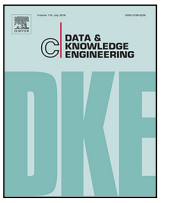
Data & Knowledge Engineering 127 (2020) 101798

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak



Top-*k* user-specified preferred answers in massive graph databases

Noseong Park [a], Andrea Pugliese [b,*], Edoardo Serra [c], V.S. Subrahmanian [d]

[a] *Yonsei University, South Korea*
[b] *University of Calabria, Italy*
[c] *Boise State University, USA*
[d] *Dartmouth College, USA*

ARTICLE INFO

Keywords:
Graph databases
Top-k querying
Preferred answers

ABSTRACT

There are numerous applications where users wish to identify subsets of vertices in a social network or graph database that are of interest to them. They may specify sets of patterns and vertex properties, and each of these confers a score to a subgraph. The users want to find the subgraphs with top-*k* highest scores. Examples in the real world where such subgraphs involve custom scoring methods include: techniques to identify sets of coordinated influence bots on Twitter, methods to identify suspicious subgraphs of nodes involved in nuclear proliferation networks, and sets of sockpuppet accounts seeking to illicitly influence star ratings on e-commerce platforms. All of these types of applications have numerous custom scoring methods. This motivates the concept of Scoring Queries presented in this paper — unlike past work, an important aspect of scoring queries is that the users get to choose the scoring mechanism, not the system. We present the Advanced top-*k* (ATK) algorithm and show that it intelligently leverages graph indexes from the past but also presents novel pruning opportunities. We present an implementation of ATK showing that it beats out a baseline algorithm that builds on advanced subgraph matching methods with multiple graph database backends including Jena and GraphDB. We show that ATK scales well on real world graph databases from YouTube, Flickr, IMDb, and CiteSeerX.

1. Introduction

The need to efficiently search graph-structured resources such as the Internet and social networks is increasing dramatically. Major companies track what is being said about them on online social networks in order to monitor their products, brands, and reputations. Financial institutions monitor what is being said on social media in order to forecast movements of stock prices, possibly in conjunction with other variables. Law enforcement organizations monitor chatter on online social networks such as Facebook and Twitter to see if violence is being planned at protests. Governments monitor what is being said online in order to track a variety of terrorist groups. Also related is the Semantic Web, where RDF datasets can be viewed as massive graph databases and (large fragments of) SPARQL queries can be viewed as subgraph matching queries, with the problem of query answering being viewed as a kind of graph matching problem.

In particular, there are many applications where *users wish to identify subsets of vertices in the graph that are of interest to them*. In nuclear proliferation networks, for instance, security analysts know they must look for certain subgraphs in order to find suspicious entities. A nuclear smuggling front operation needs to have links with businesses with logistics/transportation capabilities, businesses with manufacturing capabilities of different types and financial businesses in order to buy the requisite items, move them to the desired destination, and pay for them. Many different methods have been proposed in the literature to score suspicious subgraphs of

* Corresponding author.
E-mail address: andrea.pugliese@unical.it (A. Pugliese).

https://doi.org/10.1016/j.datak.2020.101798
Received 23 October 2017; Received in revised form 31 October 2019; Accepted 8 February 2020
Available online 14 February 2020

such types in nuclear [1] and other contexts [2]. For this, they may specify subgraph queries structured in "blocks", some of which are "fixed" in the sense that they must be completely matched by every query answer. Multiple answers in the graph databases may end up matching such queries. The users, however, wish to score the answers in some way, also taking into account which blocks are matched by each answer. But as they know their mission and application needs better than most designers of graph DBMSs, they want to specify how to score the answers as part of their query. From the graph DBMS perspective, allowing users to specify scoring mechanisms poses a challenge because they do not know the query in advance and hence cannot plan indexing and query optimization methods as well as they could if they knew how scoring was being done. While almost all past work on top-k queries to graph databases do not allow users to express scoring methods within their queries, in this paper, we propose methods that will bring this power to the users.

In the next section we present a few sample application scenarios where such a capability is useful.

1.1. Application scenarios

- *Identifying malicious actors in networks.* There is now a huge body of work on identifying malicious actors in networks that seek to attach suspicion scores to both individual nodes and subgraphs in networks. For instance, [1] looks at nuclear smuggling networks and suggests a set of measures to identify suspicious nodes in such networks. But it does so by noting that for a node to be suspicious, it must be involved in sub-networks that allow that node to perform activities relevant to nuclear smuggling, i.e. it must be linked to nodes with capabilities to manufacture relevant parts, to transport the relevant material, and to finance the transaction as well as to route the financial transaction through multiple financial institutions. The more such subnetworks a given node is involved in, the more suspicious it is. Our team has already found new suspicious entities using this method. Similar methods have more recently been used to identify suspicious entities in networks obtained from the cache of documents released via the Panama papers scandal [2]. All of these applications use *custom scoring* methods appropriate to their problem. When a security analyst uses a GUI consisting of the results of these suspicion scoring methods, they would like to see the relevant subgraphs involved and understand how the suspicion scores attached to the subgraphs are linked to suspicion of a particular node.
- *Potential job-matches on LinkedIn.* LinkedIn provides valuable services to companies by matching company job requirements with the skills of users. When it contracts to advertise a job, both LinkedIn and the company involved wish to ensure that the best matches for the job are found. In this case, LinkedIn may define subgraph queries that identify vertices in the LinkedIn graph database (people are vertices, edges are contacts between people) that are of interest. These queries may also require that users have certain skills, satisfy nationality requirements, or require (through another query block) that the users be linked to people already in the company who have a high job performance score and/or who have graduated from universities (and with similar GPAs) that the company has had good experiences with in the past. Each answer in the graph database that matches (some of) the query blocks is then scored by the company via a scoring function specific to each block. The company then specifies how to aggregate these scores into a unified score. For instance, if a particular user Bob is both well-connected with people in the company and comes from a school/university from which the company has had good experiences in the past, they may want to give him a higher score than if he met just one of these criteria.
- *Identifying "interesting" users on Twitter/Facebook (running example).* Suppose we are given a graph representing a set of users and companies with *work*, *friend*, and *follows* relationships (such a graph could be derived from social networks such as Twitter and Facebook). User vertices have properties named *postsAboutX* and *endorsedY* telling us the number of times the user posted about topic X (we can think of this as "mentions" in Twitter) and the number of endorsements of topic Y (we can think of this as say the number of times the person said something positive about Y according to a sentiment analysis program [3]). A company may be interested in extracting subsets of users $\{x, y, z\}$ such that (i) x works for the company, (ii) x follows both y and z, (iii) y and z are friends with each other and endorsed a specific topic of interest to the company, and (iv) all of the three users posted a minimum number of times about another topic *ABC*. By identifying users x, y, z satisfying these conditions, the company is finding small groups of users who are heavily engaged in the topic *ABC* of interest to the company and who have an existing opinion of interest to the company. For each such triple x, y, z of users, the company might select one to post official tweets for the company (perhaps for some fee). For each such triple identified, the company may use custom methods to score the quality of a candidate and then choose one candidate each from the top-k triples to help spread the company's message. During the extraction of such users, the company may be interested in giving higher scores to the users which posted more about *ABC*, both when they find users that satisfy each block in the resulting queries and when they aggregate scores across the blocks. In the remainder of the paper, we will adopt this scenario as our running example. Note that triples are just one example here — the company may define a "group" of relevant users with other patterns.

1.2. Plan of the paper

The organization of this paper is as follows. Related work is discussed in Section 2. In Section 3, we present formal definitions of graph databases and scoring queries, together with the notion of top-k answers to such queries — moreover, we provide an initial baseline algorithm that contains various optimizations for subgraph matching. Section 4 presents our ATK (Advanced top-k) algorithm that uses two types of pruning, along with an extension of the DOGMA graph index [4] to efficiently answer top-k queries. Section 5 presents the results of experiments in which we compare our ATK algorithm with the baseline on four real-world graph datasets from CiteSeerX, IMDb, YouTube, and Flickr. We also compare our ATK algorithm with the baseline built on top of two commercial RDF engines: Jena and GraphDB. Finally, Section 6 outlines conclusions and future work.

2. Related work

We consider related work in three areas. We start by looking at approaches to the general subgraph matching problem, where the objective is to find all exact matches of a given subgraph. Then, we discuss techniques for approximate subgraph matching, where inexact matches are allowed and evaluated using scoring functions. Finally, we consider past work specifically targeted at top-k query processing and provide a comparison with our proposed approach.

- *General subgraph matching.* The problem of efficiently evaluating subgraph matching queries over huge graphs has been addressed in different scenarios, among which social network analysis and RDF database management play an important role [5]. Many social network analysis methods [6,7] operate in main memory. For social networks of the size of Facebook and Flickr, such an approach is infeasible and disk-based data management becomes mandatory. More importantly, complex queries involving even a few joins can quickly cause such approaches to run into trouble. In the RDF realm, approaches differ with respect to their storage regime and query answering strategies [8]. However, the great majority of RDF databases are triple oriented, in the sense that they focus on the storage and retrieval of individual triples. Some systems (e.g. [9–11]) use relational databases as their back-end and a relational query engine to answer queries. Others propose native storage formats for RDF (e.g. [4,12]). [13] shows that storing RDF in a vertical database leads to significant query time improvements, whereas [14] focuses on physical data structures and join processing. [15] uses triple selectivity estimation techniques similar to those used in relational database systems. With regard to XML data, a tree semantics is the most common assumption, but some works look at the graph structure induced by ID references and study the problem of querying under a graph semantics. For instance, [16] proposes an approach to reduce the number of results by first applying a filtering technique to eliminate "semantically weak" answers, then a ranking technique for presenting the remaining answers in decreasing order of "semantic significance". The approach studied classes of ranking functions under which query evaluation is tractable. Other works on general graph data management and subgraph matching (e.g. [17–22]) typically employ heuristics to predict the cost of answering strategies based on statistics about the dataset and the current state of query processing and then choose a strategy to minimize cost. For instance, [23] proposes to transform vertices into points in a vector space, thus converting queries into distance-based multi-way joins over the vector space. [24] proposes a two-step join optimization algorithm based on a cluster based join index. GADDI [25] employs a structural distance based approach and a dynamic matching scheme to minimize redundant computations. GADDI can handle graphs with thousands of vertices, which are common in many biological applications. [21] proposes SUMMA which improves GADDI and employs more advanced indices, becoming capable to handle graphs with up to tens of millions of vertices. The algorithm in [22] employs an aggressive pruning strategy based on an index storing label distributions. [26] argues that existing indices over *sets* of data graphs do not support efficient pruning when they face graphs with tens of thousands of vertices. They propose an index that is specifically targeted at this query evaluation scenario. COSI [27] built upon DOGMA and proposed a cloud-based algorithm for subgraph matching that works very efficiently on graphs with over 770M edges. [28] extends DOGMA to handle skewed degree distributions in graph databases and shows how to quickly answer complex subgraphs queries on graph data with over 1B edges. None of the above works handle uncertainty or user defined scoring in their queries. A comprehensive evaluation of the performance and scalability of indexed subgraph matching techniques is presented in [29]. The authors identify a set of key factors that influence the performance (e.g., number of vertices, density, number of distinct labels, query size) and assess various techniques' relative performance when varying such factors (e.g. index construction time, index size, query evaluation time) over both real and synthetic graph datasets.
- *Approximate subgraph matching.* [30] proposed a basic algorithm for the problem of approximately matching a query against a database of *many small* graphs. The algorithm is based on A^* search whereby each vertex in the search tree represents a vertex pair matching. Each vertex pair matching has an associated cost that is inversely proportional to the "goodness" of the match. The basic notion of *edit distance* [31] between two graphs is used in many approximate subgraph matching systems proposed since. Grafil [32] tries to identify frequently occurring but selective subgraphs in a graph database, which are called *features*. Given a query graph, all features in the query are determined — this allows the computation of a feature similarity score (based on the number of missing features) and return approximate matches. SAGA [33] measures the distance between two graphs as a weighted linear combination of the structural difference, vertex mismatches, and vertex gaps. TALE [34] characterizes a vertex by its label, the list of its neighbors, and the interconnections between the neighbor vertices. The idea is to determine the subset V of "interesting" vertices in the query graph and then to look up all graphs in the database which contain the vertices in V. The measures proposed in SAGA and TALE for determining the degree of similarity between graphs are guided by the authors' intuition about the problem domain (such as the importance of vertex labels in biological networks). Another line of work [35,36] proposes a probabilistic model to align a query graph and a *host* graph. This work describes a probability distribution over the set of all possible alignments using features from either graph and then finds the most likely alignment. None of the above works have demonstrated scalability to operate on graphs with millions of vertices and edges. Probabilistic matching between a query and *one huge* graph database (upto over 1B edges) was shown in [37] — however, the proposed approach only allows setting fixed "probabilistic" scores.
- *Top-k query processing.* One way to improve the performance of triplestores for top-k SPARQL queries has been proposed in [38]. They propose a rank-aware join algorithm for queries with additive SUM scoring functions for substitutions. Our framework supports many complex user-defined scoring functions, not only restricted to SUM. [39] suggested using twig queries to score substitutions and a query edge is mapped to a path in the data. The score of a substitution is the sum of the weights of all the edges involved in the edge to path mappings. [40] extends this work to graph data. The k most similar subgraphs to a

