THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Making Pattern Queries Bounded in Big Graphs

Citation for published version:
Cao, Y, Fan, W, Huai, J & Huang, R 2015, Making Pattern Queries Bounded in Big Graphs. in *2015 IEEE 31st International Conference on Data Engineering.* Institute of Electrical and Electronics Engineers, pp. 161-172, 31st International Conference on Data Engineering, Seoul, Korea, Republic of, 13/04/15. https://doi.org/10.1109/ICDE.2015.7113281

Digital Object Identifier (DOI):
10.1109/ICDE.2015.7113281

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
2015 IEEE 31st International Conference on Data Engineering

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 17. Feb. 2025

Making Pattern Queries Bounded in Big Graphs

Yang Cao[1,2] Wenfei Fan[2,1] Jinpeng Huai[1] Ruizhe Huang[2]

[1]*RCBD and SKLSDE Lab, Beihang University,* [2]*University of Edinburgh*

***Abstract*—It is cost-prohibitive to find matches $Q(G)$ of a pattern query Q in a big graph G. We approach this by fetching a small subgraph G_Q of G such that $Q(G_Q) = Q(G)$. We show that many practical patterns are *effectively bounded* under access constraints $\mathcal{A}$ commonly found in real life, such that G_Q can be identified in time determined by Q and $\mathcal{A}$ only, independent of the size $|G|$ of G. This holds no matter whether pattern queries are localized (*e.g.,* via subgraph isomorphism) or non-localized (graph simulation). We provide algorithms to decide whether a pattern Q is effectively bounded, and if so, to generate a query plan that computes $Q(G)$ by accessing G_Q, in time independent of $|G|$. When Q is not effectively bounded, we give an algorithm to extend access constraints and make Q bounded in G. Using real-life data, we experimentally verify the effectiveness of the approach, *e.g.,* about 60% of queries are effectively bounded for subgraph isomorphism, and for such queries our approach outperforms the conventional methods by 4 orders of magnitude.**

I. Introduction

Given a pattern query Q and a graph G, *graph pattern matching* is to find the set $Q(G)$ of matches of Q in G. It is used in, *e.g.,* social marketing, knowledge discovery, mobile network analysis, intelligence analysis for identifying terrorist organizations [25], and the study of adolescent drug use [17].

When G is big, graph pattern matching is cost-prohibitive. Facebook has 1.26 billion nodes and 140 billion links in its social graph, about 300PB of user data [28]. When the size $|G|$ of G is 1PB, a linear scan of G takes 1.9 days using SSD with scanning speed of 6GB/s. Worse still, graph pattern matching is intractable if it is defined with subgraph isomorphism [31], and it takes $O((|V|+|V_Q|)(|E|+|E_Q|))$-time if we use graph simulation [20], where $|G| = |V|+|E|$ and $|Q| = |V_Q|+|E_Q|$.

Can we still efficiently compute exact answers $Q(G)$ when G is big while we have constrained resources, such as a single processor? We approach this by *making big graphs small*, capitalizing on a set $\mathcal{A}$ of access constraints, which are a combination of indices and simple cardinality constraints defined on the labels of neighboring nodes of G. We determine whether Q is *effectively bounded* under $\mathcal{A}$, *i.e.,* for *all* graphs G that satisfy $\mathcal{A}$, there exists a subgraph $G_Q \subset G$ such that

(a) $Q(G_Q) = Q(G)$, and
(b) the size $|G_Q|$ of G_Q and the time for identifying G_Q are both determined by $\mathcal{A}$ and Q only, *independent of* $|G|$.

If Q is effectively bounded, we can generate a query plan that for all G satisfying $\mathcal{A}$, computes $Q(G)$ by accessing (visiting and fetching) a small G_Q in time *independent of* $|G|$, no matter how big G is. Otherwise, we will identify extra access constraints on an input G and make Q bounded in G.

A large number of real-life queries are effectively bounded under simple access constraints, as illustrated below.

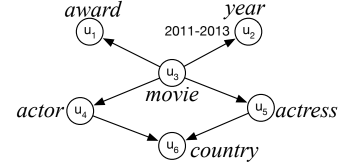


Fig. 1. Pattern query Q_0 on IMDb

Example 1: Consider IMDb [22], a graph G_0 in which nodes represent movies, casts, and awards from 1880 to 2014, and edges denote various relationships between the nodes. An example search on IMDb is to *find pairs of first-billed actor and actress (main characters) from the same country who co-stared in a award-winning film released in 2011-2013.*

The search can be represented as a pattern query Q_0 shown in Fig. 1. Graph pattern matching here is to find the set $Q_0(G_0)$ of matches, *i.e.,* subgraphs G' of G_0 that are isomorphic to Q_0; we then extract and return actor-actress pairs from each match G'. The challenge is that G_0 is large: the IMDb graph has 5.1 million nodes and 19.5 million edges. Add to this that subgraph isomorphism is NP-complete.

Not all is lost. Using simple aggregate queries one can readily find the following *real-life* cardinality constraints on the movie dataset from 1880–2014: (1) in each year, every award is presented to no more than 4 movies (C1); (2) each movie has at most 30 first-billed actors and actresses (C2), and each person has only one country of origin (C3); and (3) there are no more than 135 years (C4, *i.e.,* 1880-2014), 24 major movie awards (C5) and 196 countries (C6) in total [22]. An index can be built on the labels and nodes of G_0 for each of the constraints, yielding a set $\mathcal{A}_0$ of 8 access constraints.

Under $\mathcal{A}_0$, pattern Q_0 is *effectively bounded*. We can find $Q_0(G_0)$ by accessing *at most* 17923 nodes and 35136 edges in G_0, *regardless of* the size of G_0, by the following query plan:

(a) identify a set V_1 of 135 year nodes, 24 award nodes and 196 country nodes, by using the indices for constraints C4-C6;

(b) fetch a set V_2 of at most $24 \times 3 \times 4 = 288$ award-winning movies released in 2011–2013, with no more than $288 \times 2 = 576$ edges connecting movies to awards and years, by using those award and year nodes in V_1 and the index for C1;

(c) fetch a set V_3 of at most $(30+30)*288 = 17280$ actors and actresses with 17280 edges, using V_2 and the index for C2;

(d) connect the actors and actresses in V_3 to country nodes in V_1, with at most 17280 edges by using the index for C3. Output (actor, actress) pairs connected to the same country in V_1.

The query plan visits at most 135 + 24 + 196 + 288 + 17280 = 17923 nodes, and 576 + 17280 + 17280 = 35136

edges, using the cardinality constraints and indices in $\mathcal{A}_0$, as opposed to tens of millions of nodes and edges in IMDb. □

This example tells us that graph pattern matching is feasible in big graphs within constrained resources, by making use of effectively bounded pattern queries. To develop a practical approach out of the idea, several questions have to be answered. (1) Given a pattern query Q and a set $\mathcal{A}$ of access constraints, can we determine whether Q is effectively bounded under $\mathcal{A}$? (2) If Q is effectively bounded, how can we generate a query plan to compute $Q(G)$ in big G by accessing a bounded G_Q? (3) If Q is not bounded, can we make it "bounded" in G by adding simple extra constraints? (4) Does the approach work on both localized queries (*e.g.,* via subgraph isomorphism) and non-localized queries (via graph simulation)?

Contributions. This paper aims to answer these questions for graph pattern matching. The main results are as follows.

(1) We introduce effective boundedness for graph pattern queries (Section II). We formulate access constraints on graphs, and define effectively bounded pattern queries. We also show how to find simple access constraints from real-life data.

(2) We characterize effectively bounded *subgraph queries* Q, *i.e.,* patterns defined by subgraph isomorphism (Section III). We identify a *sufficient and necessary* condition to decide whether Q is effectively bounded under a set $\mathcal{A}$ of access constraints. Using the condition, we develop a decision algorithm in $O(|\mathcal{A}||E_Q|+||\mathcal{A}|||V_Q|^2)$ time, where $|Q| = |V_Q|+|E_Q|$, and $||\mathcal{A}||$ is the number of constraints in $\mathcal{A}$. The cost is *independent of* big graph G, and query Q is typically small in practice.

(3) We provide an algorithm to generate query plans for effectively bounded subgraph queries (Section IV). After Q is found effectively bounded under $\mathcal{A}$, the algorithm generates a query plan that, given a graph G that satisfies $\mathcal{A}$, accesses a subgraph G_Q of size independent of $|G|$, in $O(|V_Q||E_Q||\mathcal{A}|)$ time. Moreover, we show that the plan is *worst-case-optimal*, *i.e.,* for each input Q and $\mathcal{A}$, the *largest* G_Q it finds from all graphs G that satisfy $\mathcal{A}$ is the *minimum* among all worst-case G_Q identified by all other query plans.

(4) If Q is not bounded under $\mathcal{A}$, we make it *instance-bounded* (Section V). That is, for a given graph G that satisfies $\mathcal{A}$, we find an extension $\mathcal{A}_M$ of $\mathcal{A}$ such that under $\mathcal{A}_M$, we can find $G_Q \subset G$ in time decided by $\mathcal{A}_M$ and Q, and $Q(G_Q) = Q(G)$. We show that when the size of indices in $\mathcal{A}_M$ is predefined, the problem for deciding the existence of $\mathcal{A}_M$ is in low polynomial time (PTIME), but it is $\log$-APX-hard to find a minimum $\mathcal{A}_M$. When $\mathcal{A}_M$ is unbounded, all query loads can be made instance-bounded by adding simple access constraints.

(5) We extend the study to *simulation queries*, *i.e.,* patterns interpreted by graph simulation (Section VI). It is more challenging to cope with the *non-localized* and *recursive* nature of simulation queries. Nonetheless, we provide a characterization of effectively bounded simulation queries. We also show that our algorithms for checking effective boundedness, generating query plans, and for making queries instance-bounded can be adapted to simulation queries, with the same complexity.

(6) We experimentally evaluate our algorithms using real-life data (Section VII). We find that our approach is effective for both localized and non-localized queries: (a) on graphs G of billions of nodes and edges [1], our query plans outperform the conventional methods that computes $Q(G)$ directly by *4 and 3 orders of magnitude* on average, for subgraph and simulation queries, respectively, accessing at most *0.0032%* of the data in G; (b) 60% (resp. 33%) of subgraph (resp. simulation) queries are effectively bounded under simple access constraints; and (c) *all queries* can be made instance-bounded in G by extending constraints and accessing 0.016% of extra data in G; and 95% become instance-bounded by accessing at most 0.009% extra data. Our algorithms are efficient: they take at most 37ms to decide whether Q is effectively bounded and to generate an optimal query plan for all Q and constraints tested.

This work is the first effort to study effectively bounded graph queries, from fundamental problems to practical algorithms. It suggests an approach to querying graphs: (1) given a query Q, we check whether Q is effectively bounded under a set $\mathcal{A}$ of access constraints; (2) if so, we generate a query plan that given a graph G satisfying $\mathcal{A}$, computes $Q(G)$ by accessing G_Q of size independent of $|G|$, no matter how big G grows; (3) if not, we make Q instance-bounded in G with extra simple constraints. The approach works for *both* localized subgraph queries *and* non-localized simulation queries.

Given the prohibitive cost of querying big graphs, this approach helps even when only *limited queries* are effectively bounded. In fact, we find that many queries on real-life datasets are actually effectively bounded under very simple access constraints. Moreover, when a finite set of queries is not effectively bounded, we can make them instance-bounded.

All proofs of the results of the paper can be found in [3].

Related Work. We categorize related works as follows.

