Q2$ compared to that of $Q1$), its number of variable-length relationships (see $Q4$ compared to $Q3$) or unique nodes; 2) SLET takes more time for queries with variable-length relationships (e.g. see time of $Q4$) since more intermediate nodes and edges are traversed by the query; 3) queries with unique nodes (case of $Q5$ and $Q6$) take more time than simple queries ($Q1$ and $Q2$) as the algorithm goes through different levels to find the diversified top-k results which takes more time; and 4) SLET still scales very well with large data graph and complex Cypher queries.

Summary. (1) SL can find diversified top-k match records with height quality but it does not scale well with large data graphs. (2) SLET ensures an acceptable quality, that is relatively close to that of SL, and scales very well with large data graphs and complex Cypher queries. For instance, by considering a complex Cypher query C with size $(6, 5)$, two unique nodes, and three variable-length relationships having each one 3 hopes, SLET takes $227ms$ to find the diversified top-5 match records of C over a data graph of size $(1M, 1.5M)$.

VI. CONCLUSION

We studied the *diversified top-k answering* problem of Cypher queries. We showed that the problem is NP-Hard and then we proposed approximate solutions: the first one increases the quality of the matches but it demonstrates efficiency only over small data graphs; while the second one has an *early-termination property* and presents a trade-off between quality and efficiency as it finds high quality diversified matches in a reasonable time over possibly large data graphs. Experiments showed that our approach overcomes prior solutions in terms of efficiency and scalability. We are currently exploring parallelism to further improve efficiency of our algorithms. In addition, we are extending our results to deal with distributed data graphs.

REFERENCES

[1] Merve Asiler, Adnan Yazıcı, and Roy George. HyGraph: a subgraph isomorphism algorithm for efficiently querying big graph databases. *Journal of Big Data*, 9(1), 2022.

[2] Jiefeng Cheng, Xianggang Zeng, and Jeffrey Xu Yu. Top-k graph pattern matching over large graphs. In *ICDE*, pages 1033–1044, 2013.

[3] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, pages 39–50, 2011.

[4] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *VLDB Endowment*, pages 264–275, 2010.

[5] Wenfei Fan, Xin Wang, and Yinghui Wu. Diversified top-k graph pattern matching. *Proc. VLDB Endow.*, pages 1510–1521, 2013.

[6] Wenfei Fan, Xin Wang, and Yinghui Wu. Expfinder: Finding experts by graph pattern matching. In *ICDE*, pages 1316–1319, 2013.

[7] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Adding counting quantifiers to graph patterns. In *SIGMOD*, pages 1215–1230, 2016.

[8] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD*, pages 1433–1445. ACM, 2018.

[9] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[10] Sreenivas Gollapudi and Aneesh Sharma. An axiomatic approach for result diversification. In *WWW*, pages 381–390, 2009.

[11] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. Database Syst.*, 39(1):4:1–4:46, 2014.

[12] Tinghuai Ma, Siyang Yu, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. A comparative study of subgraph matching isomorphic methods in social networks. *IEEE Access*, 6:66621–66631, 2018.

[13] Houari Mahfoud. Graph pattern matching preserving label-repetition constraints. In *MEDI*, pages 268–281, 2018.

[14] Houari Mahfoud. Conditional graph pattern matching with a basic static analysis. In *MedPRAI*, pages 298–313, 2020.

[15] Houari Mahfoud. Graph pattern matching with counting quantifiers and label-repetition constraints. *Clust. Comput.*, 23(3):1529–1553, 2020.

[16] Houari Mahfoud. Expressive top-k matching for conditional graph patterns. *Neural Computing and Applications*, pages 1–17, 2021.

[17] Houari Mahfoud. Towards a Strong Containment for Efficient Matching of Expressive Graph Patterns. In *MEDES*, pages 48–55, 2022.

[18] Marcos R. Vieira, Humberto Luiz Razente, Maria Camila Nardini Barioni, Marios Hadjieleftheriou, Divesh Srivastava, Caetano Traina Jr., and Vassilis J. Tsotras. On query result diversification. In *ICDE*, pages 1163–1174, 2011.

[19] Xin Wang, Yang Wang, Yang Xu, Ji Zhang, and Xueyan Zhong. Extending graph pattern matching with regular expressions. In *DEXA*, pages 111–129, 2020.

[20] Xin Wang and Huayi Zhan. Approximating diversified top-k graph pattern matching. In *DEXA*, pages 407–423, 2018.