Table 1
Comparison with related work.

	Query blocks	Scored object	Full user defined scoring	Score aggreg. and normaliz.
[38]	✗	Substitution	✓	✗
[39]	✗	Substitution	✗	✗
[40]	✗	Substitution	✗	✗
[41]	✗	Substitution	✗	✗
[42]	✗	Vertex	✗	✓
[43]	✗	Vertex	✗	✗
Proposed framework	✓	Substitution	✓	✓

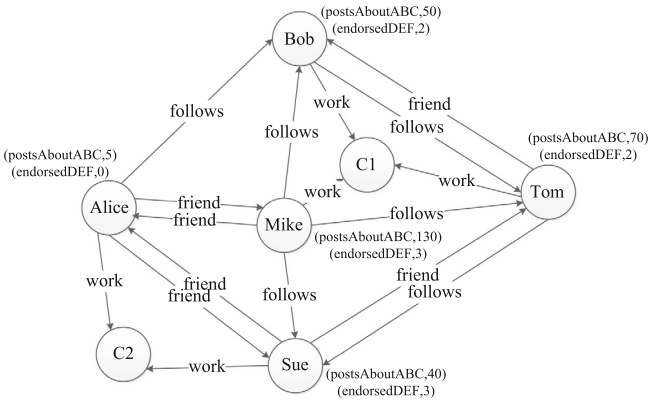


Fig. 1. Example graph database.

query graph, where similarity is defined using maximum common subgraphs, are targeted in [41]. [42] lets users score vertices via a function f that represents a "relevance" for each vertex. Once these relevance scores are computed for all vertices, a higher level is computed as $\sum_{u \in Nbr(v,h)} f(u)$, where $Nbr(v,h)$ is a set of h-hop neighbors of v. Top-k vertex set search by graph simulation has been suggested in [43], where the score of the output variable of a query is defined as the size of the maximum graph simulation matching the output variable to a vertex. Both [42] and [43] do not allow the user to choose the final scoring function. The features of some of the above works are summarized in Table 1.

3. Scoring queries on graph databases

In this section, we provide a brief summary of graph databases and define scoring queries.

3.1. Graph databases

We assume the existence of three arbitrary, but fixed, disjoint sets V, $\mathcal{L}$, $\mathcal{P}$ of *vertex names*, *edge labels*, and *vertex properties*, respectively. Each vertex property $p \in \mathcal{P}$ has an associated domain $\mathrm{dom}(p)$ which is a set (disjoint from V, $\mathcal{L}$, and $\mathcal{P}$) of values that can be assigned to p. A graph database is defined as follows.

Definition 3.1 (*Graph Database*)**.** A *graph database* $\mathcal{G}$ is a triple $\langle V, E, \wp \rangle$, where $E \subseteq V \times \mathcal{L} \times V$ is a set of labeled edges, and $\wp : V \times \mathcal{P} \to \bigcup_{p \in \mathcal{P}} \mathrm{dom}(p)$ is a (partial) *property assignment* function — if $\wp(v,p)$ is defined, then $\wp(v,p) \in \mathrm{dom}(p)$.

From now on we assume an arbitrary but fixed $\mathcal{G}$ is given.

Example 3.1 (*Running example*)**.** Fig. 1 shows the graph database we will use in our running example, corresponding to the Twitter application scenario discussed in Section 1.1. There are a set of users and companies with edge labeled links. Each vertex has two properties, *postsAboutABC* and *endorsedDEF* telling us the number of times the vertex posted about *ABC* and the number of endorsements of *DEF*. For instance, we see that the vertex "Bob" posted 50 posts in which he mentioned *ABC* and 2 posts in which he expressed positive sentiment about *DEF*.

3.2. Scoring queries

Let VAR be a set of variable symbols — throughout this paper, a variable "v" will be denoted by $\tilde{v}$. We assume VAR to be disjoint from V, $\mathcal{L}$, $\mathcal{P}$, and $\bigcup_{p \in \mathcal{P}} \mathrm{dom}(p)$.

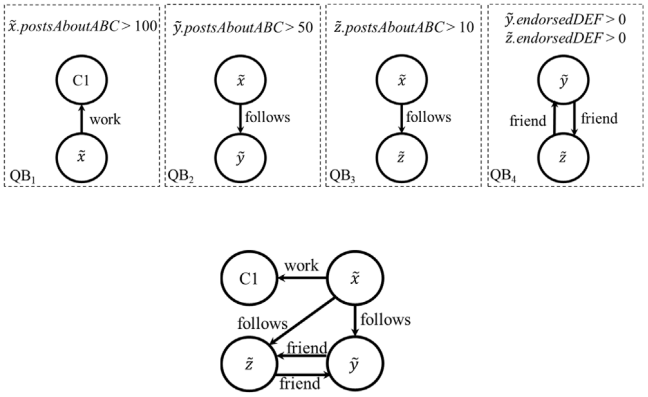


Fig. 2. Example query blocks (top) and corresponding graph (bottom).

Terms are defined as follows. (i) Any variable, member of $\bigcup_{p\in\mathcal{P}} \mathrm{dom}(p)$, or real number is a term. (ii) If $\tilde{v}$ is a variable and $p \in \mathcal{P}$ is a (numerical) property, then $\tilde{v}.p$ is a (numerical) term. (iii) If $t_1, t_2, \ldots, t_n$ are numerical terms, then $f(t_1, \ldots, t_n)$ is a numerical term where f is any function from some designated set of functions.[1] For instance, $(\tilde{v}.level - 1)$, $(\tilde{v}.level - 10) * (\tilde{y}.level + 2)$ are both terms. If t_1, t_2 are numerical terms and $op \in \{\leq, <, =, \geq, >\}$, then $t_1\ op\ t_2$ is a *numerical constraint*. If C_1, C_2 are numerical constraints, then so are $C_1 \wedge C_2, C_1 \vee C_2$ and $\neg C_1$. For instance, $\tilde{x}.level > \tilde{y}.level$ and $(\tilde{x}.level > 5 \wedge \tilde{x}.level \leq \tilde{y}.level + 3)$ are both constraints.

We are now ready to introduce the notion of query block. Let $*$ be a distinguished "wildcard" label not occurring in $\mathcal{L}$.

Definition 3.2 (*Query Block*)**.** A *query block* is a triple $\mathrm{QB} = \langle V_{\mathrm{QB}}, E_{\mathrm{QB}}, \chi_{\mathrm{QB}} \rangle$, where $V_{\mathrm{QB}} \subseteq V \cup \mathsf{VAR}$, $E_{\mathrm{QB}} \subseteq V_{\mathrm{QB}} \times (2^{\mathcal{L}} \cup \{*\}) \times V_{\mathrm{QB}}$, and χ_{QB} is a finite set of constraints over variables in V_{QB}.

Thus, a query block is a directed edge-labeled graph where (i) each vertex is either a vertex name or a variable symbol, and (ii) each edge is labeled with a set of edge labels (or $*$). Intuitively, the users use this graph to specify the structure of the sub-graphs of $\mathcal{G}$ they are looking for ($*$ matches any label). Finally, the constraints in χ_{QB} express requirements about variables or their properties. A vertex in $\mathcal{G}$ can only "match" a variable if it satisfies the constraints associated with that variable. We use $\mathsf{VAR}_{\mathrm{QB}}$ to denote the set of all variables in a query block QB, i.e. $\mathsf{VAR}_{\mathrm{QB}} = \mathsf{VAR} \cap V_{\mathrm{QB}}$.

Example 3.2. Fig. 2 (top) shows the four query blocks we use in our running example, and Fig. 2 (bottom) shows the corresponding graph.

Given two query blocks $\mathrm{QB} = \langle V_{\mathrm{QB}}, E_{\mathrm{QB}}, \chi_{\mathrm{QB}} \rangle$ and $\mathrm{QB}' = \langle V_{\mathrm{QB}'}, E_{\mathrm{QB}'}, \chi_{\mathrm{QB}'} \rangle$, we define the union of QB and QB′ as the query block $\langle V_{\mathrm{QB}} \cup V_{\mathrm{QB}'}, E_{\mathrm{QB}} \cup E_{\mathrm{QB}'}, \chi_{\mathrm{QB}} \cup \chi_{\mathrm{QB}'} \rangle$.

A (partial) *substitution* θ is a (partial) mapping $\theta : \mathsf{VAR} \to V$ where $\mathrm{dom}(\theta) \subseteq \mathsf{VAR}$ denotes the set of variables for which θ is defined. Applying θ to an expression e (vertex, term, or constraint) means replacing every variable $\tilde{x} \in \mathrm{dom}(\theta)$ occurring in it with vertex name $\theta(\tilde{x})$, and the expression resulting from such an application is denoted as $e\theta$. The composition of two substitutions θ and θ' with disjoint domains, denoted $\theta\theta'$, is the substitution that maps each variable $\tilde{x} \in \mathrm{dom}(\theta)$ to $\tilde{x}\theta$ and each variable $\tilde{x} \in \mathrm{dom}(\theta')$ to $\tilde{x}\theta'$. A ground expression is an expression with no variables — ground terms and ground constraints are evaluated in the obvious way.

Definition 3.3 (*Answer Substitution for a Query Block*)**.** Suppose $\mathrm{QB} = \langle V_{\mathrm{QB}}, E_{\mathrm{QB}}, \chi_{\mathrm{QB}} \rangle$ is a query block. An *answer substitution* θ for QB is a substitution such that (i) $\mathrm{dom}(\theta) = \mathsf{VAR}_{\mathrm{QB}}$, (ii) for each $(\alpha, SL, \beta) \in E_{\mathrm{QB}}$, there is an edge (v_1, ℓ, v_2) in $\mathcal{G}$ such that $v_1 = \alpha\theta$, $v_2 = \beta\theta$, and $\ell \in SL$ if $SL \neq \{*\}$; (iii) for each constraint $C \in \chi_{\mathrm{QB}}$, it is the case that $C\theta$ evaluates to true.