<u>*Effective boundedness.*</u> The study of effective boundedness traces back to scale independence. The latter was proposed [5] to approximately answer relational aggregate queries under certain conditions, for key/value stores. It aims to guarantee that a bounded amount of work is required to execute all queries in an application, regardless of the size of the underlying data. The idea was formalized in [12], along with a notion of access constraints for relational queries. Recently, the notion of [12] is revised in [10] by requiring that the amount of data accessed (*i.e.,* G_Q) can be identified in time determined by query Q and access constraints $\mathcal{A}$ only, referred to as *effective boundedness*; it is characterized for SPC queries [10].

This work differs from the previous work in the following. (1) We introduce access constraints on graph data, to specify cardinality constraints on the labels of neighboring nodes, and guide us to retrieve small subgraphs G_Q. (2) Under such constraints, we formalize and characterize the effective boundedness of graph patterns, an issue harder than its counterpart for relational queries [10], [12]. (3) We propose instance boundedness for queries that are not effectively bounded.

<u>*Resource-bounded and anytime algorithms.*</u> Related are also resource-bounded [16] and anytime algorithms [32]. The former study reachability queries and *personalized* pattern queries, in which some pattern nodes are designated to match

fixed nodes in a graph G. It is to compute approximate answers by accessing no more than $\alpha|G|$ nodes and edges in G, for $\alpha \in (0, 1)$ [16]. Anytime algorithms [32] allow users either to specify a budget on resources (*e.g.,* running time; known as contract algorithms [33]), or to terminate the run of the algorithms at any time and get intermediate answers (known as interruptible algorithms [19]). Contract anytime algorithms have been explored for (a) budgeted search such as bounded-cost planning [4], [29], [30], [32] under a user-specified budget; and (b) graph search via subgraph isomorphism, to find intermediate approximate answers within the budget, either by assigning dynamically maintained budgets and costs to nodes during the traversal [8], or by deciding search orders based on the frequencies of certain features in queries and graphs [27].

This work differs from the prior work as follows. (1) We aim to compute *exact answers* for pattern queries in big graphs, as opposed to heuristic answers that may not have a provable accuracy bound. (2) We *characterize* what pattern queries can be answered exactly within a cost independent of the size of big graph, based on access constraints; in contrast, the prior work does not study under what budget accurate answers are warranted by using the semantics of the data. (3) We study general pattern queries, which may be *either localized or non-localized*, and may *not be* personalized [16].

Graph indexing and compression. There are typically two ways to reduce search space. (1) Graph indexing uses pre-computed global information of G to compute distance [11], shortest paths [18] or substructure matching [26]. (2) Graph compression computes a summary G_c of a big graph G and uses G_c to answer all queries posed on G [7], [13], [24].

In contrast to the prior work, (1) we compute *exact answers* rather than heuristic. (2) Instead of using *the same* graph G_c to answer *all queries* posed on G, we adopt a *dynamic reduction scheme* that finds a subgraph G_Q of G for each query Q. Since G_Q consists of only the information needed for answering Q, it allows us to compute $Q(G)$ by using G_Q much smaller than G_c and hence, much less resources. (3) When Q is effectively bounded, for *all* graphs G we can find G_Q of size *independent of* $|G|$; in contrast, $|G_c|$ may be proportional to $|G|$.

Making big graphs small. There have been other techniques for reducing a big graph into small ones, *e.g.,* distribute query answering [23], pattern matching using views [15], and incremental pattern matching [14]. These are complementary to this work and can be readily combined with ours, *e.g.,* our methods can be readily adapted to distributed settings.

II. Effectively Bounded Graph Pattern Queries

In this section we define access schema on graphs and effectively bounded graph pattern queries. We start with a review of graphs and patterns. Assume an alphabet Σ of labels.

Graphs. A data graph is a node-labeled directed graph $G = (V, E, f, \nu)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges, in which (v, v') denotes the edge from v to v'; (3) $f()$ is a function such that for each node v in V, $f(v)$ is a label in Σ, *e.g.,* year; and (4) $\nu(v)$ is the attribute value of $f(v)$, *e.g.,* year = 2011.

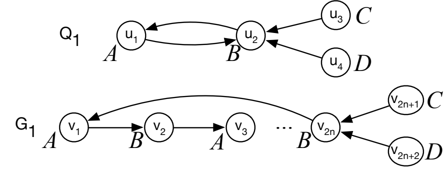


Fig. 2. Pattern query Q_1 and data graph G_1

We write G as (V, E) or (V, E, f) when it is clear from the context. The *size* of G, denoted by $|G|$, is defined to be the total number of nodes and edges in G, *i.e.,* $|G| = |V| + |E|$.

Remark. To simplify the discussion, we do not explicitly define edge labels. Nonetheless, our techniques can be readily adapted to edge labels: for each labeled edge e, we can insert a "dummy" node to represent e, carrying e's label.

Labeled set. For a set $S \subseteq \Sigma$ of labels, we say that $V_S \subseteq V$ is a *S-labeled set* of G if (a) $|V_S| = |S|$ and (b) for each label l_S in S, there exists a node v in V_S such that $f(v) = l_S$. In particular, when $S = \emptyset$, the S-labeled set in G is $\emptyset$.

Common neighbors. A node v is called a *neighbor* of another node v' in G if either (v, v') or (v', v) is an edge in G. We say that v is a *common neighbor* of a set V_S of nodes in G if for all nodes v' in V_S, v is a neighbor of v'. In particular, when V_S is $\emptyset$, all nodes of G are common neighbors of V_S.

Subgraphs. Graph $G_s = (V_s, E_s, f_s, \nu_s)$ is a *subgraph* of G if $V_s \subseteq V$, $E_s \subseteq E$, and for each $(v, v') \in E_s$, $v \in V_s$ and $v' \in V_s$, and for each $v \in V_s$, $f_s(v) = f(v)$ and $\nu_s(v) = \nu(v)$.

Pattern queries. A pattern query Q is a directed graph (V_Q, E_Q, f_Q, g_Q), where (1) V_Q, E_Q and f_Q are analogous to their counterparts in data graphs; and (2) for each node u in V_Q, $g_Q(u)$ is the *predicate* of u, defined as a conjunction of atomic formulas of the form $f_Q(u)$ op c, where c is a constant, and op is one of $=, >, <, \leq$ and $\geq$. For instance, in pattern Q_0 of Fig. 1, $g_Q(\text{year}) = \text{year} \geq 2011 \wedge \text{year} \leq 2013$. We simply write Q as (V_Q, E_Q) or (V_Q, E_Q, f_Q).

We consider two semantics of graph pattern matching.

Subgraph queries. A match of Q in G via *subgraph isomorphism* [31] is a subgraph $G'(V', E', f')$ of G that is isomorphic to Q, *i.e.,* there exists a *bijective function* h from V_Q to V' such that (a) (u, u') is in E_Q if and only if $(h(u), h(u')) \in E'$, and (b) for each $u \in V_Q$, $f_Q(u) = f'(h(u))$ and $g_Q(\nu(h(u)))$ evaluates to *true*, where $g_Q(\nu(h(u)))$ substitutes $\nu(h(u))$ for $f_Q(u)$ in $g_Q(u)$. Here $Q(G)$ is the set of all matches of Q in G.

Simulation queries. A match of Q in G via *graph simulation* [20] is a binary match relation $R \subseteq V_Q \times V$ such that (a) for each $(u, v) \in R$, $f_Q(u) = f(v)$ and $g_Q(\nu(v))$ evaluates to *true*, where $g_Q(\nu(v))$ substitutes $\nu(v)$ for $f_Q(u)$ in $g_Q(u)$; (b) for each node u in V_Q, there exists a node v in V such that (i) $(u, v) \in R$, and (ii) for any edge (u, u') in Q, there exists an edge (v, v') in G such that $(u', v') \in R$.

For any Q and G, there exists a *unique maximum* match relation R_M via graph simulation (possibly empty) [20]. Here $Q(G)$ is defined to be R_M. Simulation queries are widely used in social community analysis and social marketing [9].

Data locality. A query Q is *localized* if for any graph G that matches Q, any node u and neighbor u' of u in Q, and for any match v of u in G, there must exist a match v' of u' in G such that v' is a neighbor of v in G. Subgraph queries are localized. In contrast, simulation queries are *non-localized*.

Example 2: Consider a simulation query Q_1 and graph G_1 shown in Fig. 2, where G_1 matches Q_1. Then Q_1 is not localized: u_2 matches $v_2, \ldots, v_{2n-2}$ and v_{2n}, but for all $k \in [2, n]$, v_{2k-2} has no neighbor in G that matches the neighbor u_3 of u_2 in Q. To decide whether u_2 matches v_2, we have to inspect all the nodes on an unbounded cycle in G_1. □

We will study effective boundedness for subgraph queries in Sections III–V, and then extend the results to non-localized simulation queries in Section VI. To formalize effectively bounded patterns, we first define access constraints on graphs.

Access schema on graphs. An *access schema* $\mathcal{A}$ is a set of *access constraints* of the following form:

$$S \rightarrow (l, N),$$

where $S \subseteq \Sigma$ is a (possibly empty) set of labels, l is a label in Σ, and N is a natural number.

A graph $G(V, E, f)$ *satisfies* the access constraint if

- for any S-labeled set V_S of nodes in V, there exist at most N common neighbors of V_S with label l; and
- there exists an *index on S for l* such that for any S-labeled set V_S in G, it finds all common neighbors of V_S labeled with l in $O(N)$-time, independent of $|G|$.

We say that G *satisfies* access schema $\mathcal{A}$, denoted by $G \models \mathcal{A}$, if G satisfies all the access constraints in $\mathcal{A}$.

An access constraint is a combination of (a) *a cardinality constraint* and (b) *an index* on the labels of neighboring nodes. It tells us that for any S-node labeled set V_S, there exist a bounded number of common neighbors V_l labeled with l and moreover, V_l can be efficiently retrieved with the index.

Two special *types* of access constraints are as follows:

(1) $|S| = 0$ (*i.e.,* $\emptyset \rightarrow (l, N)$): for any G that satisfies the constraint, there exist at most N nodes in G labeled l; and

(2) $|S| = 1$ (*i.e.,* $l \rightarrow (l', N)$): for any G that satisfies the access constraint and for each node v labeled with l in G, at most N neighbors of v are labeled with l'.

Intuitively, constraints of type (1) are global cardinality constraints on all nodes labeled l, and those of type (2) state cardinality constraints on l'-neighbors of each l-labeled node.

Example 3: Constraints C1-C6 on IMDb given in Example 1 can be expressed as access constraints φ_i (for $i \in [1, 6]$):

φ_1: $(\mathsf{year}, \mathsf{award}) \rightarrow (\mathsf{movie}, 4)$; φ_4: $\emptyset \rightarrow (\mathsf{year}, 135)$;
φ_2: $\mathsf{movie} \rightarrow (\mathsf{actors/actress}, 30)$; φ_5: $\emptyset \rightarrow (\mathsf{award}, 24)$;
φ_3: $\mathsf{actor/actress} \rightarrow (\mathsf{country}, 1)$; φ_6: $\emptyset \rightarrow (\mathsf{country}, 196)$.

Here φ_2 denotes a pair $\mathsf{movie} \rightarrow (\mathsf{actors}, 30)$ and $\mathsf{movie} \rightarrow (\mathsf{actress}, 30)$ of access constraints; similarly for φ_3. Note that $\varphi_4 - \varphi_6$ are constraints of type (1); $\varphi_2 - \varphi_3$ are of type (2); and φ_1 has the general form: for any pair of year and award nodes, there are at most 4 movie nodes connected to both, *i.e.,* an award is given to at most 4 movies each year. We use $\mathcal{A}_0$ to denote the set of these access constraints. □

Effectively bounded patterns. A pattern query Q is *effectively bounded under* an access schema $\mathcal{A}$ if for *all* graphs G that satisfy $\mathcal{A}$, there exists a subgraph G_Q of G such that

(a) $Q(G_Q) = Q(G)$; and
(b) G_Q can be identified in time that is determined by Q and $\mathcal{A}$ only, *not by* $|G|$.

By (b), $|G_Q|$ is also *independent of* the size $|G|$ of G. Intuitively, Q is effectively bounded under $\mathcal{A}$ if for all graphs G that satisfy $\mathcal{A}$, $Q(G)$ can be computed by accessing a bounded G_Q rather than the entire G, and moreover, G_Q can be efficiently accessed by using access constraints of $\mathcal{A}$.

For instance, as shown in Example 1, query Q_0 is effectively bounded under the access schema $\mathcal{A}_0$ of Example 3.

Discovering access constraints. From experiments with real-life data we find that many practical queries are effectively bounded under simple access constraints $S \rightarrow (l, N)$ when $|S|$ is at most 3. We discover access constraints as follows.

(1) Degree bounds: if each node with label l has degree at most N, then for any label l', $l \rightarrow (l', N)$ is an access constraint.

(2) Constraints of type (1): such global constraints are quite common, *e.g.,* φ_6 on IMDb: $\emptyset \rightarrow (\mathsf{country}, 196)$.

(3) Functional dependencies (FDs): our familiar FDs $X \rightarrow A$ are access constraints of the form $X \rightarrow (A, 1)$, *e.g.,* $\mathsf{movie} \rightarrow \mathsf{year}$ is an access constraint of type (2): $\mathsf{movie} \rightarrow (\mathsf{year}, 1)$. Such constraints can be discovered by shredding a graph into relations and then using available FD discovery tools.

(4) Aggregate queries: such queries allow us to discover the semantics of the data, *e.g.,* grouping by (year, country, genre) we find $(\mathsf{year}, \mathsf{country}, \mathsf{genre}) \rightarrow (\mathsf{movie}, 1800)$, *i.e.,* each country releases at most 1800 movies per year in each genre.

Maintaining access constraints. The indices in an access schema can be incrementally and *locally* maintained in response to changes to the underlying graph G. It suffices to inspect $\Delta G \cup \mathsf{Nb}_G(\Delta G)$, where ΔG is the set of nodes and edges deleted or inserted, and $\mathsf{Nb}_G(\Delta G)$ is the set of neighbors of those nodes in ΔG, regardless of how big G is.

III. Effective Boundedness of Subgraph Queries

To make practical use of effective boundedness, we first answer the following question, denoted by $\mathsf{EBnd}(\mathsf{Q}, \mathcal{A})$:

- Input: A pattern query $Q(V_Q, E_Q)$, an access schema $\mathcal{A}$.
- Question: Is Q effectively bounded under $\mathcal{A}$?

We start with subgraph queries. The good news is that

(a) there exists a sufficient and necessary condition, *i.e.,* a *characterization*, for deciding whether a subgraph query Q is effectively bounded under $\mathcal{A}$; and better still,
(b) $\mathsf{EBnd}(\mathsf{Q}, \mathcal{A})$ is decidable in low polynomial time in the size of Q and $\mathcal{A}$, independent of any data graph.

We prove these results in the rest of the section.

A. Characterizing the Effective Boundedness

The effective boundedness of subgraph queries is characterized in terms of a notion of *coverage*, given as follows.

The *node cover* of $\mathcal{A}$ on Q, denoted by $\mathsf{VCov}(Q, \mathcal{A})$, is the set of nodes in Q computed inductively as follows:

(a) if $\emptyset \to (l, N)$ is in $\mathcal{A}$, then for each node u in Q with label l, $u \in \mathsf{VCov}(Q, \mathcal{A})$; and
(b) if $S \to (l, N)$ is in $\mathcal{A}$, then for each S-labeled set V_S in Q, if $V_S \subseteq \mathsf{VCov}(Q, \mathcal{A})$, then all common neighbors of V_S in Q that are labeled with l are also in $\mathsf{VCov}(Q, \mathcal{A})$.

Intuitively, a node u is covered by $\mathcal{A}$ if in any graph G satisfying $\mathcal{A}$, there exist a bounded number of *candidate matches* of u, and the candidates can be retrieved by using indices in $\mathcal{A}$. Obviously, (a) u is covered if its candidates are bounded by type (1) constraints. (b) If for some $\varphi = S \to (l, N)$ in $\mathcal{A}$, u is labeled with l and is a common neighbor of V_S that is covered by $\mathcal{A}$, then u is covered by $\mathcal{A}$, since its candidates are bounded (by N and the bounds on candidate matches of V_S), and can be retrieved by using the index of φ.

The *edge cover* of $\mathcal{A}$ on Q, denoted by $\mathsf{ECov}(Q, \mathcal{A})$, is the set of edges in Q defined as follows: (u_1, u_2) is in $\mathsf{ECov}(Q, \mathcal{A})$ if and only if there exist an access constraint $S \to (l, N)$ in $\mathcal{A}$ and a S-labeled set V_S in Q such that (1) u_1 (resp. u_2) is in V_S and $V_S \subseteq \mathsf{VCov}(Q, \mathcal{A})$ and (2) $f_Q(u_2) = l$ (resp. $f_Q(u_1) = l$).

Intuitively, (u_1, u_2) is in $\mathsf{ECov}(Q, \mathcal{A})$ if one of u_1 and u_2 is covered by $\mathcal{A}$ and the other has a bounded number of candidate matches by $S \to (l, N)$. Thus, we can verify their matches in a graph G by accessing a bounded number of edges.

Note that $\mathsf{VCov}(Q, \mathcal{A}) \subseteq V_Q$ and $\mathsf{ECov}(Q, \mathcal{A}) \subseteq E_Q$.

The node and edge covers characterize effectively bounded subgraph queries (see [3] for a proof, which uses three lemmas and the data locality of subgraph queries).

Theorem 1: *A subgraph query Q is effectively bounded under an access schema $\mathcal{A}$ if and only if (iff) $\mathsf{VCov}(Q, \mathcal{A}) = V_Q$ and $\mathsf{ECov}(Q, \mathcal{A}) = E_Q$.* □

Example 4: For query $Q_0(V_0, E_0)$ of Fig. 1 and access schema $\mathcal{A}_0$ of Example 3, one can verify that $\mathsf{VCov}(Q_0, \mathcal{A}_0) = V_0$ and $\mathsf{ECov}(Q_0, \mathcal{A}_0) = E_0$. From this and Theorem 1 it follows that Q_0 is effectively bounded under $\mathcal{A}_0$. □

B. Checking Effectively Bounded Subgraph Queries

Capitalizing on the characterization, we show that whether Q is effectively bounded under $\mathcal{A}$ can be efficiently decided.

Theorem 2: *For subgraph queries Q, $\mathsf{EBnd}(Q, \mathcal{A})$ is in*

(1) $O(|\mathcal{A}||E_Q| + ||\mathcal{A}|||V_Q|^2)$ time in general; and
(2) $O(|\mathcal{A}||E_Q| + |V_Q|^2)$ time when either
- *for each node in Q, its parents have distinct labels; or*
- *all access constraints in $\mathcal{A}$ are of type (1) or (2).* □

Algorithm EBChk

Input: A subgraph query Q and an access schema $\mathcal{A}$.
Output: "yes" if Q is effectively bounded and "no" otherwise.

1. **for each** $S \to (l, N)$ in $\mathcal{A}$ ($S \neq \emptyset$) **do**
2.   find all $\bar{V}_S^u \mapsto (u, N)$ in Q and add them to Γ; /*$f(u) = l$*/
3. $\mathcal{B} := \{v \in V_Q \mid \emptyset \to (f_Q(v), N) \text{ is in } \mathcal{A}\}$;
4. $\mathcal{C} := \mathcal{B}$; /*Initialize $\mathsf{VCov}(Q, \mathcal{A})$*/
5. $\mathsf{InitAuxi}(L, ct)$; /*Initialize auxiliary structures*/
6. **while** $\mathcal{B}$ is not empty **do**
7. $v = \mathcal{B}.\mathsf{pop}()$;
8.   **for each** ϕ in $L[v]$ **do**
9. Update $(ct[\phi])$; /*Update counter $ct[\phi]$*/
10.   **if** $ct[\phi] = \emptyset$ and $u \notin \mathcal{C}$ **do** /*suppose ϕ: $\bar{V}_S^u \mapsto (u, N)$*/
11. $\mathcal{B} := \mathcal{B} \cup \{u\}$; $\mathcal{C} := \mathcal{C} \cup \{u\}$;
12. **if** $V_Q \subseteq \mathcal{C}$ and all edges in Q are in $\mathsf{ECov}(Q, \mathcal{A})$ **then**
13.   **return** "yes";
14. **return** "no";

Fig. 3. Algorithm EBChk

Here $|\mathcal{A}|$ denotes the total length of access constraints in $\mathcal{A}$, $||\mathcal{A}||$ is the number of constraints in $\mathcal{A}$, and a node u' is a *parent* of u in Q if there exists an edge from u' to u in Q.

Algorithm. We prove Theorem 2 by providing a checking algorithm. The algorithm is denoted by EBChk and shown in Fig. 3. Given a subgraph query $Q(V_Q, E_Q)$ and an access schema $\mathcal{A}$, it checks whether (a) $V_Q \subseteq \mathsf{VCov}(Q, \mathcal{A})$ and (b) $E_Q \subseteq \mathsf{ECov}(Q, \mathcal{A})$; it returns "yes" if so, by Theorem 1.

To check these conditions, we actualize $\mathcal{A}$ on Q: for each $S \to (l, N)$ in $\mathcal{A}$ ($S \neq \emptyset$), and each node u in Q with $f_Q(u) = l$, the *actualized constraint* is $\bar{V}_S^u \mapsto (u, N)$, where $\bar{V}_S^u$ is the maximum set of neighbors of u in Q such that (a) there exists a S-labeled set $V_S \subseteq \bar{V}_S^u$ and (b) for each u' in $\bar{V}_S^u$, $f_Q(u') \in S$.

Actualized constraints help us deduce $\mathsf{VCov}(Q, \mathcal{A})$: a node u of Q is in $\mathsf{VCov}(Q, \mathcal{A})$ if and only if either

- there exists $\emptyset \to (l, N)$ in $\mathcal{A}$ and $f_Q(u) = l$; or
- $\bar{V}_S^u \mapsto (u, N)$ and there exists a S-labeled set of Q that is a subset of $\bar{V}_S^u \cap \mathsf{VCov}(Q, \mathcal{A})$.

When $\mathsf{VCov}(Q, \mathcal{A})$ is in place, we can easily check whether $E_Q \subseteq \mathsf{ECov}(Q, \mathcal{A})$ by definition and using the actualized constraints, without explicitly computing $\mathsf{ECov}(Q, \mathcal{A})$.

We next present the details of algorithm EBChk.

Auxiliary structures. EBChk uses three auxiliary structures.

(1) It maintains a set $\mathcal{B}$ of nodes in Q that are in $\mathsf{VCov}(Q, \mathcal{A})$ but it remains to be checked whether other nodes can be deduced from them. Initially, $\mathcal{B}$ includes nodes whose labels are covered by type (1) constraints in $\mathcal{A}$ (line 3). EBChk uses $\mathcal{B}$ to control the **while** loop (lines 5-10): it terminates when $\mathcal{B} = \emptyset$, *i.e.,* all candidates for $\mathsf{VCov}(Q, \mathcal{A})$ are found.

(2) For each node v, EBChk uses an inverted index $L[v]$ to store all actualized constraints $\bar{V}_S^u \mapsto (u, N)$ such that $v \in \bar{V}_S^u$. That is, $L[v]$ indexes these constraintsthat can be used on v.