Example 3.3. Consider query block QB_3 of our running example. The answer substitutions for QB_3 are shown below.

	θ_1	θ_2	θ_3	θ_4	θ_5
$\tilde{x}$	Bob	Mike	Mike	Mike	Tom
$\tilde{z}$	Tom	Bob	Tom	Sue	Sue

Note that $\{\tilde{x}/Bob, \tilde{z}/Alice\}$ is not an answer substitution for QB_3 because the constraint $\tilde{z}.postsAboutABC > 10$ is violated, as $\wp(Alice, postsAboutABC) = 5$.

[1] In this paper, we assume that addition, multiplication, subtraction, division, exponentiation, are all functions. Others may be added as needed.

The existence of an answer substitution θ for a query block QB guarantees that there is a way of mapping all the variables in QB to vertices of the graph database so that it satisfies all other requirements specified by QB: presence of vertex names specified in the query and satisfaction of the constraints. We use AS(QB) to denote the set of all answer substitutions for QB. A substitution θ is said *partial for* QB if $\mathrm{dom}(\theta) \neq \mathsf{VAR}_{\mathrm{QB}}$.

Definition 3.4 (*Scoring Function*)**.** Let $\mathrm{QB} = \langle V_{\mathrm{QB}}, E_{\mathrm{QB}}, \chi_{\mathrm{QB}} \rangle$ be a query block and θ a substitution in AS(QB). A *scoring function* SF is a function that assigns a value $\mathrm{SF}(\mathrm{QB}, \theta) \in \mathbb{R}$ to the pair (QB, θ).

We assume that scoring functions are *monotonically increasing* in the following sense. In most real-world applications, $\mathrm{SF}(\mathrm{QB}, \theta)$ will be computed from the subgraph of the database corresponding to the answer θ by capturing some values associated with each node in the answer subgraph. For instance, in our running Twitter example (cf. Example 3.3) looking for three nodes $\tilde{x}, \tilde{y}, \tilde{z}$, a solution θ has a concrete value for the number of posts about *ABC* by the answer nodes $\tilde{x}\theta, \tilde{y}\theta, \tilde{z}\theta$. A scoring function is monotonically increasing if its value increases when the values of the selected properties of these answer nodes increase. In this sense, we can think of SF as a function from $\mathbb{R}^h$ to $\mathbb{R}$ whose h arguments are obtained from vertices/properties in the answer to a query block and we say that $\mathrm{SF}(\mathrm{QB}, \theta)$ is monotonically increasing if there exists a partial order $\preceq_m$ over $\mathbb{R}^h$ such that $\forall X, Y \in \mathbb{R}^h$, it holds that $X \preceq_m Y \Rightarrow \mathrm{SF}(X) \leq \mathrm{SF}(Y)$.[2] In Example 3.4 below, we have $h = 2$.

Example 3.4. In our running example, a possible scoring function is $\mathrm{SF}(\mathrm{QB}_3, \theta) = e^{-10^3/p}$ with $p = \wp(\tilde{z}\theta, postsAboutABC) + \wp(\tilde{x}\theta, postsAboutABC)$. This function is monotonically increasing because if we write p as $a + b$ then the needed partial order $\preceq_m$ is $[a, b] \preceq_m [a', b']$ if $a \leq a'$ and $b \leq b'$.

Definition 3.5 (*Scoring Query Block*)**.** A *scoring query block* is a triple $\mathrm{SQB} = \langle \mathrm{QB}, \mathrm{SF}, \mathrm{AV} \rangle$ where QB is a query block, SF is a scoring function, and $\mathrm{AV} \subseteq \mathsf{VAR}_{\mathrm{QB}}$.

Example 3.5. In our running example, two possible scoring query blocks are $\mathrm{SQB}_{3,1} = \langle \mathrm{QB}_3, \mathrm{SF}_{3,1}, \emptyset \rangle$ and $\mathrm{SQB}_{3,2} = \langle \mathrm{QB}_3, \mathrm{SF}_{3,2}, \{\tilde{z}\} \rangle$ where $\mathrm{SF}_{3,1}(\mathrm{QB}_3, \theta)$ is the scoring function of Example 3.4 and $\mathrm{SF}_{3,2}(\mathrm{QB}_3, \theta) = \wp(\tilde{z}\theta, postsAboutABC)$.

The meaning of specifying $\mathrm{AV} = \emptyset$ will be clearer shortly. Our notion of scoring query blocks is rich enough to express normalized "preference" scoring, capturing the preference expressed by the user over a given subgraph of the graph database.

Definition 3.6 (*Preference Scoring*)**.** Given a scoring query block $\mathrm{SQB} = \langle \mathrm{QB}, \mathrm{SF}, \mathrm{AV} \rangle$ and a substitution $\theta \in \mathrm{AS}(\mathrm{QB})$, the *preference scoring* of θ in SQB is

$$\mathrm{SF}(\mathrm{SQB}, \theta) = \begin{cases} \mathrm{SF}(\mathrm{QB}, \theta), & \text{if } \mathrm{AV} = \emptyset \\ \frac{\mathrm{SF}(\mathrm{QB}, \theta)}{\sum_{\theta' \in \mathrm{AS}(\mathrm{QB}\theta|_{\bar{\mathrm{AV}}})} \mathrm{SF}(\mathrm{QB}, \theta|_{\bar{\mathrm{AV}}}\theta')}, & \text{otherwise,} \end{cases}$$

where $\theta|_{\bar{\mathrm{AV}}}$ is θ restricted to the subdomain $\bar{\mathrm{AV}} = \mathrm{dom}(\theta) - \mathrm{AV}$.

Intuitively, if the set AV is not empty, then we normalize the scores of all substitutions for the query block $\mathrm{QB}\theta|_{\bar{\mathrm{AV}}}$ in order to get a preference score. We refer to this as *normalized score*.

Example 3.6. In our running example, suppose we want to compute the preference scoring of substitution $\theta_2 = \{\tilde{x}/Mike, \tilde{z}/Bob\}$ in $\mathrm{SQB}_{3,1}$ and $\mathrm{SQB}_{3,2}$. We have $\mathrm{SF}(\mathrm{SQB}_{3,1}, \theta_2) = e^{-10^3/p} = 0.0038$ with $p = \wp(\tilde{z}\theta, postsAboutABC) + \wp(\tilde{x}\theta, postsAboutABC) = 180$. In order to compute $\mathrm{SF}(\mathrm{SQB}_{3,2}, \theta_2)$ we need to look at the answer substitutions in $\mathrm{AS}(\mathrm{QB}_3\theta|_{\bar{\mathrm{AV}}})$ where $\theta|_{\bar{\mathrm{AV}}}$ is $\{\tilde{x}/Mike\}$ — therefore, $\mathrm{QB}_3\theta|_{\bar{\mathrm{AV}}}$ is the third query block of Fig. 2 after replacing $\tilde{x}$ with *Mike*. The set $\mathrm{AS}(\mathrm{QB}_3\theta|_{\bar{\mathrm{AV}}})$ contains the substitutions $\theta' = \{\tilde{z}/Bob\}$, $\theta'' = \{\tilde{z}/Tom\}$, and $\theta''' = \{\tilde{z}/Sue\}$. We therefore have[3]

$$\begin{aligned} \mathrm{SF}(\mathrm{SQB}_{3,2}, \theta_2) &= \frac{\mathrm{SF}_{3,2}(\mathrm{QB}_3, \theta_2)}{\mathrm{SF}_{3,2}(\mathrm{QB}_3, \theta|_{\bar{\mathrm{AV}}}\theta') + \mathrm{SF}_{3,2}(\mathrm{QB}_3, \theta|_{\bar{\mathrm{AV}}}\theta'') + \mathrm{SF}_{3,2}(\mathrm{QB}_3, \theta|_{\bar{\mathrm{AV}}}\theta''')} \\ &= \frac{\wp(Bob, pAA)}{\sum_{v \in \{\mathrm{Bob,Tom,Sue}\}} \wp(v, pAA)} = 0.3333 \end{aligned}$$

Note that $\theta|_{\bar{\mathrm{AV}}}\theta' = \theta_2$, $\theta|_{\bar{\mathrm{AV}}}\theta'' = \theta_3$, and $\theta|_{\bar{\mathrm{AV}}}\theta''' = \theta_4$.

We now define scoring queries.

Definition 3.7 (*Scoring Query*)**.** A *scoring query* is a tuple $\mathrm{SQ} = \langle S, F, \chi^g, \mathrm{SF}^g \rangle$ where:

- $S = \{\mathrm{SQB}_1, \ldots, \mathrm{SQB}_m\}$ with $\mathrm{SQB}_i = \langle \mathrm{QB}_i = \langle V_{\mathrm{QB}_i}, E_{\mathrm{QB}_i}, \chi_{\mathrm{QB}_i} \rangle, \mathrm{SF}_i, \mathrm{AV}_i \rangle$.
- $F \subseteq S$ and for each $\langle V_{\mathrm{QB}_i}, E_{\mathrm{QB}_i}, \chi_{\mathrm{QB}_i} \rangle \in F$, $V_{\mathrm{QB}_i} \cap V \neq \emptyset$.

[2] A user who wishes to include categorical attributes and predicates over them in her scoring function, could specify (i) a total order on the domain of every categorical attribute, (ii) a function that combines these "local" total orders into a "global" partial order, also taking into account predicates, and (iii) a way to transform such partial order into a monotonically increasing scoring function.

[3] In the remainder of the paper, for the sake of readability, we sometimes use *pAA* as a shorthand for *postsAboutABC*.

- χ^g is a set of global constraints on $\bigcup_{i=1}^{m} V_{\mathrm{QB}_i}$.
- $\mathrm{SF}^g : \{0,1\}^m \times [0,1]^m \rightarrow [0,1]$ is a "global" scoring function that combines the scores from the scoring query blocks.

Intuitively, this query defines a set S of scoring query blocks, a subset F of which must be matched. It includes constraints χ^g on the variables in S and a global scoring function that merges the scores from each scoring query block. We now define the concept of a "valid" subset of a scoring query.

Definition 3.8 (*Valid Subset of a Scoring Query*)**.** Given a scoring query $\mathrm{SQ} = \langle S, F, \chi^g, \mathrm{SF}^g \rangle$, a *valid subset of* SQ is a set $\mathrm{Sub}^{\mathrm{SQ}} \subseteq S$ such that $F \subseteq \mathrm{Sub}^{\mathrm{SQ}}$ and for each $\mathrm{SQB}_h \in \mathrm{Sub}^{\mathrm{SQ}} \setminus F$, there exists a sequence $\mathrm{SQB}_1, \ldots, \mathrm{SQB}_s$ with $\mathrm{SQB}_i \in \mathrm{Sub}^{\mathrm{SQ}} \setminus (F \cup \{\mathrm{SQB}_h\})$ such that:

- there exists a scoring query block $\mathrm{SQB}_l \in F$ which shares at least one variable with QB_1;
- scoring query blocks QB_i and QB_{i+1} share at least one variable, for each $i \in \{1, \ldots, s-1\}$;
- scoring query blocks QB_s and QB_h share at least one variable.

The set of all valid subsets of SQ is denoted by $\mathrm{USub}^{\mathrm{SQ}}$.

The above definition basically requires that for each scoring query block $\mathrm{SQB}_h \in \mathrm{Sub}^{\mathrm{SQ}} \setminus F$ there exists a sequence of scoring query blocks in $\mathrm{Sub}^{\mathrm{SQ}} \setminus (F \cup \{\mathrm{SQB}_h\})$ that connects the variables in SQB_h with the variables in the query blocks in F.

Example 3.7. In our running example, consider the scoring query $\mathrm{SQ} = \langle \{\mathrm{SQB}_1, \mathrm{SQB}_2, \mathrm{SQB}_3, \mathrm{SQB}_4\}, \{\mathrm{SQB}_1\}, \emptyset, \mathrm{SF}^g \rangle$ where SQB_1, SQB_2, SQB_3, and SQB_4 are associated with QB_1, QB_2, QB_3, and QB_4 in Fig. 2, respectively. We have $\mathrm{USub}^{\mathrm{SQ}} = \{\{\mathrm{SQB}_1\}, \{\mathrm{SQB}_1, \mathrm{SQB}_2\}, \{\mathrm{SQB}_1, \mathrm{SQB}_3\}, \{\mathrm{SQB}_1, \mathrm{SQB}_2, \mathrm{SQB}_4\}, \{\mathrm{SQB}_1, \mathrm{SQB}_3, \mathrm{SQB}_4\}, \{\mathrm{SQB}_1, \mathrm{SQB}_2, \mathrm{SQB}_3, \mathrm{SQB}_4\}\}$. Note that the subset $\{\mathrm{SQB}_1, \mathrm{SQB}_4\}$ is not valid because there is no connection between the variables in SQB_1 and SQB_4.

Definition 3.9 (*Substitution for a Scoring Query*)**.** Suppose $\mathrm{SQ} = \langle S, F, \chi^g, \mathrm{SF}^g \rangle$ is a scoring query and $\mathrm{Sub}^{\mathrm{SQ}} \in \mathrm{USub}^{\mathrm{SQ}}$. Let $\mathrm{mod}\,\mathrm{QB}(\mathrm{SQ}, \mathrm{Sub}^{\mathrm{SQ}})$ be the query block obtained as $\{\langle \emptyset, \emptyset, \chi^g \rangle\} \cup \bigcup_{\langle \mathrm{QB}, \cdot, \cdot \rangle \in \mathrm{Sub}^{\mathrm{SQ}}} \mathrm{QB}$. A *substitution for* SQ *w.r.t.* $\mathrm{Sub}^{\mathrm{SQ}}$ is a substitution in $\mathrm{AS}(\mathrm{mod}\,\mathrm{QB}(\mathrm{SQ}, \mathrm{Sub}^{\mathrm{SQ}}))$.

Definition 3.10 (*Score of a Substitution*)**.** Suppose we are given a scoring query $\mathrm{SQ} = \langle S, F, \chi^g, \mathrm{SF}^g \rangle$, a valid subset $\mathrm{Sub}^{\mathrm{SQ}}$ of SQ, and a substitution θ for SQ w.r.t. $\mathrm{Sub}^{\mathrm{SQ}}$. The score of $\mathrm{Sub}^{\mathrm{SQ}}$ and θ is $P(\mathrm{Sub}^{\mathrm{SQ}}, \theta) = \mathrm{SF}^g(D, M)$ where D and M are two vectors (in $\{0,1\}^m$ and $[0,1]^m$, respectively) such that if $\mathrm{SQB}_i \notin \mathrm{Sub}^{\mathrm{SQ}}$, then $d_i = 0$ and $m_i = 0$; otherwise, $d_i = 1$ and $m_i = \mathrm{SF}(\langle \mathrm{mod}\,\mathrm{QB}(\mathrm{SQ}, \mathrm{Sub}^{\mathrm{SQ}}), \mathrm{SF}_i, \mathrm{AV}_i \rangle, \theta)$.

We assume SF^g is monotonically increasing w.r.t. D and M, i.e. $\mathrm{SF}^g(D, M) \geq \mathrm{SF}^g(D', M')$ whenever $d_i \geq d'_i$ and $m_i \geq m'_i$ for all i, where $d_i \in D$, $d'_i \in D'$, $m_i \in M$, and $m'_i \in M'$.

We are now ready to define the top-k preferred answers to a scoring query.

Definition 3.11 (*Top-k Answers to a Scoring Query*)**.** Given a scoring query $\mathrm{SQ} = \langle S, F, \chi^g, \mathrm{SF}^g \rangle$ and a non-negative integer k, we define the *top-k answers to* SQ as the set $Top\text{-}k = \{(\mathrm{Sub}^{\mathrm{SQ}_1}, \theta_1), \ldots, (\mathrm{Sub}^{\mathrm{SQ}_k}, \theta_k)\}$ such that:

- $\mathrm{Sub}^{\mathrm{SQ}_i} \in \mathrm{USub}^{\mathrm{SQ}}$ and θ_i is a substitution for SQ w.r.t. $\mathrm{Sub}^{\mathrm{SQ}_i}$ (for each $i \in \{1, \ldots, k\}$);
- for each pair $(\mathrm{Sub}^{\mathrm{SQ}}, \theta)$ were θ is a substitution for SQ w.r.t. $\mathrm{Sub}^{\mathrm{SQ}}$ and $(\mathrm{Sub}^{\mathrm{SQ}}, \theta) \notin Top\text{-}k$, we have that $\forall (\mathrm{Sub}^{\mathrm{SQ}_i}, \theta_i) \in Top\text{-}k$, $P(\mathrm{Sub}^{\mathrm{SQ}_i}, \theta_i) \geq P(\mathrm{Sub}^{\mathrm{SQ}}, \theta)$.

Example 3.8. Suppose for th sake of example that for all query blocks of our running example, we want to compute a score using the sigmoid function $sigmoid(t) = \frac{1}{1+e^{-t}}$ over the sum of variables denoting *postsAboutABC* property values. As our global scoring function, we want to use the sigmoid function over the average of D, multiplied by the average of M. In this case we have:

- $\mathrm{SQB}_1 = \langle \mathrm{QB}_1, \mathrm{SF}_1 = sigmoid(\frac{\tilde{x}.pAA}{1K}), \mathrm{AV}_i = \{\tilde{x}\} \rangle$
- $\mathrm{SQB}_2 = \langle \mathrm{QB}_2, \mathrm{SF}_2 = sigmoid(\frac{\tilde{x}.pAA + \tilde{y}.pAA}{1K}), \mathrm{AV}_i = \emptyset \rangle$
- $\mathrm{SQB}_3 = \langle \mathrm{QB}_3, \mathrm{SF}_3 = sigmoid(\frac{\tilde{x}.pAA + \tilde{z}.pAA}{10K}), \mathrm{AV}_i = \emptyset \rangle$
- $\mathrm{SQB}_4 = \langle \mathrm{QB}_4, \mathrm{SF}_4 = sigmoid(\frac{\tilde{z}.pAA + \tilde{y}.pAA}{10K}), \mathrm{AV}_i = \emptyset \rangle$
- $\mathrm{SQ} = \langle \{\mathrm{SQB}_1, \mathrm{SQB}_2, \mathrm{SQB}_3, \mathrm{SQB}_4\},\ F = \{\mathrm{SQB}_1\},\ \chi^G,\ \mathrm{SF}^G = sigmoid(\frac{\sum_{d_i \in D} d_i}{4} \times \frac{\sum_{m_i \in M} m_i}{4}) \rangle$.

3.3. Baseline top-k answering algorithm

A straightforward algorithm to compute the top-k answers to a scoring query first computes all answer substitutions and then computes the score of each answer substitution in order to find the best k. Any subgraph matching algorithm could be used for this purpose. In this section, we describe our baseline algorithm, which builds upon the DOGMA framework [4] that considers our specific problem situation (queries with constants and large disk-resident graphs).

The following definition merges together a set of scoring query blocks.

Definition 3.12 (*Unified Query*)**.** Given a scoring query SQ and a set $\text{Sub}^{\text{SQ}} = \{\text{SQB}_1, \text{SQB}_2, \ldots, \text{SQB}_n\} \in \text{USub}^{\text{SQ}}$, with $\text{SQB}_i = \langle \text{QB}_i, \text{SF}_i, \text{AV}_i \rangle$ and $\text{QB}_i = \langle V_{\text{QB}_i}, E_{\text{QB}_i}, \chi_{\text{QB}_i} \rangle$, the *unified query of* Sub^{SQ} is *unified* $(\text{Sub}^{\text{SQ}}) = \langle \bigcup_{i=1}^{n} V_{\text{QB}_i}, \bigcup_{i=1}^{n} E_{\text{QB}_i}, \bigcup_{i=1}^{n} \chi_{\text{QB}_i} \rangle$.

Algorithm 1 is our Base algorithm.[4] For each valid subset Sub^{SQ} of SQ, answer substitutions are found by applying findSubstitutions to the unified query. A score for each answer substitution is computed, and finally the top-k are returned. In the algorithm, $\text{dom}(\theta_\emptyset) = \emptyset$.

ALGORITHM 1: Base

```
1  Function answerQuery (SQ)
   Input: Scoring query SQ = ⟨S, F, χ^g, SF^g⟩
   Output: Top-k answers and their scores
2  S ← ∅;
3  foreach Sub^SQ ∈ USub^SQ do
4  |   foreach θ ∈ findSubstitutions(unified(Sub^SQ), θ_∅) do
5  |   |   S ← S ∪ { (θ, computeScore(Sub^SQ, θ))};
6  return top-k of S

7  Function findSubstitutions (Q, θ)
   Input: Unified query Q = ⟨V_Q, E_Q, χ_Q⟩, partial substitution θ
   Output: Set AS of answer substitutions obtained by extending θ
8  if θ maps every variable in Q then
9  |   return θ;
10 NV ← {(c, ṽ)|(ṽ, c) or (c, ṽ) ∈ E_Q };
11 foreach (c, ṽ) ∈ NV do
12 |   R_{c,ṽ} ← getNeighborNum(c, edgeLabel(c, ṽ));
13 |   if R_{c,ṽ} = 0 then return ∅;
14 (c, ũ) ← (c, ṽ) ∈ NV with minimum R_{c,ṽ} ;
15 N_ũ ←getValidNeighbors(Q, c, ũ) ;
16 AS ← ∅ ;
17 foreach m ∈ N_ũ do
18 |   θ′ ← θ ∪ {ũ/m};
19 |   AS ← AS∪ answerQuery(Qθ′,θ′);
20 return AS;
```

The findSubstitutions function uses function getNeighborNum that returns the number of c's neighbors which are connected through an edge with a given label. Moreover, function getValidNeighbors, when applied to a unified query Q, a vertex c, and a variable $\tilde{u}$, returns the neighbors of c that can be mapped to $\tilde{u}$ given E_Q and χ_Q. At this point, we access the disk index and also use DOGMA's pruning technique based on IPD values to filter neighbors that cannot be part of a valid answer (see Section 4.1).

Instead of doing massive table-joins [44–46], findSubstitutions uses DOGMA to perform a depth-first graph traversal search from constants. Line 8 checks whether a complete substitution for the unified query has been generated. Line 10 inspects every edge of the having one end mapped to a vertex of the graph and not the other. $R_{c,\tilde{v}}$ is the number of c's neighbors in the graph that are connected through an edge having the same label as the one between c and $\tilde{v}$ — therefore, $R_{c,\tilde{v}}$ is the number of candidates for $\tilde{v}$. In line 14 we select the edge in the subgraph query with the lowest number of candidates. In line 17, we substitute $\tilde{w}$ with each candidate m and recursively continue the assembly of answers.