(3) For each actualized constraint $\phi = \bar{V}_S^u \mapsto (u, N)$, EBChk maintains a set $ct[\phi]$ to keep track of those labels of S that are *not covered* by nodes in $\bar{V}_S^u \cap \mathsf{VCov}(Q, \mathcal{A})$ yet. Initially, $ct[\phi] = S$. When $ct[\phi]$ is empty, EBChk concludes that there

is a S-labeled subset of $\bar{V}_S^u$ covered by $\mathsf{VCov}(Q, \mathcal{A})$, and thus deduces that u should also be in $\mathsf{VCov}(Q, \mathcal{A})$ (line 10).

Using these, EBChk works in the following two steps.

(1) Computing Γ. It finds all actualized constraints of $\mathcal{A}$ on Q and puts them in Γ (lines 1-2). This can be done by scanning all nodes of Q and their neighbors for each access constraint in $\mathcal{A}$. Observe that there are at most $||\mathcal{A}|||V_Q|$ actualized constraints in Γ, *i.e.,* Γ is bounded by $O(||\mathcal{A}|||E_Q|)$.

(2) Computing $\mathsf{VCov}(Q, \mathcal{A})$, stored in a variable $\mathcal{C}$. After initializing auxiliary structures as described above via procedure InitAuxi (omitted; lines 3-5), EBChk processes nodes in $\mathcal{B}$ one by one (lines 6-11). For each $u \in \mathcal{B}$ and each actualized constraint $\phi = \bar{V}_S^v \mapsto (v, N)$ in $L[u]$, it updates the set $ct[\phi]$ by removing label $f_Q(u)$ by procedure Update (omitted; line 9). When $ct[\phi] = \emptyset$, *i.e.,* there exists a S-labeled subset in $\bar{V}_S^v$ that is covered by $\mathcal{C}$, EBChk adds u to $\mathcal{C}$ and $\mathcal{B}$ (lines 10-11). When $\mathcal{B}$ is empty, *i.e.,* all nodes have been inspected, EBChk checks whether $V_Q \subseteq \mathsf{VCov}(Q, \mathcal{A})$ and whether all edges are covered by $\mathsf{ECov}(Q, \mathcal{A})$. It returns "yes" if so (lines 12-13).

Example 5: Given subgraph query Q_0 of Fig. 1 and access schema $\mathcal{A}_0$ of Example 3, EBChk first computes the set Γ of actualized constraints: $\phi_1 = (u_1, u_2) \mapsto (u_3, 4)$, $\phi_2 = u_3 \mapsto (u_4/u_5, 30)$, and $\phi_3 = u_4/u_5 \mapsto (u_6, 1)$. It then sets both $\mathcal{B}$ and $\mathcal{C}$ to be $\{u_1, u_2, u_6\}$, and initializes $ct[\phi_1], \ldots, ct[\phi_3]$ and lists $L[u_1], \ldots, L[u_6]$ accordingly. EBChk then pops u_1 and u_2 off from $\mathcal{B}$ and finds that u_3 can be deduced. Thus it adds u_3 to $\mathcal{B}$ and $\mathcal{C}$. It then pops u_3 off from $\mathcal{B}$, processes u_4 and u_5, and confirms that u_4 and u_5 should be included in $\mathcal{C}$. At this point, it finds that $\mathcal{C}$ contains all the nodes in Q and moreover, each edge in Q is also covered by at least one access constraint in $\mathcal{A}_0$. Thus it returns "yes". □

Correctness & Complexity. The correctness of EBChk follows from Theorem 1 and the properties of actualized constraints stated above. We next analyze the time complexity of EBChk.

(1) General case. Observe the following. (a) Computing Γ is in $O(|\mathcal{A}||E_Q|)$ time, since for each φ in $\mathcal{A}$, we can find all actualized constraints of φ in $O(\Sigma_{v \in V_Q} \mathsf{deg}(v)|\varphi|) = O(|\varphi||E_Q|)$ time, where $\mathsf{deg}(v)$ is the number of neighbors of v. (b) Computing $\mathsf{VCov}(Q, \mathcal{A})$ takes $O(||\mathcal{A}|||V_Q|^2)$ time. For each φ in $\mathcal{A}$, the sets $ct(\phi)$ for all corresponding actualized constraints ϕ in Γ are updated in time $O(\Sigma_{v \in V_Q}(\mathsf{deg}(v)^2)) = O(|V_Q|^2)$. As each ϕ in Γ is processed once, the total time is bounded by $O(||\mathcal{A}|||V_Q|^2)$. (c) The checking of lines 12-13 takes $O(|\mathcal{A}||E_Q| + |V_Q|^2)$ time. Thus, EBChk takes $O(|\mathcal{A}||E_Q| + ||\mathcal{A}|||V_Q|^2 + |V_Q|^2) = O(|\mathcal{A}||E_Q| + ||\mathcal{A}|||V_Q|^2)$ time.

(2) Special cases. We next show that EBChk can be optimized to $O(|\mathcal{A}||E_Q| + |V_Q|^2)$ time for each of the two special cases given in Theorem 2. The idea is to use a counter $n[\phi]$ instead of $ct[\phi]$ in EBChk such that $n[\phi]$ always equals $|ct[\phi]|$. This does not hurt the correctness since in the special cases, each time when we update $ct[\phi]$, we remove a distinct label. With this new auxiliary structure, step (b) in the analysis above is in $O(||\mathcal{A}|||E_Q|)$ time in total since the counters are updated $O(||\mathcal{A}||(\Sigma_{v \in V_Q} \mathsf{deg}(v))) = O(||\mathcal{A}|||E_Q|)$ times in total, and each updates takes $O(1)$ time: it just decreases $n[\phi]$ by 1.

IV. Generating Query Plans

After a subgraph query $Q(V_Q, E_Q)$ is found effectively bounded under an access schema $\mathcal{A}$, we need to generate a "good" query plan for Q that, given any (big) graph G, computes $Q(G)$ by fetching a small G_Q such that $Q(G) = Q(G_Q)$ and $|G_Q|$ is determined by Q and $\mathcal{A}$, independent of $|G|$.

The main results of this section are as follows:

- a notion of worst-case optimality for query plans; and
- an algorithm to generate worst-case-optimal query plans in $O(|V_Q||E_Q||\mathcal{A}|)$ time.

Below we first formalize query plans and define the worst-case optimality. We then present the algorithm.

Query plans. A *query plan* $\mathcal{P}$ for Q under $\mathcal{A}$ is a sequence of *node fetching* operations of the form $\mathsf{ft}(u, V_S, \varphi, g_Q(u))$, where u is a l-labeled node in Q, V_S denotes a S-labeled set of Q, φ is a constraint $\varphi = S \to (l, N)$ in $\mathcal{A}$, and $g_Q(u)$ is the predicate of u (refer to Section II for the definition).

On a graph G, the operation is to retrieve a set $\mathsf{cmat}(u)$ of *candidate matches* for u from G: given V_S that was retrieved from G earlier, it fetches common neighbors of V_S from G that (i) are labeled with l and (ii) satisfy the predicate $g_Q(u)$ of u. These nodes are fetched by using the index of φ and are stored in $\mathsf{cmat}(u)$. In particular, when $S = \emptyset$, the operation fetches all l-labeled nodes in G as $\mathsf{cmat}(u)$ for u.

The operations $\mathsf{ft}_1\mathsf{ft}_2 \cdots \mathsf{ft}_n$ in $\mathcal{P}$ are executed one by one, in this order. There may be multiple operations for the same node u in Q, each fetching a set V_i^u of candidates for u from G. We will ensure that for ft_i and ft_j for u, V_j^u has less nodes than V_i^u if $i < j$, and we say that ft_j *reduces* $\mathsf{cmat}(u)$ fetched by ft_i. We denote V_k^u by V_u, where ft_k is the last operation for u in $\mathcal{P}$, *i.e.,* it fetches the smallest $\mathsf{cmat}(u)$ for u.

Building G_Q. Intuitively, $\mathcal{P}$ tells us what nodes to retrieve from G. From the data fetched by $\mathcal{P}$, a subgraph $G_Q(V_\mathcal{P}, E_\mathcal{P})$ is built and used to compute $Q(G)$. More specifically, (a) $V_\mathcal{P} = \bigcup_{u \in Q} V_u$, *i.e.,* it contains maximally reduced $\mathsf{cmat}(u)$ for each node u in Q; and (b) $E_\mathcal{P}$ consists of the following: for each node pairs (v, v') in $V_u \times V_{u'}$, if (u, u') is an edge in Q, we check whether (v, v') is an edge in G and if so, include it in $E_\mathcal{P}$. This is done by accessing a bounded amount of data: we first find $\varphi_{u'} = S \to (f_Q(u'), N)$ in $\mathcal{A}$ and a S-labeled set V_s such that $v \in V_S$; we then fetch common neighbors of V_S by using the index of $\varphi_{u'}$ and check whether v' is one of them. As Q is effectively bounded under $\mathcal{A}$ (*i.e.,* $\mathsf{ECov}(Q, A) = E_Q$), if (v, v') is an edge in G then such $\varphi_{u'}$ and V_S exist.

Bounded plans. We say that a query plan $\mathcal{P}$ for Q under $\mathcal{A}$ is *effectively bounded* if for all $G \models \mathcal{A}$, it builds a subgraph G_Q of G such that (a) $Q(G_Q) = Q(G)$ and (b) the time for fetching data from G by *all operations* in $\mathcal{P}$ depends on $\mathcal{A}$ and Q only. That is, $\mathcal{P}$ fetches a bounded amount of data from G and builds G_Q from it. By (b), $|G_Q|$ is *independent of* $|G|$.

Optimality. We naturally want an *optimal* plan $\mathcal{P}$ that finds us a minimum G_Q, *i.e.,* for each graph $G \models \mathcal{A}$, G_Q identified by $\mathcal{P}$ has the smallest size among all subgraphs identified by any effectively bounded query plans. Unfortunately, the result below shows that this is impossible (see [3] for a proof).

Algorithm QPlan

Input: An effectively bounded subgraph query Q, access schema $\mathcal{A}$.
Output: A worst-case optimal and effectively bounded query plan $\mathcal{P}$.

1. Build actualized graph $Q_\Gamma(V_\Gamma, E_\Gamma)$ from Q and Γ;
2. **for each** u in V_Γ **do**
3. $\quad$ $\mathsf{size}[u] := +\infty$; $\mathsf{sn}[u] := \mathit{false}$;
4. $\quad$ **if** there exists $\varphi = \emptyset \to (l, N)$ in $\mathcal{A}$ with $f_Q(u) = l$ **do**
5. $\quad\quad$ append $\mathsf{ft}(u, \mathit{nil}, \varphi, g_Q(u))$ to $\mathcal{P}$;
6. $\quad\quad$ $\mathsf{sn}[u] := \mathit{true}$; $\mathsf{size}[u] := N$;
7. **while** there exists u in V_Γ such that $\mathsf{check}(u) = \mathit{true}$ **do**
8. $\quad$ $(V_u, \varphi_u, \mathsf{size}[u], \mathsf{sn}[u]) := \mathsf{ocheck}(u)$;
9. $\quad$ append $\mathsf{ft}(u, V_u, \varphi_u, g_Q(u))$ to $\mathcal{P}$;
10. **return** $\mathcal{P}$;

Fig. 4. Algorithm QPlan

Theorem 3: *There exists no query plan that is both effectively bounded and optimal for all graphs $G \models \mathcal{A}$.* □

This motivates us to introduce worst-case optimality. An effectively-bounded query plan $\mathcal{P}$ for Q under $\mathcal{A}$ is *worst-case optimal* if for any other effectively bounded query plan $\mathcal{P}'$ for Q under $\mathcal{A}$, $\max_{G \models \mathcal{A}} |G_Q| \leq \max_{G \models \mathcal{A}} |G'_Q|$, where G_Q and G'_Q are subgraphs identified by $\mathcal{P}$ and $\mathcal{P}'$, respectively.