4. ATK algorithm

This section describes the proposed advanced pruning ATK algorithm. We start by briefly discussing the index structure used by the algorithm for both subgraph matching and the computation of upper bounds on scores. We then describe the overall structure of the algorithm and its main components.

4.1. ATK graph database index

ATK employs a disk-based index that leverages the DOGMA index and extends it to store additional information. DOGMA decomposes the data graph into a large number of small, densely connected subgraphs and stores them in a binary tree structured index. Initially, it iteratively halves the number of vertices by merging randomly selected vertices with all of their neighbors. This process is repeated till a graph with less than or equal to Z vertices is reached. The graph DBMS administrator can choose Z so that graphs will fit into a disk page and their partitioning can be done in reasonable time (in our experiments, we set $Z = 100$). After reaching a single graph, DOGMA builds a binary tree by iteratively partitioning each graph into two in order to minimize edge cut. Each node of the index tree thus represents a subgraph of the original graph database. Leaves of the tree correspond to disk pages.

[4] It should be observed that, in the formalizations of the algorithms, the bodies of the **foreach**, **if**, **else**, and **else if** instructions are delimited using vertical bars on the left.

DOGMA has been shown to be able to increase the I/O-efficiency by exploiting data locality. Moreover, for every vertex, it stores the *internal partition distance* (IPD), i.e. the number of hops from the vertex to the nearest other vertex outside the disk page. Using IPD, a minimum distance between vertices can quickly be computed, and candidates can be pruned if their distance is higher than the distance between their respective query graph variables. For example, assume a partial substitution θ where $\tilde{v}$ is mapped to c_1 and $\tilde{u}$ to c_2 with $dist(\tilde{v}, \tilde{u}) = 4$ (where $dist(x, y)$ denotes the length of the shortest path between x and y), $IPD(c_1) = 2$ and $IPD(c_2) = 3$, and c_1 and c_2 reside on different disk pages. This partial substitution cannot create a complete substitution because $dist(\tilde{v}, \tilde{u}) < IPD(c_1) + IPD(c_2)$, so it can be pruned right away.

We make the following extension to DOGMA in order to support scoring queries. For each leaf node N, we store the edges that connect the node to other leaf nodes, along with aggregates (maximum, minimum, and average) of the properties of the vertices represented by N or adjacent to a vertex in N (called "boundary" vertices). As it will be clearer in the following, this additional information allows the ATK algorithm to compute upper bounds much more efficiently.

4.2. Overall structure of ATK

The ATK algorithm, given a scoring query $SQ = \langle S, F, \chi^g, SF^g \rangle$, works by recursively extending pairs of the form (Sub^{SQ}, θ), where $Sub^{SQ} \in USub^{SQ}$ and θ is a (partial) substitution for $unified(Sub^{SQ})$. The algorithm is based on two main operations: *block extension* and *variable extension*.

Definition 4.1 (*Block Extension, Variable Extension*)**.** Consider a pair (Sub^{SQ}, θ) where $Sub^{SQ} \in USub^{SQ}$ and θ is a (partial) substitution for $unified(Sub^{SQ})$. Suppose $SQB \in S \setminus Sub^{SQ}$ is a scoring query block such that $Sub^{SQ} \cup \{SQB\} \in USub^{SQ}$. The result of block extension is a pair $(Sub^{SQ} \cup \{SQB\}, \theta)$. Now assume θ is partial for $unified(Sub^{SQ})$. The result of variable extension is a pair (Sub^{SQ}, θ') where $dom(\theta') = dom(\theta) \cup \{\tilde{x}\}$ and $\tilde{x} \in \mathsf{VAR}_{unified(Sub^{SQ})} \setminus dom(\theta)$.

Intuitively, block extension adds a new scoring query block to Sub^{SQ} without changing θ. Variable extension maps a variable that is not already mapped by θ.

The ATK algorithm starts by calling extendVariable $(SQ, F, \theta_\emptyset)$ and proceeds by alternately performing block and variable extensions (Algorithm 2). For each kind of extension, ATK needs to choose which block to extend, or which variable and associated vertex in the graph database to match the variable. The algorithm continuously maintains the top-k answers computed so far along with their associated scores — this is set T in Algorithm 2. In order to prune, ATK compares an upper bound of the score of (Sub^{SQ}, θ) with the minimum score of the answers in T. The latter is updated whenever a new answer substitution is found.

The next few sections discuss the block and variable extension operations in detail.

4.3. Block extension

Block extension is implemented by the extendBlock function. The set EB contains scoring query blocks that can be used to extend (Sub^{SQ}, θ). When we extend (Sub^{SQ}, θ) to $(A = Sub^{SQ} \cup \{SQB\}, \theta)$, we have to consider three cases. Let us first assume $|T| = k$, which means T already stores k answers. In the first case, biggestUpperBound $(A, \theta) < \min_{(\cdot,\cdot,p) \in T} p$. This means that it is not possible to find *any valid superset* of Sub^{SQ} that can be used to extend θ whose score is greater than at least one substitution in T. In this case, we completely prune away this search path. In the second case, we have that upperBound $(A, \theta) < \min_{(\cdot,\cdot,p) \in T} p$, meaning that no substitution extending θ w.r.t. A has a score greater than at least one substitution in T. In this case, we further extend A by adding other blocks and without changing θ. This operation is done by recursively calling extendBlock (SQ, A, θ). If neither the first nor the second case above applies, or if $|T| < k$, we extend θ w.r.t. A by calling extendVariable (SQ, A, θ).

The upperBound function is used to compute an upper bound on the scores of the extensions of θ w.r.t. any given Sub^{SQ} as follows. To build the D and M vectors to be used as in Definition 3.10, we set $m_i = d_i = 0$ for $SQB_i \notin Sub^{SQ}$ and $d_i = 1$ for $SQB_i \in Sub^{SQ}$. When computing m_i for a $SQB_i \in Sub^{SQ}$, the upper bound is simply set to 1 if $AV_i \neq \emptyset$. Otherwise, we compute an upper bound for the extensions of θ by deriving an upper bound for each variable $\tilde{v}$'s property. If $\tilde{v} \in dom(\theta)$, then we take the exact value by looking at $\tilde{v}\theta$. For the candidate vertex set of a variable $\tilde{v} \notin dom(\theta)$,[5] we could in principle retrieve all their properties and take the maximum value — however, this would require a prohibitively expensive retrieval. We find good upper bounds more efficiently by using the maximum values stored by the ATK index along with disk pages. For a variable $\tilde{v} \notin dom(\theta)$, we look at its distance in $unified(Sub^{SQ})$ to all vertices $c \in dom(\theta)$. If $dist(c, \tilde{v}) < IPD(c)$ for some c, then we know that $\tilde{v}$ has to be mapped in the same disk page as c. We can then use the maximum values stored along with the disk pages where c resides. If multiple maximum values are possible, we take the minimum. If $dist(c, \tilde{v}) < IPD(c) + 1$, we do the same but use the maximum values of the disk page together with its boundary vertices. If instead $dist(c, \tilde{v}) > IPD(c) + 1$ for all c, we have no local information so we use global maximum values. We finally obtain the upper bound using these exact and maximum values.

Example 4.1. Suppose we have three query blocks $SQB_1 = \langle p \to \tilde{x}, SF_1, \emptyset \rangle$, $SQB_2 = \langle \tilde{x} \to \tilde{a} \to \tilde{b}, SF_2, \emptyset \rangle$ and $SQB_3 = \langle \tilde{x} \to \tilde{b}, SF_3, \emptyset \rangle$, and suppose $IPD(c) = 2$. Let $\theta = \{\tilde{x}/c\}$ be an answer substitution for SQB_1. Suppose we want to compute an upper bound for the score of extensions of θ w.r.t. $Sub^{SQ} = \{SQB_1, SQB_2\}$. We have that $dist(c, \tilde{a}) = 1$ and $dist(c, \tilde{b}) = 2$ in $unified(Sub^{SQ})$. In this case,

[5] We can get all candidates by checking the distance. For example, let c be a constant and $h = dist(c, \tilde{v})$. We retrieve all neighbors of c whose distance to c is less than h in the graph.

ALGORITHM 2: ATK block and variable extension functions

Data: T: global set of preferred answers as triples of the form $(\text{Sub}^{\text{SQ}}, \theta, p)$. R: global set of the already examined valid subsets of SQ

```
1  Function extendBlock (SQ, Sub^SQ, θ)
   Input: Scoring query SQ = ⟨S, F, χ^g, SF^g⟩, set Sub^SQ ∈ USub^SQ, part. subst. θ
2  EB ← {SQB | SQB ∈ S \ Sub^SQ, Sub^SQ ∪ {SQB} ∈ USub^SQ}
3  foreach SQB ∈ EB do
4      A ← Sub^SQ ∪ {SQB}
5      if A ∉ R then
6          if |T| = k and biggestUpperBound(A, θ) < min_{(·,·,p)∈T} p then
7              return
8          else if |T| = k and upperBound (A, θ) < min_{(·,·,p)∈T} p then
9              extendBlock(SQ, A, θ)
10         else
11             extendVariable(SQ, A, θ)
12         R ← R ∪ {A}

13 Function extendVariable (SQ, Sub^SQ, θ)
   Input: Scoring query SQ, set Sub^SQ ∈ USub^SQ, part. subst. θ
14 if dom(θ) = VAR_{unified(Sub^SQ)} then
15     updateT (Sub^SQ, θ, computeScoreAdvanced(Sub^SQ, θ))
16     if (Sub^SQ = F) then R ← ∅
17     extendBlock(SQ, Sub^SQ, θ)
18     return
19 NV ← {(c, ṽ) | c and ṽ are neighbors in Sub^SQ and ṽ ∉ dom(θ)}
20 NV ← NV ∪ {(θ(z̃), ṽ) | z̃, ṽ are neighbors in Sub^SQ, z̃ ∈ dom(θ), ṽ ∉ dom(θ)}
21 foreach (c, ṽ) ∈ NV do
22     R_{c,ṽ} ← getNeighborNum(c, edgeLabel(c, ṽ))
23     if R_{c,ṽ} = 0 then return
24 choose (c, ũ) ∈ argmax_{(c,ṽ)∈NV} R_{c,ṽ}
25 N_ũ ← getValidNeighbors(Sub^SQ, c, ũ)
26 if (|T| = k) then
27     N_ũ ← {m | m ∈ N_ũ and biggestUpperBound(Sub^SQ, θ ∪ {ũ/m}) > min_{(·,·,p)∈T} p}
28 Sort N_ũ by descending order of biggestUpperBound(Sub^SQ, θ ∪ {ũ/m})
29 foreach m ∈ N_ũ do
30     θ′ ← θ ∪ {ũ/m}
31     extendVariable(SQ, Sub^SQ, θ′)
```

$D = (1, 1, 0)$ and $M = (m_1, m_2, 0)$, where m_1 is the exact score computed from c's property values and m_2 is the maximum score computed from upper bounds on $\tilde{a}$' s and $\tilde{b}$' s property values. Since $dist(c, \tilde{a}) < IPD(c)$, we know that $\tilde{a}$ is mapped to the same disk page as c, so we can retrieve the maximum values of its properties from the index. Moreover, since $dist(c, \tilde{b}) < IPD(c) + 1$, variable $\tilde{b}$ is mapped to the same disk page plus its boundary vertices, so again we retrieve the upper bound directly from the index.