That is, given any Q and $\mathcal{A}$, for *all* $G \models \mathcal{A}$, the *largest* subgraph G_Q identified by $\mathcal{P}$ is no larger than the worst-case subgraphs identified by any other effectively bounded query plans.

Worst-case optimal query plans are within reach in practice.

Theorem 4: *There exists an algorithm that, given any effectively bounded subgraph query Q under an access schema $\mathcal{A}$, finds a query plan that is both effectively bounded and worst-case optimal for Q under $\mathcal{A}$, in $O(|V_Q||E_Q||\mathcal{A}|)$ time.* □

Algorithm. We prove Theorem 4 by giving such an algorithm, denoted by QPlan and shown in Fig. 4. The algorithm inspects each node u of Q, finds an access constraint φ in $\mathcal{A}$ such that its index can help us retrieve candidates $\mathsf{cmat}(u)$ for u from an input graph G, generates a fetching operation accordingly, and stores it in a list $\mathcal{P}$. It then iteratively reduces $\mathsf{cmat}(u)$ for each u in Q to optimize $\mathcal{P}$, until $\mathcal{P}$ cannot be further improved.

The algorithm uses the following structures.

(1) An *actualized graph* $Q_\Gamma(V_\Gamma, E_\Gamma)$, which is a directed graph constructed from Q and the set Γ of all actualized constraints of $\mathcal{A}$ on Q (see Section III). Here (a) $V_\Gamma = V_Q$; and (b) for any two nodes u_1 and u_2 in V_Γ, (u_1, u_2) is in E_Γ iff there exists a constraint $\bar{V}_S \mapsto (u_2, N)$ in Γ such that u_1 is in $\bar{V}_S$. Intuitively, Q_Γ represents deduction relations for nodes in V_Q, and guides us to extract candidate matches for Q.

(2) For each node u in Q, a counter $\mathsf{size}[u]$ to store the cardinality of $\mathsf{cmat}(u)$, and a Boolean flag $\mathsf{sn}[u]$ to indicate whether the fetching operations in current $\mathcal{P}$ can find $\mathsf{cmat}(u)$.

With these structures, algorithm QPlan works as follows. It first builds actualized graph Q_Γ (line 1), and initializes $\mathsf{size}[u] = +\infty$ and $\mathsf{sn}[u] = \mathit{false}$ for all the nodes u in Q_Γ (lines 2-3). It then finds nodes u_0 for which $\mathsf{cmat}(u)$ can be retrieved by using the index specified in some type (1) constraints $\emptyset \to (l, N)$ in $\mathcal{A}$ (lines 4-6). For each u_0, QPlan adds a fetching operation to $\mathcal{P}$ and sets $\mathsf{sn}[u_0] = \mathit{true}$ and $\mathsf{size}[u_0] = N$.

After the initialization, QPlan recursively processes nodes u of Q to retrieve or reduce their $\mathsf{cmat}(u)$ (lines 7-9), starting from those nodes u_0 identified in line 4. It picks the next node u by a function check (omitted). Here $\mathsf{check}(u)$ does the following: it (i) finds the set V^p_u of parents of u in Q_Γ such that $\mathsf{sn}[v] = \mathit{true}$ for all $v \in V^p_u$, (ii) selects a subset V_u of V^p_u such that V_u forms a S-labeled set for some constraint $\varphi_u = S \to (f_Q(u), N)$ in $\mathcal{A}$, and moreover, $N * \Pi_{v \in V_u} \mathsf{size}[v]$ is minimum among all such S-labeled sets of u; and (iii) returns *true* if $N * \Pi_{v \in V_u} \mathsf{size}[v] < \mathsf{size}[u]$. If $\mathsf{check}(u) = \mathit{true}$, QPlan sets $\mathsf{size}[u] = N * \Pi_{v \in V_u} \mathsf{size}[v]$ and $\mathsf{sn}(u) = \mathit{true}$ by function ocheck (omitted), and adds a fetching operation to $\mathcal{P}$ for u using φ_u and V_u. It proceeds until for no u in Q, $\mathsf{check}(u) = \mathit{true}$ (line 7). At this point, QPlan returns $\mathcal{P}$ (line 10).

Example 6: Given query Q_0 of Fig. 1 and access schema $\mathcal{A}_0$ of Example 3, QPlan finds $\mathcal{P}$ as follows. Using the actualized constraints Γ of $\mathcal{A}_0$ on Q_0 (see Example 5), it first builds Q_Γ, which is the same as Q_0 except the directions of the edges (u_3, u_1) and (u_3, u_2) are reversed. Using type (1) constraints in $\mathcal{A}_0$, QPlan adds $\mathsf{ft}_1(u_1, \mathit{nil}, \varphi_5, \mathit{true})$, $\mathsf{ft}_2(u_2, \mathit{nil}, \varphi_4, \mathsf{year} \geq 2011 \wedge \mathsf{year} \leq 2013)$ and $\mathsf{ft}_3(u_6, \mathit{nil}, \varphi_6, \mathit{true})$ to $\mathcal{P}$. In the **while** loop, it finds $\mathsf{check}(u_3) = \mathit{true}$ and adds $\mathsf{ft}_4(u_3, \{u_1, u_2\}, \varphi_1, \mathit{true})$ to $\mathcal{P}$. As a consequence of ft_4, it finds that $\mathsf{check}(u_4)$ and $\mathsf{check}(u_5)$ become *true* and thus adds $\mathsf{ft}_5(u_4, \{u_3\}, \varphi_2, \mathit{true})$ and $\mathsf{ft}_6(u_5, \{u_4\}, \varphi_2, \mathit{true})$ to $\mathcal{P}$. Now $\mathcal{P}$ cannot be further improved, and it returns $\mathcal{P}$ with 6 fetching operations,

We next show how this $\mathcal{P}$ identifies G_Q from the IMDb graph G_0 of Example 1 for Q_0. (a) It executes its fetching operations one by one, and retrieves $\mathsf{cmat}(u)$ from G_0 for u ranging over u_1–u_6, with at most 24, 3, 288, 8640, 8640 and 196 nodes, respectively. These are treated as the nodes of G_Q, no more than 17791 in total. (b) It then adds edges to G_Q. For each $(v_3, v_1) \in \mathsf{cmat}(u_3) \times \mathsf{cmat}(u_1)$, it checks whether (v_3, v_1) is an edge in G_0 by using $\mathsf{cmat}(u_1)$, $\mathsf{cmat}(u_2)$ and $\mathsf{cmat}(u_3)$, and the index of φ_1 of $\mathcal{A}_0$, as suggested by fetching operation ft_4 for u_3 given above. If so, (v_3, v_1) is included in G_Q. This checks $24 \times 3 \times 4$ neighbors of $\mathsf{cmat}(u_3)$ in the worst case. Similarly, it examines at most 288, 8640, 8640, 8640 and 8640 candidates matches in G_0 for edges (u_3, u_2), (u_3, u_4), (u_3, u_5), (u_4, u_6) and (u_4, u_6) in Q_0, respectively. This yields at most 34848 edges in G_Q in total. Note that query plan $\mathcal{P}$ is exactly the one described in Example 1, and accesses at most 17923 nodes and 35136 edges in total. Only *part of the data* accessed by $\mathcal{P}$ is included in G_Q for answering Q. □

Correctness & Complexity. For the correctness of QPlan, observe the following about the query plan $\mathcal{P}$ generated for Q and $\mathcal{A}$. (1) $\mathcal{P}$ *is effectively bounded*: indeed, (a) the total amount of data fetched by $\mathcal{P}$ is decided by $\mathcal{A}$ and Q since $\mathcal{P}$ only uses indices in $\mathcal{A}$ to retrieve data; and (b) $Q(G_Q) = Q(G)$ since G_Q includes all candidate matches from G for nodes and edges in Q. By the data locality of subgraph queries, if a node v in G matches a node u in Q, then for any neighbor u' of u in Q, matches of u' must be neighbors of v in G. That is why $\mathsf{cmat}(u)$ collects candidate node matches from neighbors; similarly for edges. (2) $\mathcal{P}$ *is worst-case optimal*: since the **while** loop reduces $|\mathsf{cmat}(u)|$ to be the minimum.

To see that QPlan is in $O(|V_Q||E_Q||\mathcal{A}|)$ time, observe the following. (1) Line 1 is in $O(|\mathcal{A}||E_Q|)$ time. (2) The **for** loop

(lines 2-6) is in $O(|V_Q|)$ time by using the inverted indices. (3) The **while** loop (lines 7-9) iterates $|V_Q|^2$ times, since for each node u in Q, (a) $\mathsf{cmat}(u)$ is reduced only if $\mathsf{cmat}(u')$ is reduced for its "ancestors" u' in Q_Γ, $|V_Q|-1$ times at most, by the definition of $\mathsf{size}[u]$ and check (*i.e.,* $\mathsf{size}[u]$ remains larger than $\mathsf{size}[u']$), and (b) each reduction to $\mathsf{cmat}(u')$ requires us to check once whether $\mathsf{cmat}(u)$ is also reduced as a consequence. In each iteration, $\mathsf{check}(u)$ and $\mathsf{ocheck}(u)$ take $O(\mathsf{deg}(u)|\mathcal{A}|)$ time. As $O(|V_Q| * \Sigma_{u \in V_Q} \mathsf{deg}(u)|\mathcal{A}|) = O(|V_Q||E_Q||\mathcal{A}|)$, the **while** loop takes $O(|V_Q||E_Q||\mathcal{A}|)$ time in total.

V. Making Queries Instance Bounded

Consider a frequent query load $\mathcal{Q}$, such as a *finite* set of parameterized queries as found in recommendation systems. If some queries Q in $\mathcal{Q}$ are not effectively bounded under an access schema $\mathcal{A}$, can we still compute $Q(G)$ in a big graph G? The main conclusion of this section is positive: one can often make all queries in $\mathcal{Q}$ instance-bounded in G and answer them in G by accessing a bounded amount of data.

Extending access schemas. The idea is to extend $\mathcal{A}$ such that its indices suffice to help us fetch bounded subgraphs of G for answering $\mathcal{Q}$. Consider a *constant* M. An *M-bounded extension* $\mathcal{A}_M$ of $\mathcal{A}$ includes all access constraints in $\mathcal{A}$ and additional access constraints of types (1) and (2) (see Section II):

Type (1): $\emptyset \to (l', N)$ Type (2): $l \to (l', N)$

such that $N \leq M$. Note that $\mathcal{A}_M$ is also an access schema.

Instance-bounded patterns. Consider $G \models A_M$. A set $\mathcal{Q}$ of pattern queries is *instance-bounded in G* under $\mathcal{A}_M$ if for all $Q \in \mathcal{Q}$, there exists a subgraph G_Q of G such that

(a) $Q(G_Q) = Q(G)$; and
(b) G_Q can be found in time determined by $\mathcal{A}_M$ and Q *only*.

As a result of (b) and the use of constant M, $|G_Q|$ is a function of $\mathcal{A}$, Q and M. As opposed to effective boundedness, instance boundedness aims to process *a finite set* $\mathcal{Q}$ of queries on *a particular instance* G by accessing a bounded amount of data.

Given these, we answer $\mathcal{Q}$ in a big G as follows. If some queries in $\mathcal{Q}$ are not effectively bounded under $\mathcal{A}$, we extend $\mathcal{A}$ to $\mathcal{A}_M$ by adding simplest access constraints such that all queries in $\mathcal{Q}$ are instance-bounded in G under $\mathcal{A}_M$.

Proposition 5: *For any finite set $\mathcal{Q}$ of subgraph queries, access schema $\mathcal{A}$ and graph $G \models \mathcal{A}$, there exist M and an M-bounded extension $\mathcal{A}_M$ under which $\mathcal{Q}$ is instance-bounded in G.* □

That is, additional access constraints of types (1) and (2) suffice to make $\mathcal{Q}$ instance-bounded in G. We show in [3] that $\mathcal{A}_M$ extends $\mathcal{A}$ with at most $\frac{L_\mathcal{Q}(L_\mathcal{Q}+1)}{2}$ additional constraints, where $L_\mathcal{Q}$ is the total number of labels in $\mathcal{Q}$.

Resource-bounded extensions. Proposition 5 always holds when M is sufficiently large. When M is a small predefined bound indicating our constrained resources, we have to answer the following question, denoted by $\mathsf{EEP}(\mathcal{Q}, \mathcal{A}, M, G)$:

- Input: A finite set $\mathcal{Q}$ of subgraph queries, an access schema $\mathcal{A}$, a natural number M, and a graph $G \models \mathcal{A}$.
- Question: Does there exist a M-bounded extension $\mathcal{A}_M$ of $\mathcal{A}$ such that $\mathcal{Q}$ is instance-bounded in G under $\mathcal{A}_M$?

This problem is decidable in PTIME.

Theorem 6: $\mathsf{EEP}(\mathcal{Q}, \mathcal{A}, M, G)$ *is in* $O(|G| + (|\mathcal{A}| + |\mathcal{Q}|)|E_\mathcal{Q}| + (||\mathcal{A}|| + |\mathcal{Q}|)|V_\mathcal{Q}|^2)$ *time, where* $|G| = |V| + |E|$, $|E_\mathcal{Q}| = \sum_{Q \in \mathcal{Q}} |E_Q|$, $|V_\mathcal{Q}| = \sum_{Q \in \mathcal{Q}} |V_Q|$ *and* $|\mathcal{Q}| = |E_\mathcal{Q}| + |V_\mathcal{Q}|$. □

For a frequent query load $\mathcal{Q}$, we identify A_M; if $\mathcal{A}_M$ exists, we build additional indices on G and make $G \models \mathcal{A}_M$, as *preprocessing* offline. We can then repeatedly instantiate and process query templates of $\mathcal{Q}$ by accessing a bounded amount of data in G, and *incrementally maintain* indices in response to changes to G. Note that real-life queries are typically small.

We prove Theorem 6 by giving a checking algorithm. The algorithm, denoted by EEChk, consists of two steps.

Step (1) (Maximum M-bounded extension): Find all types (1) and (2) constraints $\emptyset \to (l', N)$ and $l \to (l', N)$ on G for all labels l and (l, l') that are in *both* Q and G, such that $N \leq M$ and G satisfies their corresponding cardinality constraints. Let $\mathcal{A}_M$ include all these constraints and all those in $\mathcal{A}$.

Step (2) (Checking): Check whether $\mathcal{Q}$ is instance-bounded in G under $\mathcal{A}_M$ by using a mild revision of $\mathsf{EBChk}(Q, \mathcal{A}_M)$ (see Section III) for each $Q \in \mathcal{Q}$; return "yes" if $\mathsf{EBChk}(Q, \mathcal{A}_M)$ returns "yes" for all Q in $\mathcal{Q}$, and "no" otherwise.

Example 7: Consider a given bound $M = 150$, the IMDb graph G_0 of Example 1, a set $\mathcal{Q}$ with only Q_0 of Fig. 1, and an access schema $\mathcal{A}$ consisting of all constraints in $\mathcal{A}_0$ of Example 3 except φ_4 and φ_5. Given these, EEChk finds a M-bounded extension $\mathcal{A}_M$ of $\mathcal{A}$. (1) It finds, among others, that G satisfies the cardinality constraints of two type 1 access constraints $\varphi_4 = \emptyset \to (\mathsf{year}, 135)$ and $\varphi_5 = \emptyset \to (\mathsf{award}, 24)$, and $135 < M$ and $24 < M$. It extends $\mathcal{A}$ by including φ_4 and φ_5, yielding $\mathcal{A}_M$. (2) It then invokes $\mathsf{EBChk}(Q, \mathcal{A}_M)$ and confirms that $\mathcal{Q}$ is instance-bounded in G under $\mathcal{A}_M$. □

<u>Correctness & Complexity</u>. The correctness of EEChk is ensured by the following. (1) If there exists $\mathcal{A}'_M$ such that $\mathcal{Q}$ is instance-bounded in G under $\mathcal{A}'_M$, then $\mathcal{Q}$ is instance-bounded in G under $\mathcal{A}_M$ for $\mathcal{A}'_M \subseteq \mathcal{A}_M$; hence it suffices to consider the maximum M-bounded extension $\mathcal{A}_M$ of $\mathcal{A}$. (2) Checking instance boundedness is a mild revision of $\mathsf{EBChk}(Q, \mathcal{A}_M)$, with the same complexity stated in Theorem 2.

For the complexity, observe that Step (1) of EEChk is in $O(|G|)$ time, $|\mathcal{A}_M|$ and $||\mathcal{A}_M||$ are bounded by $|\mathcal{A}| + |\mathcal{Q}|$ and $||\mathcal{A}|| + |\mathcal{Q}|$, respectively. Step (2) takes $O((|\mathcal{A}| + |\mathcal{Q}|)|E_\mathcal{Q}| + (||\mathcal{A}|| + |\mathcal{Q}|)|V_\mathcal{Q}|^2)$ time by the complexity of EBChk.

Remark. One might want to find a *minimum* M-extension $\mathcal{A}_M$ of $\mathcal{A}$ such that $\mathcal{Q}$ is instance-bounded under $\mathcal{A}_M$, and $\mathcal{A}_M$ has the least number of access constraints among all M-extensions of $\mathcal{A}$ that make $\mathcal{Q}$ instance-bounded in G. Unfortunately, it is $\log$APX-hard to find such a minimum M-extension for given $\mathcal{Q}$, $\mathcal{A}$, M and G. Here $\log$APX-hard problems are NP optimization problems for which no PTIME algorithms have approximation ratio below $c \log n$, where c is some constant and n is the input size (cf. [6]; see [3] for a proof of this).

VI. Effectively Bounded Simulation Queries

We have seen that effective boundedness helps us answer subgraph queries in big graphs within constrained resources. A natural question asks whether the same idea works for simulation queries, which are non-localized and recursive.

This section settles this question in positive. For effectively bounded simulation queries, we provide (1) a characterization (Section VI-A); (2) a checking algorithm (Section VI-B); and (3) an algorithm for generating effectively bounded and worst-case optimal query plans (Section VI-C), all with the same complexity as their counterparts for subgraph queries. We also give (4) an algorithm for making a finite set of unbounded simulation queries instance-bounded (Section VI-D). We contend that the effective-boundedness approach is generic: it works on general pattern queries, localized or non-localized.

A. Characterization for Simulation Queries

Simulation queries introduce challenges to the analysis.

Example 8: Consider the simulation query $Q_1(V_1, E_2)$ of Example 2, and an access schema $\mathcal{A}_1$ with $\varphi_A = B \to (A, 2)$, $\varphi_B = CD \to (B, 2)$, $\varphi_C = \emptyset \to (C, 1)$, and $\varphi_D = \emptyset \to (D, 1)$. One can verify that $\mathsf{VCov}(Q_1, \mathcal{A}_1) = V_1$ and $\mathsf{ECov}(Q_1, \mathcal{A}_1) = E_1$. However, Q_1 is not effectively bounded. Indeed, G_1 of Fig. 2 matches Q_1, and the maximum match relation $Q_1(G_1)$ "covers" a cycle in G_1 with length proportional to $|G_1|$. That is, while $\mathcal{A}_1$ constrains the neighbors of each node in Q_1, it does not suffice: as shown in Example 2, to check whether v_1 of G_1 matches u_1 of Q_1, we need to inspect nodes of G_1 far beyond the neighbors of v_1, due to the non-localized and recursive nature of simulation queries. $\square$

This suggests a stronger notion of node covers. The *node cover* of an access schema $\mathcal{A}$ on a simulation query Q, denoted by $\mathsf{sVCov}(Q, \mathcal{A})$, is the set of nodes in Q computed as follows:

(a) if a type (1) constraint $\emptyset \to (l, N)$ is in $\mathcal{A}$, then for each node u in Q with label l, $u \in \mathsf{sVCov}(Q, \mathcal{A})$; and
(b) if $S \to (l, N)$ is in $\mathcal{A}$, then for each S-labeled set V_S in Q, a common neighbor u of V_S in Q is in $\mathsf{sVCov}(Q, \mathcal{A})$ if (i) u is labeled with l, (ii) $V_S \subseteq \mathsf{sVCov}(Q, \mathcal{A})$ and (iii) for each node u_S in V_S, (u, u_S) is an edge of Q.

As opposed to VCov for subgraph queries, a node u is in $\mathsf{sVCov}(Q, \mathcal{A})$ if in any graph $G \models \mathcal{A}$, the number of candidate matches of u is bounded in G, *no matter whether* these nodes are in the same neighborhood *or not*. We include u in $\mathsf{sVCov}(Q, \mathcal{A})$ only if some of its children are covered by $\mathcal{A}$ and they bound the candidate matches of u by an access constraint. When we enforce $V_Q = \mathsf{sVCov}(Q, \mathcal{A})$ (see Theorem 9 below), this ensures that *all children* of u have a bounded number of candidates in G. This rules out unbounded matches when retrieving maximum matches by using the indices of $\mathcal{A}$.

The *edge cover* of $\mathcal{A}$ on Q, denoted by $\mathsf{sECov}(Q, \mathcal{A})$, is defined in the same way as $\mathsf{ECov}(Q, \mathcal{A})$ for subgraph queries (Section III), using $\mathsf{sVCov}(Q, \mathcal{A})$ instead of $\mathsf{VCov}(Q, \mathcal{A})$.

Covers for simulation queries are more restrictive than their counterparts for subgraph queries: $\mathsf{sVCov}(Q, \mathcal{A}) \subseteq \mathsf{VCov}(Q, \mathcal{A}) \subseteq V_Q$ and $\mathsf{sECov}(Q, \mathcal{A}) \subseteq \mathsf{ECov}(Q, \mathcal{A}) \subseteq E_Q$.

Analogous to Theorem 1, one can verify the following (see [3] for a proof, which does not use data locality).

Theorem 7: *A simulation query $Q(V_Q, E_Q)$ is effectively bounded under an access schema $\mathcal{A}$ if and only if $V_Q = \mathsf{sVCov}(Q, \mathcal{A})$ and $E_Q = \mathsf{sECov}(Q, \mathcal{A})$.* $\square$

Example 9: Recall Q_1 and $\mathcal{A}_1$ from Example 8. One can verify that neither u_1 nor u_2 in Q_1 is in $\mathsf{sVCov}(Q_1, \mathcal{A}_1)$ and hence, Q_1 is not effectively bounded under $\mathcal{A}_1$ by Theorem 7. This is consistent with the observation of Example 8.

Now define $Q_2(V_2, E_2)$ by reversing the directions of (u_3, u_2) and (u_4, u_2) in Q_1. Then $\mathsf{sVCov}(Q_2, \mathcal{A}_1) = V_2$ and $\mathsf{sECov}(Q_2, \mathcal{A}_1) = E_2$. Hence, Q_2 is effectively bounded under $\mathcal{A}_1$ by Theorem 7. Given G_1 of Fig. 2, we can find $Q_2(G_1) = \emptyset$ without fetching the unbounded cycle of G_1. $\square$

B. Deciding Effective Boundedness of Simulation Queries

We now revisit $\mathsf{EBnd}(Q, \mathcal{A})$ (Section III): given a simulation query Q and an access schema $\mathcal{A}$, it is to decide whether Q is effectively bounded under $\mathcal{A}$. We show that graph simulation does not increase the complexity of $\mathsf{EBnd}(Q, \mathcal{A})$.