Function biggestUpperBound, given a valid subset Sub^{SQ} and a partial substitution θ, returns the biggest possible upper bound of θ, i.e. an upper bound that is greater than or equal to the upper bounds of any extension of θ w.r.t. Sub'^{SQ}, where $\text{Sub}^{\text{SQ}} \subset \text{Sub}'^{\text{SQ}} \subseteq S$. This computation is performed by defining a distance $dist'$ from c to $\tilde{v}$ for every valid subset $\text{Sub}^{\text{SQ}} \in \text{USub}^{\text{SQ}}$ as $dist'(c, \tilde{v}) = \max\{dist(c, \tilde{v}) \mid dist(c, \tilde{v})$ is computed w.r.t. $\text{unified}(\text{Sub}^{\text{SQ}})\}$ and using its values for computing m_i. Note that we can precompute $dist'(c, \tilde{v})$ between every pair of vertices/variables in $\text{unified}(\text{Sub}^{\text{SQ}})$. The unified query usually has relatively few vertices — even very complex unified queries will likely have 20–30 vertices, with 100 being a generous estimate of the maximum size. So the quadratic computation is not prohibitive.

The following result ensures that our computation is correct.

Proposition 4.1. *Given a scoring query* SQ, *a set* $\text{Sub}^{\text{SQ}} \in \text{USub}^{\text{SQ}}$, *and a substitution* θ *for* $\text{unified}(\text{Sub}^{\text{SQ}})$, $\text{biggestUpperBound}(\text{Sub}^{\text{SQ}}, \theta)$ *is greater than or equal to the upper bound of any extension of* θ *w.r.t. any* Sub'^{SQ} *such that* $\text{Sub}^{\text{SQ}} \subset \text{Sub}'^{\text{SQ}} \in \text{USub}^{\text{SQ}}$.

Proof. Let $D' = (d'_0, d'_1, \ldots)$ and $M' = (m'_0, m'_1, \ldots)$ be the two vectors computed by $\text{upperBound}(S, \theta)$ using $dist'$. Now assume $P^{\mathcal{G}}(D', M')$ is not the biggest upper bound, which means there exists a valid subset $\text{Sub}'^{\text{SQ}} \supset \text{Sub}^{\text{SQ}}$ such that an extension of θ w.r.t. Sub'^{SQ} has a higher score. Let $M = (m_0, m_1, \ldots)$ be the vector used to compute the upper bound for this extension of θ. We have that $m_i > m'_i$ for some i (since in this case $d'_i = 1$ for all i). However, this implies that there exist a vertex c and a variable $\tilde{v}$ such that $dist'(c, \tilde{v}) < dist(c, \tilde{v})$, which contradicts the definition of $dist'$.

Finally, function updateT adds triples of the form $(\text{Sub}^{\text{SQ}}, \theta, p)$ to T, also making sure that (i) T always contains k triples at most, and (ii) the triples in T are those with the higher values of p. To this aim, if $|T| = k$, it adds a triple $(\text{Sub}^{\text{SQ}}, \theta, p)$ to T only if $\min_{(\cdot,\cdot,p')\in T} p' < p$ — in this case, it also removes a triple with minimum p'.

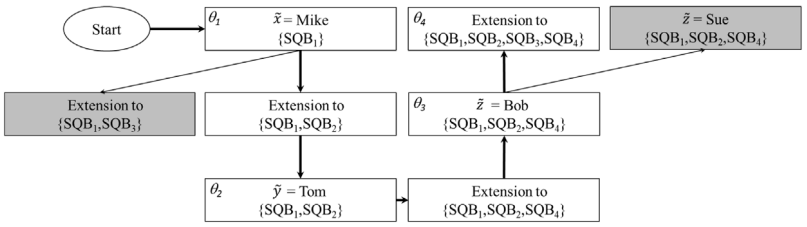


Fig. 3. Example run of the ATK algorithm. Gray boxes represent pruned search.

Table 2
Graph databases used in the experiments.

	$\|V\|$	$\|E\|$
CiteSeerX [47]	0.93M	2.9M
IMDb [48]	2.1M	7.7M
YouTube [49]	4.6M	14.9M
Flickr [50]	6.2M	15.2M

4.4. Variable extension

The extendVariable function is similar to findSubstitutions of the Base algorithm, but we add two key parts. First, we call extendBlock if we generate an answer substitution for unified(Sub^{SQ}) (line 17). Second, we compute a biggest upper bound when $\tilde{w}$ is mapped to m and prune the partial substitution if this is smaller than the minimum score in T (line 27) .

In order to compute a normalized score for a scoring query block SQB_i w.r.t. an answer substitution θ (function computeScore-Advanced) we construct a subquery $\mathrm{QB}_i\theta|_{\bar{\mathrm{AV}}}$ and find all its answer substitutions $\mathrm{AS}(\mathrm{QB}_i\theta|_{\bar{\mathrm{AV}}})$. While processing $\mathrm{QB}_i\theta|_{\bar{\mathrm{AV}}}$, we can ignore all constants that are not directly connected to any variables. By Definition 3.6, $\sum_{\theta' \in \mathrm{AS}(\mathrm{QB}_i\theta|_{\bar{\mathrm{AV}}})} \mathrm{SF}_i(\mathrm{QB}_i, \theta|_{\bar{\mathrm{AV}}}\theta')$ will be the denominator. This creates another overhead, but in general, the number of variables in $\mathrm{QB}_i\theta|_{\bar{\mathrm{AV}}}$ is small and we can reuse results once $\mathrm{AS}(\mathrm{QB}_i\theta|_{\bar{\mathrm{AV}}})$ is computed. In particular, many partial substitutions will be pruned during the depth-first search and only a small number of substitutions will be left to handle in line 15.

Example 4.2. In the case of our running example, ATK works as follows. It starts with the call extendVariable $(\mathrm{SQ}, \{P\mathrm{QB}_1\}, \theta_\emptyset)$, which is depicted as "Start" in Fig. 3. There are three candidates for variable $\tilde{x}$ of SQB_1, namely Mike, Tom, and Bob. Only Mike meets the constraint $\tilde{x}.postsAboutABC > 100$. $\theta_1 = \{\tilde{x}/Mike\}$ is a complete substitution for $\{\mathrm{SQB}_1\}$, so its score is computed and the top-1 is now θ_1. θ_1 is extended to $\{\mathrm{SQB}_1, \mathrm{SQB}_2\}$, and along the bold path, the following answer substitutions are found:

- $\theta_1 = \{\tilde{x}/Mike\}$ for $\{\mathrm{SQB}_1\}$;
- $\theta_2 = \{\tilde{x}/Mike, \tilde{y}/Tom\}$ for $\{\mathrm{SQB}_1, \mathrm{SQB}_2\}$;
- $\theta_3 = \{\tilde{x}/Mike, \tilde{y}/Tom, \tilde{z}/Bob\}$ for $\{\mathrm{SQB}_1, \mathrm{SQB}_2, \mathrm{SQB}_4\}$;
- $\theta_4 = \theta_3$ for $\{\mathrm{SQB}_1, \mathrm{SQB}_2, \mathrm{SQB}_3, \mathrm{SQB}_4\}$.

Thus, θ_4 is now the top-1 with a score of 46.7%. The depth-first search now backtracks to the mapping which sets $\tilde{z}$ to Sue, but its upper bound score of 46.6% is too low to beat θ_4 (because Sue's property value is too low). Thus, this mapping attempt is pruned. The extension of θ_1 to $\{\mathrm{SQB}_1, \mathrm{SQB}_3\}$ is also pruned because its upper bound score with $d_2 = 0$ and $d_4 = 0$ is less than the current top-1's score. Note that we only show the steps that find θ_4 and two pruning cases in Fig. 3 — all other depth-first search cases are omitted in the figure.

5. Experimental evaluation

5.1. Setting

We tested our framework using the 4 graph databases in Table 2. For each database, we generated 3000 scoring queries consisting of h query blocks each. We started by randomly retrieving h subgraphs with e edges each, ensuring that each subgraph partially overlapped with at least one of the others. We then selected a fixed portion of vertices (60% to 80%, depending on the query size and graph database used) from the set of retrieved ones and converted them to variables in all subgraphs. This ensured that the generated queries had at least one answer.

For generating constraints, we selected 30% of variables. For each selected variable $\tilde{v}$ that replaced a vertex u with property $u.p = z$, we wrote a constraint whose form is either $\tilde{v}.p \geq z$ or $\tilde{v}.p \leq z$ (if u had multiple numerical properties, we chose one randomly). This way, we also guaranteed that there existed at least one substitution for each query block (we actually obtained up to tens of millions of substitutions).

Table 3
p-values. p_1 is obtained using the paired t-test between Base and ATK1 for all query groups — p_2 compares ATK1 and ATK2 in the same way.

	p_1	p_2
CiteSeerX	3×10^{-3}	1.7×10^{-3}
IMDb	5.8×10^{-4}	2.5×10^{-3}
YouTube	1.7×10^{-4}	2.2×10^{-2}
Flickr	2×10^{-3}	7.19×10^{-9}

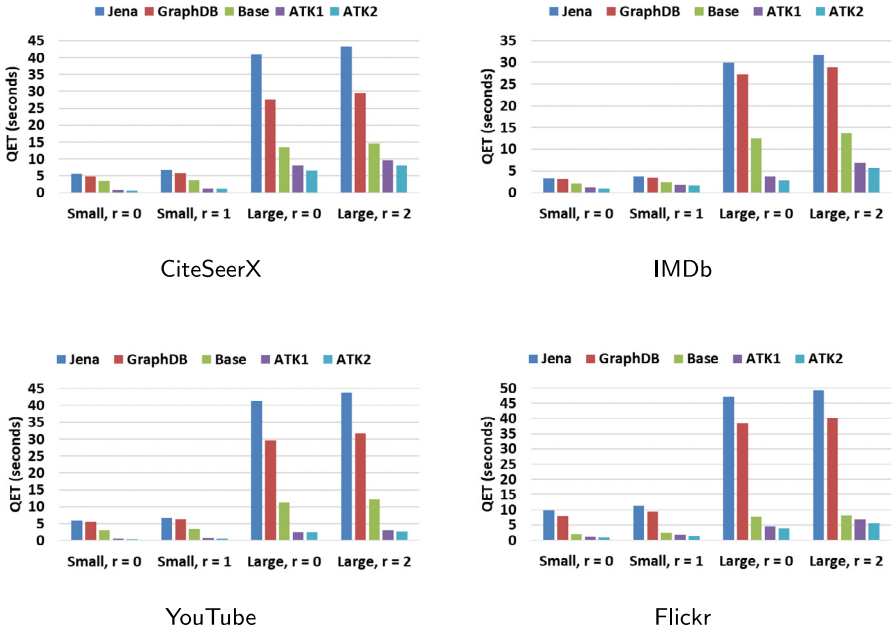


Fig. 4. Query evaluation times for different query groups and algorithms.

In r out of the h scoring query blocks, we used normalized preference scoring, i.e. we chose a set of $|\mathsf{VAR}_{\mathrm{QB}_i}|/2$ variables and designated them as AV_i. Then we defined:

$$\mathrm{SF}_i(\mathrm{SQB}_i, \theta) = \frac{1}{1+e^{-t}}, \text{ where } t = \begin{cases} \frac{\sum_{\bar{v} \in \mathsf{VAR}_{\mathrm{QB}_i}} \wp(\bar{v}\theta, property)}{1K}, & \text{if } \mathrm{AV}_i = \emptyset; \\ \frac{\sum_{\bar{v} \in \mathrm{AV}_i} \wp(\bar{v}\theta, property)}{1K}, & \text{otherwise.} \end{cases}$$

Moreover, we defined $\mathrm{SF}^{\mathcal{G}}(D, M) = \frac{1}{1+e^{-t}} \times \frac{\sum_i m_i}{|M|}$, where $t = \sum_i d_i$.