Theorem 8: *For simulation queries Q, $\mathsf{EBnd}(Q, \mathcal{A})$ has the same complexity as for subgraph queries, in both the general case and the two special cases stated in Theorem 2.* $\square$

To prove Theorem 8 we give a checking algorithm, denoted by sEBChk, which is the same as EBChk of Fig. 3 except that it uses a revised notion of actualized constraints. For each $S \to (l, N)$ in $\mathcal{A}$ with $S \neq \emptyset$, and each node u in Q with $f_Q(u) = l$, its *actualized constraint for simulation* is $\bar{V}^u_S \mapsto (u, N)$, where $\bar{V}^u_S$ is the maximum set of neighbors of u in Q such that (a) there exists a S-labeled set $V_S \subseteq \bar{V}^u_S$, and (b) for each $u' \in \bar{V}^u_S$, (i) $f_Q(u') \in S$; and (ii) (u, u') is an edge of Q. In contrast to its counterpart defined in Section III, this notion further requires condition (ii) to cope with $\mathsf{sVCov}(Q, \mathcal{A})$.

Example 10: Given $Q_2(V_2, E_2)$ and $\mathcal{A}_1$ considered in Example 9, sEBChk first computes the set Γ of actualized constraints for $\mathcal{A}_1$ on Q_2: $\phi_1 = (u_3, u_4) \mapsto (u_2, 2)$, $\phi_2 = u_2 \mapsto (u_1, 2)$. It then initializes both $\mathcal{B}$ and $\mathcal{C}$ to be $\{u_3, u_4\}$, sets $ct[\phi_1] = 2$, $ct[\phi_2] = 1$, and initializes lists $L[u_1]$, ..., $L[u_4]$ accordingly (see Fig. 3). As in Example 5, it finds that $V_2 \subseteq \mathcal{C}$ and that each edge of E_2 is covered by some constraint in $\mathcal{A}_1$. Thus it returns "yes", *i.e.,* Q_2 is effectively bounded under $\mathcal{A}_1$. $\square$

The correctness of sEBChk follows from the characterization of Theorem 7. Along the same lines as the analysis of EBChk, the proof uses the following property of $\mathsf{sVCov}(Q, \mathcal{A})$: a node u of Q is in $\mathsf{sVCov}(Q, \mathcal{A})$ if and only if either

- there exists $\emptyset \to (l, N)$ in $\mathcal{A}$ and $f_Q(u) = l$; or
- $\bar{V}^u_S \mapsto (u, N)$ and there exists a S-labeled set of Q that is a subset of $\bar{V}^u_S \cap \mathsf{sVCov}(Q, \mathcal{A})$.

Algorithm sEBChk has the same complexity as EBChk: sEBChk is the same as EBChk except the computation of the set Γ of all actualized constraints (lines 1-2 of Fig. 3), which remains in $O(|\mathcal{A}||E_Q|)$ time, the same as for subgraph queries.

C. Generating Effectively Bounded Query Plans

We next show that for effectively-bounded simulation queries Q under an access schema $\mathcal{A}$, we can generate query plans $\mathcal{P}$ such that in any graph G, $\mathcal{P}$ computes $Q(G)$ by accessing a bounded subgraph G_Q of Q, leveraging the indices of $\mathcal{A}$, such that $Q(G) = Q(G_Q)$. Indeed, Theorem 4, the result for subgraph queries, carries over to simulation queries.

Theorem 9: *There exists an algorithm that, given any effectively bounded simulation query Q under an access schema $\mathcal{A}$, generates an effectively bounded and worst-case optimal query plan in $O(|V_Q||E_Q||\mathcal{A}|)$ time.* □

We show that a minor revision sQPlan of algorithm QPlan (Fig. 4) suffices to do these, retaining the same complexity as QPlan. The only difference is that we use actualized constraints for simulation given above, and the stronger notion of node covers instead of data locality. Due to the space constraint we defer the proof and analysis to [3].

Example 11: Given $Q_2(V_2, E_2)$ of Example 9 and $\mathcal{A}_1$ of Example 8, sQPlan generates a query plan $\mathcal{P}$. Using the set Γ of actualized constraints of $\mathcal{A}_1$ on Q_2 (see Example 10), QPlan builds $Q_\Gamma(V_\Gamma, E_\Gamma)$, where $V_\Gamma = V_2$, and E_Γ contains (u_3, u_2), (u_4, u_2) and (u_2, u_1). Initially, it adds $\mathsf{ft}(u_3, \textit{nil}, \varphi_C, \textit{true})$ and $\mathsf{ft}(u_4, \textit{nil}, \varphi_D, \textit{true})$ to $\mathcal{P}$. It then finds that u_2 and u_1 can be deduced from u_3 and u_4 by using Q_Γ, and thus adds $\mathsf{ft}(u_2, \{u_3, u_4\}, \varphi_B, \textit{true})$ and $\mathsf{ft}(u_1, \{u_2\}, \varphi_A, \textit{true})$ to $\mathcal{P}$.

For any graph $G \models \mathcal{A}_1$, we compute $Q_2(G)$ by using $\mathcal{P}$. It retrieves 8 candidate matches for nodes in Q_2, *i.e.,* 4 for u_1, 2 for u_2, 1 for each of u_3 and u_4. It then finds at most 12 edges between these candidates that are possible edge matches by using the indices of $\mathcal{A}_1$: 4 for each of (u_1, u_2) and (u_2, u_1), and 2 for each of (u_2, u_3) and (u_2, u_4). That is, $\mathcal{P}$ fetches a subgraph G_{Q_2} of Q_2, by accessing 8 nodes and 12 edges. □

D. Making Simulation Queries Instance Bounded

Finally, we study finite sets $\mathcal{Q}$ of simulation queries when they are not effectively bounded under an access schema $\mathcal{A}$. We show that Proposition 5 also holds here: for any graph $G \models \mathcal{A}$, there exists an M-bounded extension $\mathcal{A}_M$ of $\mathcal{A}$ under which $\mathcal{Q}$ is instance-bounded in G for some bound M (see Section V for M-bounded extensions, and [3] for a proof).

For a predefined and small M, we revisit $\mathsf{EEP}(\mathcal{Q}, \mathcal{A}, M, G)$ to decide whether there exists an M-bounded extension $\mathcal{A}_M$ of $\mathcal{A}$ that makes $\mathcal{Q}$ instance-bounded in G (see Section V). We show that Theorem 6 remains intact on simulation queries.

Theorem 10: *For simulation queries,* $\mathsf{EEP}(\mathcal{Q}, \mathcal{A}, M, G)$ *is in* $O(|G| + (|\mathcal{A}| + |\mathcal{Q}|)|E_\mathcal{Q}| + (||\mathcal{A}|| + |\mathcal{Q}|)|V_\mathcal{Q}|^2)$ *time.* □

As a proof, we show that a minor revision sEEChk of EEChk (Section V) can check EEP for simulation queries, with the same complexity as EEChk (see [3] for a proof).

VII. Experimental Study

Using real-life data, we conducted three sets of experiments to evaluate (1) the effectiveness of our query evaluation approach based on effective boundedness, (2) the effectiveness of instance boundedness and (3) the efficiency of our algorithms.

Experimental setting. We used three real-life datasets.

Internet Movie Data Graph (IMDbG) was generated from the Internet Movie Database (IMDb) [22], with 5.1 million nodes, 19.5 million edges and 168 labels in IMDbG.

Knowledge graph (DBpediaG) was taken from DBpedia 3.9 [2], with 4.1 million nodes, 19.5 million edges and 1434 labels.

Webbase-2001 (WebBG) recorded Web pages produced in 2001 [1], in which nodes are URLs, edges are directed links between them, and labels are domain names of the URLs. It has 118 million nodes, 1 billion edges and 0.18 million labels.

Access schema. We extracted 168, 315 and 204 access constraints from IMDbG, DBpediaG and WebBG, respectively, by using degree bounds, label frequencies and data semantics. For example, (actress, year) $\rightarrow$ (feature_film, 104) is a constraint on IMDbG, stating that each actress starred in no more than 104 feature films per year. We found it easy to extract access constraints from real-life data. There are many more constraints for our datasets, which we did not use in our tests.

For each constraint $S \rightarrow (l, N)$, we built index by (a) creating a table in which each tuple encodes an actualized constraint $V_S \mapsto (u, N)$; and (b) building an index on the attributes for V_S in the new table, using MySQL 5.5.35.

Pattern queries. For each dataset, we randomly generated 100 pattern queries using its labels, controlled by #n, #e and #p, the number of nodes, edges and match predicates in the ranges [3, 7], [#n-1, 1.5*#n] and [2, 8], respectively. We did not use big patterns to favor conventional methods VF2 and optVF2 (see below), which do not work on large queries.

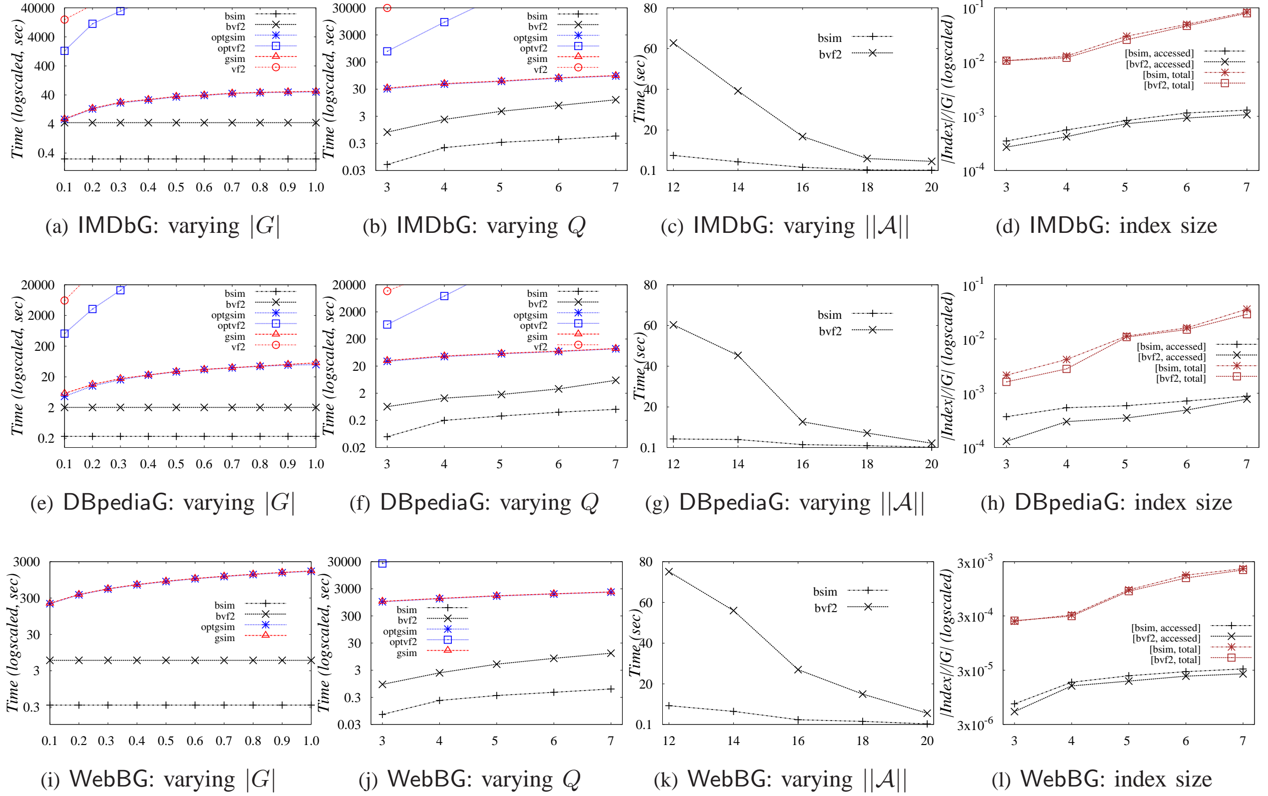
Algorithms. We implemented the following algorithms in C++: (1) EBChk, QPlan, EEChk for subgraph queries, and sEBChk, sQPlan, sEEChk for simulation queries; (2) pattern matching algorithms bVF2 and bSim for subgraph and simulation queries, by using query plans generated by QPlan and sQPlan, respectively; (3) conventional matching algorithms gsim [21] and VF2 (using C++ Boost Graph Library) for simulation and subgraph queries, respectively, and their optimized versions optgsim and optVF2 by using indices in the access constraints.