We generated two groups of "small" queries with $h = 2$, $e = 2$, and $r \in \{0, 1\}$, and two groups of "large" queries with $h = 5$, $e = 3$, and $r \in \{0, 2\}$. We compared the performance of the Base algorithm with the ATK algorithm and distinguished two variants of the latter — we call ATK1 the variant where we only prune partial substitutions, and ATK2 the one where we prune both partial substitutions and block extensions. This allowed us to assess whether the effort devoted to pruning block extensions along with partial substitutions pays off in terms of overall query evaluation performance.

Moreover, we compared ATK with two of the most popular RDF query engines, Jena [11] and GraphDB [51], used as function findSubstitutions in the Base algorithm.

5.2. Results

We started by measuring average query evaluation times (QET) for different query groups and algorithms, with $k = 5$. The results are reported in Fig. 4.

In general, the performance of the ATK algorithm is very satisfactory. ATK2 always outperforms ATK1 (with an average performance improvement around 15%) which confirms that pruning block extensions as well as partial substitutions is always beneficial. In comparison with Base, ATK2 saved 32% to 85% of query evaluation time for small queries, whith an average of 63%. For large queries, it saved 32% to 78% with an average of 59%. In the majority of cases, the performance advantage of ATK2 over Base when $r = 0$ is higher than the one obtained when $r > 0$ — for instance, with large queries over IMDb, the advantage is 78% when $r = 0$ and 58% when $r = 2$. This is natural because the components of the M vector used in function upperBound are simply set to 1 if AV is not empty. An interesting good result is that ATK2 still shows a good performance advantage over ATK1 when $r > 0$.

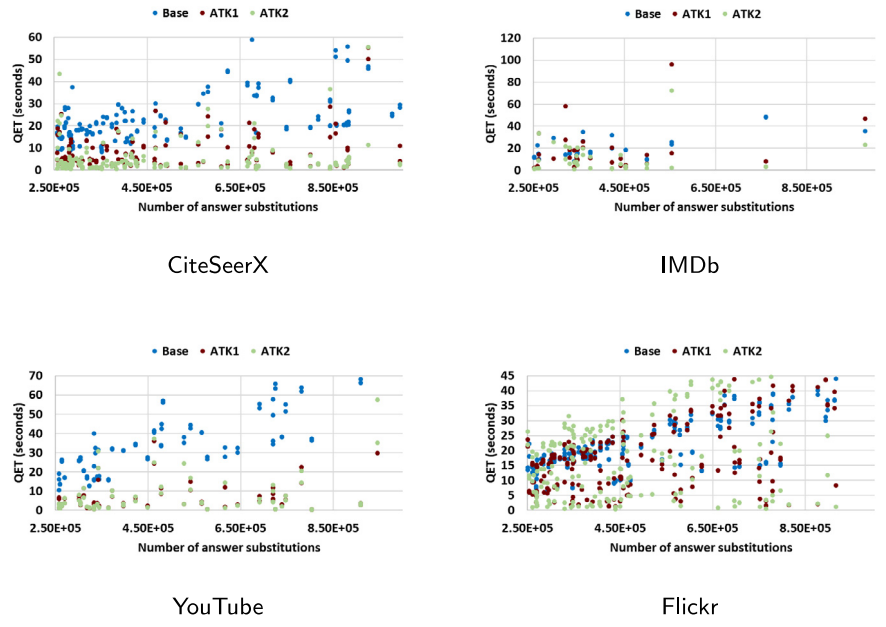


Fig. 5. Query evaluation times vs. number of answer substitutions.

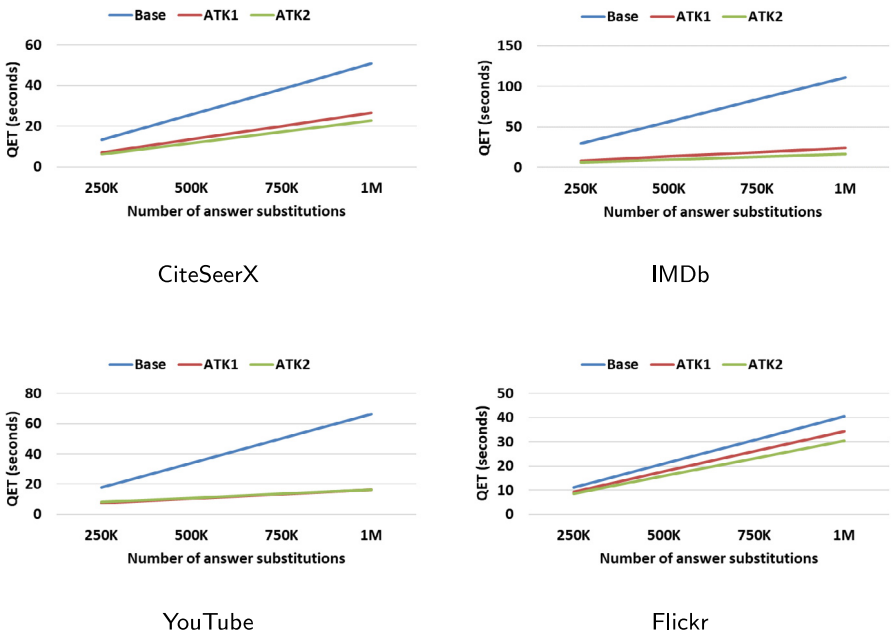


Fig. 6. Trend lines for query evaluation times vs. number of answer substitutions.

Jena and GraphDB need more time to answer queries than Base. As indicated in [4,44–46], traditional relational storage techniques such as triplestores have lower performance than graph database systems when facing these kinds of queries, due to inefficient table-joins.

Table 3 shows p-values that demonstrate that these results on query evaluation time reduction are statistically significant. The highest one is p_2 on YouTube, which is still lower than the usual significance threshold of 0.05. For all other cases, p-values are much lower than the threshold. This shows that all of our claims of improved efficiency via our two pruning methods are statistically significant as is the assertion that ATK2 is better than ATK1.

We also retrieved the number of answer substitutions computed for all queries with $k = 5$ and drew a scatter plot showing query evaluation times vs. number of answer substitutions (Fig. 5). Fig. 6 shows the linear trends (with least square error) of the data in

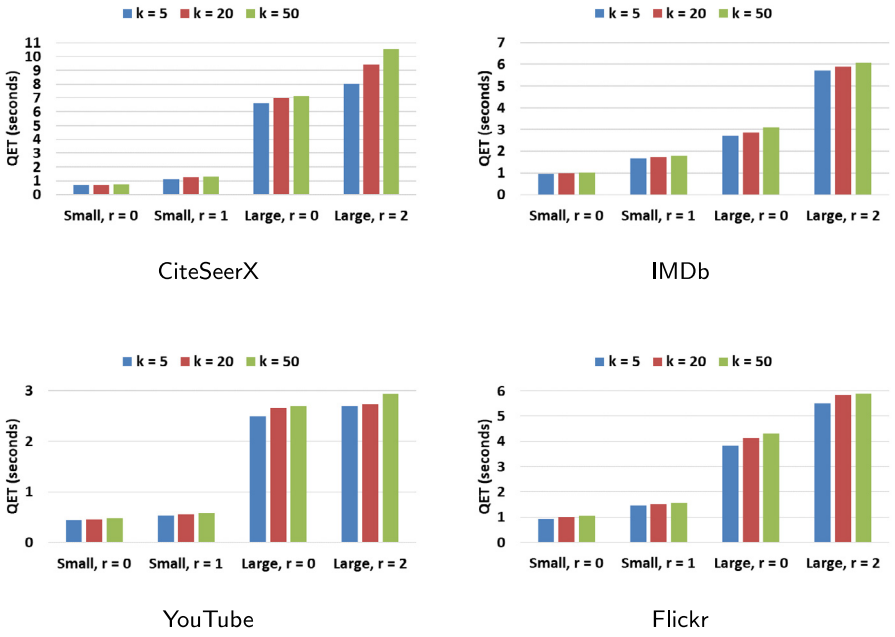


Fig. 7. Query evaluation times of ATK2 for different query groups and values of k.

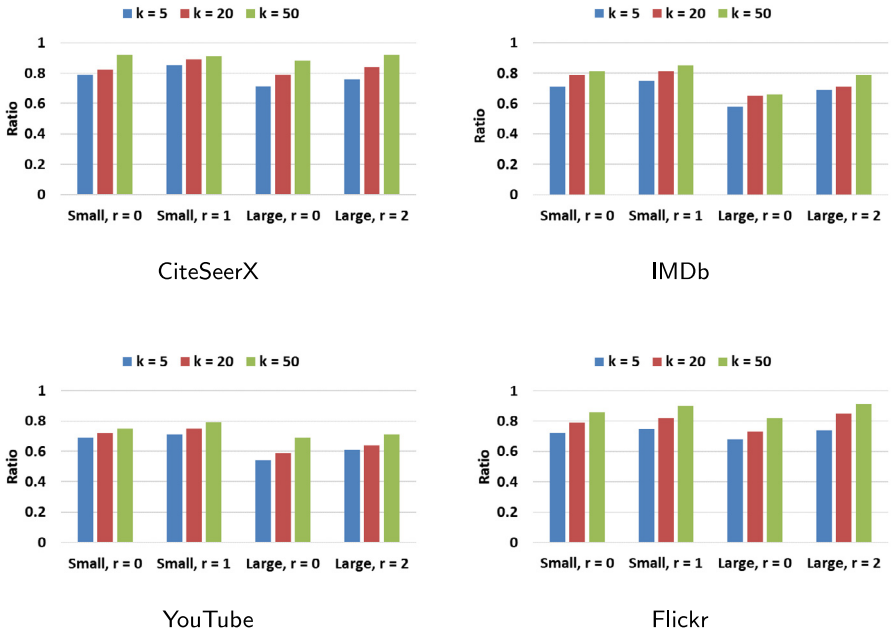


Fig. 8. Impact of pruning partial substitutions for different query groups and values of k.

Fig. 5. As expected, both ATK1 and ATK2 show better trends than Base, showing the best performance on the IMDb and YouTube datasets. Again, ATK2 outperforms ATK1.

We then assessed how the query evaluation times of ATK2 scale with the value of k. The results are shown in Fig. 7. The algorithm appears to scale gracefully — the increment in evaluation time is over 10% in just 5 out of 48 cases. On average, the increment was less than 6% when moving from $k = 5$ to $k = 20$ and less that 5% when moving from $k = 20$ to $k = 50$. Interestingly, the average increment was just 11.03% even when moving from $k = 5$ to $k = 50$.

Finally, we assessed how the pruning partial substitutions through function extendVariable actually affects the performance of ATK2 in the same setting of Fig. 7. Fig. 8 shows the ratio between the query evaluation times obtained by ATK2 when

applying function extendVariable and those obtained without such pruning. The result confirm the advantages of applying function extendVariable – the ratios range from 54% to 92%, with an average of 76%.

6. Conclusions

As stated in the Introduction, there are numerous real-world applications in a wide variety of areas where there is a need to identify certain subgraphs using a combination of three elements: (i) intrinsic properties of vertices, (ii) different subgraph patterns with different degrees of interest by the user, and (iii) scoring the "value" of discovering these subgraphs from the perspective of the user. We show that (i) and (ii) can be captured by a class of queries we define in the paper and that (iii) is captured via a user-defined scoring mechanism. The goal of our paper is to develop efficient algorithms to find the subgraphs in the database having the highest scores, while satisfying the requirements articulated in (i) and (ii). But because the scoring method is articulated by the users in their queries rather than assumed a priori, creating and efficiently implementing such scoring methods is a very complex task. We present the ATK algorithm that uses rapidly estimated upper bounds on the possible scores of retrieved subgraphs in order to prune. Our experiments show that ATK beats a baseline algorithm, even when built on top of the commercial graph database engines Jena and GraphDB, and scales well when dealing with real world graph data derived from YouTube, IMDb, Flickr, CiteSeerX. Even on graphs with more than 4M vertices and 15M edges, answering top-k queries only takes a few seconds. As our future work, we plan to develop variants of the ATK algorithm that are specifically designed for parallel/distributed computing infrastructures (e.g., along the lines followed in [27]). Such kinds of infrastructures pose specific important challenges, but we are confident that their use will allow us to support user-defined subgraph matching queries on even larger graphs.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

Some of the authors may have been partly funded by the POR grant J28C17000380006 "Start-(H)Open' funded by the Calabria Region Administration and by the PON-FESR grant "Catch 4.0"' funded by the Italian Ministry of Economic Development.

References

[1] I.A. Andrews, S. Kumar, F. Spezzano, V. Subrahmanian, Spinn: Suspicion prediction in nuclear networks, in: ISI, 2015, pp. 19–24.
[2] M. Joaristi, E. Serra, F. Spezzano, Inferring bad entities through the panama papers network, in: ASONAM, 2018, pp. 767–773.
[3] V. Subrahmanian, D.R. Recupero, AVA: adjective-verb-adverb combinations for sentiment analysis, IEEE Intell. Syst. 23 (4) (2008) 43–50.
[4] M. Bröcheler, A. Pugliese, V.S. Subrahmanian, DOGMA: A disk-oriented graph matching algorithm for RDF databases, in: ISWC, 2009, pp. 97–113.
[5] M.S. Martín, C. Gutierrez, Representing, querying and transforming social networks with RDF/SPARQL, in: ESWC, 2009, pp. 293–307.
[6] M. Huisman, M.A.V. Duijn, Software for social network analysis, in: Models and Methods in Social Network Analysis, 2005, pp. 270–316.
[7] W. De Nooy, A. Mrvar, V. Batagelj, Exploratory Social Network Analysis with Pajek, Cambridge University Press, 2005.
[8] H. MahmoudiNasab, S. Sakr, An experimental evaluation of relational rdf storage and querying techniques, in: DASFAA Workshops, 2010, pp. 215–226.
[9] J. Broekstra, A. Kampman, F. van Harmelen, Sesame: An architecture for storing and querying RDF data and schema information, in: Spinning the Semantic Web, 2003, pp. 197–222.
[10] M. Sintek, M. Kiesel, Rdfbroker: A signature-based high-performance RDF store, in: ESWC, 2006, pp. 363–377.
[11] Apache Software Foundation, Apache Jena, 2014, http://jena.apache.org.
[12] A. Kiryakov, D. Ognyanov, D. Manov, OWLIM - a pragmatic semantic repository for OWL, in: WISE Workshops, 2005, pp. 182–192.
[13] D.J. Abadi, A. Marcus, S. Madden, K.J. Hollenbach, Scalable semantic Web data management using vertical partitioning, in: VLDB, 2007, pp. 411–422.
[14] T. Neumann, G. Weikum, Scalable join processing on very large RDF graphs, in: SIGMOD, 2009, pp. 627–640.
[15] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, D. Reynolds, SPARQL basic graph pattern optimization using selectivity estimation, in: WWW, 2008, pp. 595–604.
[16] B. Kimelfeld, Y. Sagiv, Twig patterns: From XML trees to graphs, in: WebDB, 2006, pp. 26–31.
[17] J. Cheng, Y. Ke, W. Ng, Efficient query processing on graph databases, ACM Trans. Database Syst. 34 (1) (2009) 2:1–2:48.
[18] R. Giugno, D. Shasha, Graphgrep: A fast and universal method for querying graphs, in: ICPR, 2002, pp. 112–115.
[19] Y. Ke, J. Cheng, J.X. Yu, Querying large graph databases, in: DASFAA, 2010, pp. 487–488.
[20] S. Sakr, GraphREL: A decomposition-based and selectivity-aware relational framework for processing sub-graph queries, in: DASFAA, 2009, pp. 123–137.
[21] S. Zhang, S. Li, J. Yang, SUMMA: subgraph matching in massive graphs, in: CIKM, 2010, pp. 1285–1288.
[22] K. Zhu, Y. Zhang, X. Lin, G. Zhu, W.W. 0011, NOVA: A novel and efficient framework for finding subgraph isomorphism mappings in large graphs, in: DASFAA, 2010, pp. 140–154.
[23] L. Zou, L. Chen, M.T. Özsu, Distancejoin: Pattern match query in a large graph database, PVLDB 2 (1) (2009) 886–897.
[24] J. Cheng, J.X. Yu, B. Ding, P.S. Yu, H. Wang, Fast graph pattern matching, in: ICDE, 2008, pp. 913–922.
[25] S. Zhang, S. Li, J. Yang, GADDI: distance index based subgraph matching in biological networks, in: EDBT, 2009, pp. 192–203.
[26] R. Di Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, D. Shasha, SING: Subgraph search in non-homogeneous graphs, BMC Bioinform. 11 (2010) 96.
[27] M. Bröcheler, A. Pugliese, V.S. Subrahmanian, COSI: cloud oriented subgraph identification in massive social networks, in: ASONAM, 2010, pp. 248–255.
[28] M. Bröcheler, A. Pugliese, V.S. Subrahmanian, A budget-based algorithm for efficient subgraph matching on huge networks, in: ICDE Workshops, 2011, pp. 94–99.
[29] F. Katsarou, N. Ntarmos, P. Triantafillou, Performance and scalability of indexed subgraph query processing methods, PVLDB 8 (12) (2015) 1566–1577.
[30] N. Nilsson, Principles of artificial intelligence, Tioga, Palo Alto CA, 1980.
[31] A. Sanfeliu, K. Fu, A distance measure between attributed relational graphs for pattern recognition, IEEE Trans. Syst. Man Cybern. 13 (3) (1983) 353–362.

[32] X. Yan, P.S. Yu, J. Han, Substructure similarity search in graph databases, in: SIGMOD, 2005, pp. 766–777.
[33] Y. Tian, R.C. McEachin, C. Santos, D.J. States, J.M. Patel, SAGA: a subgraph matching tool for biological graphs, Bioinformatics 23 (2) (2007) 232–239.
[34] Y. Tian, J.M. Patel, TALE: A tool for approximate large graph matching, in: ICDE, 2008, pp. 963–972.
[35] R. Myers, R.C. Wilson, E.R. Hancock, Bayesian graph edit distance, IEEE Trans. Pattern Anal. Mach. Intell. 22 (6) (2000) 628–635.
[36] R.C. Wilson, E.R. Hancock, Bayesian compatibility model for graph matching, Pattern Recognit. Lett. 17 (1996) 263–276.
[37] M. Bröcheler, A. Pugliese, V.S. Subrahmanian, Probabilistic subgraph matching on huge social networks, in: ASONAM, 2011, pp. 271–278.
[38] S. Magliacane, A. Bozzon, E.D. Valle, Efficient execution of top-k SPARQL queries, in: ISWC, 2012, pp. 344–360.
[39] Y. Qi, K.S. Candan, M.L. Sapino, Sum-max monotonic ranked joins for evaluating top-k twig queries on weighted data graphs, in: VLDB, 2007, pp. 507–518.
[40] J. Cheng, X. Zeng, J.X. Yu, Top-k graph pattern matching over large graphs, in: ICDE, 2013, pp. 1033–1044.
[41] Y. Zhu, L. Qin, J.X. Yu, H. Cheng, Finding top-k similar graphs in graph databases, in: EDBT, 2012, pp. 456–467.
[42] X. Yan, B. He, F. Zhu, J. Han, Top-k aggregation queries over large networks, in: ICDE, 2010, pp. 377–380.
[43] W. Fan, X. Wang, Y. Wu, Diversified top-k graph pattern matching, PVLDB 6 (13) (2013) 1510–1521.
[44] M. Arenas, S. Conca, J. Pérez, Counting.beyond.a. yottabyte, or.how. SPARQL, Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard, in: WWW, 2012, pp. 629–638.
[45] J. Cheng, J.X. Yu, A survey of relational approaches for graph pattern matching over large graphs, in: Graph Data Management: Techniques and Applications, 2011, pp. 112–141.
[46] S. Sakr, G. Al-Naymat, Relational processing of rdf queries: a survey, SIGMOD Rec. 38 (4) (2009) 23–28.
[47] Citeseerx, public dataset, 2011, http://csxstatic.ist.psu.edu/about/data.
[48] Imdb, imdb dataset, 2012, http://www.imdb.com/interfaces.
[49] X. Cheng, C. Dale, J. Liu, Statistics and social network of youtube videos, in: IWQoS, 2008, pp. 229–238.
[50] M. Cha, A. Mislove, P.K. Gummadi, A measurement-driven analysis of information propagation in the flickr social network, in: WWW, 2009, pp. 721–730.
[51] Ontotext, graphdb, 2014, http://www.ontotext.com.

Noseong Park received his Ph.D. degree in Computer Science from the University of Maryland, College Park in 2016. He was an Assistant Professor at the University of North Carolina, Charlotte and at George Mason University. Currently, he is an Assistant Professor in the department of artificial intelligence at Yonsei University, South Korea. His main research interests include graph data management, data mining, and machine learning.

Andrea Pugliese received his Ph.D. degree in Computer and Systems Engineering from the University of Calabria, Italy in 2005. He was Visiting Professor with the University of Maryland Institute for Advanced Computer Studies, USA. Currently, he is an Associate Professor in the DIMES Department at University of Calabria. His main research interests include graph data management, indexing techniques, activity detection, and computer security. He is a member of the ACM.

Edoardo Serra received his Ph.D. degree in Computer Engineering from the University of Calabria, Italy, in 2012. After his Ph.D., he was Research Associate at University of Maryland (till Aug 2015). From Aug 2015, he is Assistant Professor in the computer science department at Boise State University (BSU). His research interests are in the field of Data Science with applications in cybersecurity and national security.

V.S. Subrahmanian is the Dartmouth College Distinguished Professor in Cybersecurity, Technology, and Society and Director of the Institute for Security, Technology, and Society at Dartmouth. He previously served as a Professor of Computer Science at the University of Maryland from 1989–2017 where he created and headed both the Lab for Computational Cultural Dynamics and the Center for Digital International Governmen. He also served for 6+ years as Director of the University of Maryland's Institute for Advanced Computer Studies. Prof. Subrahmanian is an expert on big data analytics including methods to analyze text/geospatial/relational/social network data, learn behavioral models from the data, forecast actions, and influence behaviors with applications to cybersecurity and counter-terrorism. He has written five books, edited ten, and published over 300 refereed articles. He is a Fellow of the American Association for the Advancement of Science and the Association for the Advancement of Artificial Intelligence and received numerous other honors and awards. His work has been featured in numerous outlets such as the Baltimore Sun, the Economist, Science, Nature, the Washington Post, American Public Media. He serves on the editorial boards of numerous journals including Science, the Board of Directors of the Development Gateway Foundation (set up by the World Bank), SentiMetrix, Inc., and on the Research Advisory Board of Tata Consultancy Services. He previously served on DARPA's Executive Advisory Council on Advanced Logistics and as an ad-hoc member of the US Air Force Science Advisory Board.