The experiments were conducted on an Amazon EC2 memory optimized instance r3.4xlarge with 122GB memory and 52 EC2 compute units. All the experiments were run 3 times. The average is reported here.

Experimental results. We next report our findings.

Exp-1: Effectiveness of effective boundedness.

(1) Percentage of effectively bounded queries. We checked the randomly generated queries using algorithms EBChk and sEBChk, and found the following: (1) 61%, 67% and 58% of subgraph queries on IMDbG, DBpediaG and WebBG are effectively bounded under the access constraints described above, and (2) 32%, 41% and 33% for simulation queries, respectively. These tell us that (a) by using a small number of simple access constraints, many subgraph and simulation queries are effectively bounded; and (b) more subgraph queries are bounded than simulation queries under the same constraints, due to their locality (Section II), as expected.



(a) IMDbG: varying $|G|$ (b) IMDbG: varying Q (c) IMDbG: varying $||\mathcal{A}||$ (d) IMDbG: index size

(e) DBpediaG: varying $|G|$ (f) DBpediaG: varying Q (g) DBpediaG: varying $||\mathcal{A}||$ (h) DBpediaG: index size

(i) WebBG: varying $|G|$ (j) WebBG: varying Q (k) WebBG: varying $||\mathcal{A}||$ (l) WebBG: index size

Fig. 5. Effectiveness of effectively bounded query evaluation

(2) Effectiveness of bounded queries. To evaluate the impact of effectively bounded queries, we compared their running time by bVF2 and bSim (with query plans generated by QPlan and sQPlan) vs. the conventional methods VF2, optVF2 and gsim, optgsim. As VF2 and optVF2 are slow, we only report their performance when they ran to completion. Unless stated otherwise, we used all access constraints and full-size datasets.

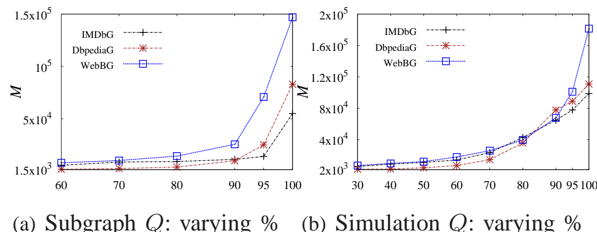
(a) Impact of $|G|$. Varying the size $|G|$ by using scale factors from 0.1 to 1, we report the results on the three datasets in Figures 5(a), 5(e) and 5(i). Observe the following. (1) The evaluation time of effectively bounded queries is *independent of* $|G|$. Indeed, bVF2 and bSim consistently took 4.45s, 2.02s, 5.8s and 0.25s, 0.23s, 0.34s on all subgraphs of IMDbG, DBpediaG and WebBG, respectively. (2) VF2 and optVF2 could *not* run to completion within 40000s on all subgraphs of WebBG, and on subgraphs of IMDbG and DBpediaG with scale factor above 0.3. On the full-size WebBG, bVF2 took 0.9s as opposed to 25729s by optVF2 for queries that optVF2 could process within reasonable time, at least *28587 times faster*. (3) Algorithms optgsim and gsim are sensitive to $|G|$ (note the logarithmic scale of the y-axis), and are much slower than bSim. For instance, on the full-size WebBG, bSim took 0.34s vs. 1630s by optgsim, *4793 times faster*. The improvement of bVF2 over optVF2 is bigger than that of bSim over optgsim as optVF2 has a higher complexity and thus, is more sensitive to $|G|$.

(b) Impact of Q. To evaluate the impact of patterns, we varied #n of Q from 3 to 7. The results, as shown in Figures 5(b), 5(f) and 5(j), tell us the following. (1) The smaller Q is, the faster all the algorithms are, as expected. (2) For all queries, bVF2 and bSim are efficient: they return answers within 12.7s on all three datasets. (3) Algorithms VF2 and optVF2 do not scale with Q. When #n > 4, none of them could run to completion within 40000s, on all three datasets. (4) Algorithms gsim and optgsim are much slower than bSim for all queries.

(c) Impact of $||\mathcal{A}||$. To evaluate the impact of access constraints on bVF2 and bSim, we varied $||\mathcal{A}||$ from 12 to 20 and processed effectively bounded queries using the varied indices in $\mathcal{A}$. As shown in Figures 5(c), 5(g) and 5(k), more access constraints help QPlan and sQPlan get better query plans, as expected. For example, on WebBG, when 20 access constraints were used, bSim and bVF2 took 0.36s and 5.6s, respectively, while they were 9.3s and 75.1s when $||\mathcal{A}|| = 12$.

(3) Size of accessed data. In the same setting as Exp-1(2)(b) above, we examined the size of data accessed by bVF2 and bSim. For each effectively bounded query Q, we examined (a) $|\mathsf{accessed}_Q|$, the size of data accessed, and (b) $|\mathsf{index}_Q|$, the size of indices in those access constraints used, by bVF2 and bSim for answering Q. We report the average in Figures 5(d), 5(h) and 5(l). The results tell us that the query plans accessed no more than 0.13% of $|G|$ for all subgraph and simulation queries on all datasets, with indices less than 8% of $|G|$. This further confirms the effectiveness of our approach.

Exp-2: Effectiveness of instance boundedness. Varying x, we examined the minimum M that makes $x\%$ of queries instance-bounded under M-bounded extensions on IMDbG, DBpediaG and WebBG, via EEChk and sEEChk. As Figures 6(a) and 6(b) show, a small M (compared to $|G|$) suffices to make a large percentage of the queries instance-bounded. For instance, when M is 14113, 25218 and 70916 (resp. 77873,



(a) Subgraph Q: varying % (b) Simulation Q: varying %

Fig. 6. Effectiveness of instance boundedness

89068, 101134), over 95% of all subgraph (resp. simulation) queries randomly generated are instance-bounded in IMDbG, DBpediaG and WebBG, respectively; that is, M is 0.057%, 0.107% and 0.006% of $|G|$ (resp. 0.32%, 0.38% and 0.009%). When M is 181448 (0.016% of WebBG), *all* subgraph and simulation queries become instance-bounded in all datasets.

Expt-3: Efficiency. Finally, we evaluated the efficiency of our algorithms. We found that EBChk, QPlan, sEBChk and sQPlan took at most 7ms, 37ms, 6ms and 32ms, respectively, for all queries on three datasets with all the access constraints.

Summary. From the experiments we find the following. (1) The approach by effective boundedness is practical for pattern queries on large graphs. (a) It is easy to find access constraints from real-life datasets. (b) About 60% (resp. 33%) subgraph (resp. simulation) queries are effectively bounded under a small number of access constraints. (c) Effectively bounded queries scale well with big graphs: their evaluation time is *independent of* $|G|$. (2) The approach is effective for both localized and non-localized queries: bVF2 and bSim outperform optVF2 and optgsim by 4 and 3 orders of magnitude on average on WebBG, respectively. (3) A small M suffices to make queries instance-bounded: 0.006% (resp. 0.009%) of $|G|$ for 95% of subgraph (resp. simulation) queries on WebBG, and 0.013% (resp. 0.016%) to bound *all* queries. (4) Our algorithms are efficient: they take no more than 37ms in all cases.

VIII. Conclusion

We propose to answer graph pattern queries in big graphs by making use of effective boundedness. We have developed techniques underlying the approach: access constraints on graphs, effectively bounded pattern queries, characterizations and algorithms for deciding whether pattern queries are effectively bounded, algorithms for generating (worst-case) optimal query plans if so, and otherwise, algorithms for making queries instance-bounded. We have verified, analytically and experimentally, the effectiveness of the approach: it works for both localized queries and non-localized queries.

One topic for future work is to develop a systematic method for discovering access constraints on graphs. Another topic is to study *incremental* boundedness: Given an access schema $\mathcal{A}$, a graph G and a pattern query Q, it is to incrementally compute $Q(G \oplus \Delta G)$ in response to all changes ΔG to G, by accessing a bounded amount of data from G under $\mathcal{A}$.

Acknowledgment. Cao, Fan and Huang are supported in part by 973 Programs 2014CB340302 and 2012CB316200, NSFC 61133002, Guangdong Innovative Research Team Program 2011D005, Shenzhen Peacock Program 1105100030834361, EPSRC EP/J015377/1, and a Google Faculty Research Award.

References

[1] *http://law.di.unimi.it/webdata/webbase-2001/.*
[2] *http://wiki.dbpedia.org/Downloads39.*
[3] Full version. *http://homepages.inf.ed.ac.uk/s1165433/boundedG.pdf.*
[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.
[5] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. PIQL: Success-tolerant query processing in the cloud. In *VLDB*, 2011.
[6] G. Ausiello. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer, 1999.
[7] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *WWW*, 2004.
[8] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. A budget-based algorithm for efficient subgraph matching on huge networks. In *ICDE Workshops*, 2011.
[9] J. Brynielsson, J. Högberg, L. Kaati, C. Martenson, and P. Svenson. Detecting social positions using simulation. In *ASONAM*, 2010.
[10] Y. Cao, W. Fan, T. Wo, and W. Yu. Bounded conjunctive queries. *PVLDB*, 2014.
[11] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5), 2003.
[12] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. In *PODS*, 2014.
[13] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, 2012.
[14] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.
[15] W. Fan, X. Wang, and Y. Wu. Answering graph pattern queries using views. In *ICDE*, 2014.
[16] W. Fan, X. Wang, and Y. Wu. Querying big graphs within bounded resources. In *SIGMOD*, 2014.
[17] R. W. Faris and S. T. Ennett. Adolescent aggression: The role of peer group status motives, peer aggression, and group characteristics. *Social Networks*, 34(4), 2013.
[18] A. Gubichev, S. J. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*, 2010.
[19] R. Haenni and N. Lehmann. Resource bounded and anytime approximation of belief function computations. *IJAR*, 31, 2002.
[20] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simlations on finite and infinite graphs. In *FOCS*, 1995.
[21] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
[22] IMDb. *http://www.imdb.com/stats/search/* .
[23] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.
[24] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, 2008.
[25] K. Ogaard, H. Roy, S. Kase, R. Nagi, K. Sambhoos, and M. Sudit. Discovering patterns in social networks with graph matching algorithms. In *SBP*, 2013.
[26] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.
[27] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 2008.
[28] C. Smith. By the numbers: 98 amazing facebook statistics. DMR, Mar. 2014.
[29] R. T. Stern, R. Puzis, and A. Felner. Potential search: A bounded-cost search algorithm. In *ICAPS*, 2011.
[30] J. T. Thayer, R. Stern, A. Felner, and W. Ruml. Faster bounded-cost search using inadmissible estimates. In *ICAPS*, 2012.
[31] J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1), 1976.
[32] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3), 1996.
[33] S. Zilberstein, F. Charpillet, P. Chassaing, et al. Real-time problem-solving with contract algorithms. In *IJCAI*, pages 1008–1015, 1999.