Answering Why-questions by Exemplars in Attributed Graphs

Mohammad Hossein Namaki* Qi Song* Yinghui Wu* Shengqi Yang†

Washington State University* WeWork Technology†

{m.namaki, qi.song, yinghui.wu}@wsu.edu* shengqi.yang@wework.com†

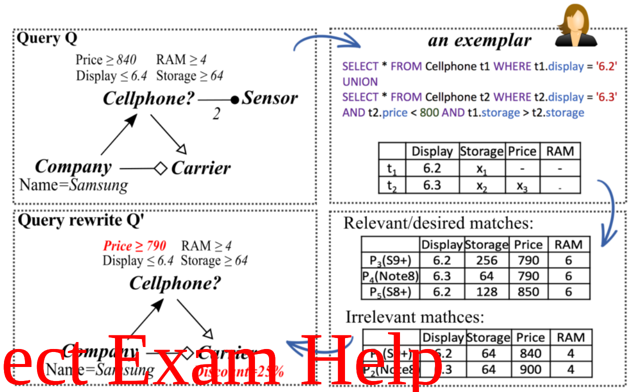
ABSTRACT

This paper studies the problem of *answering Why-questions* for graph pattern queries. Given a query Q, its answers $Q(G)$ in a graph G, and an exemplar $\mathcal{E}$ that describes desired answers, it aims to compute a query rewrite Q', such that $Q'(G)$ incorporates relevant entities and excludes irrelevant ones *w.r.t.* $\mathcal{E}$ under a closeness measure. (1) We characterize the problem by *Q-Chase*. It rewrites Q by applying a sequence of applicable operators guided by $\mathcal{E}$, and backtracks to derive optimal query rewrite. (2) We develop feasible Q-Chase-based algorithms, from anytime solutions to fixed-parameter approximations to compute query rewrites. These algorithms implement Q-Chase by detecting picky operators at run time, which discriminately enforce $\mathcal{E}$ to retain answers that are closer to exemplars, and effectively prune both operators and irrelevant matches, by consulting a cache of star patterns (called *star views*). Using real-world graphs, we experimentally verify the efficiency and effectiveness of Q-Chase techniques and their applications.

ACM Reference Format:

Mohammad Hossein Namaki, Qi Song, Yinghui Wu, Shengqi Yang. 2019. Answering Why-questions by Exemplars in Attributed Graphs. In 2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319890

1 INTRODUCTION



```
SELECT * FROM Cellphone t1 WHERE t1.display = '6.2'
UNION
SELECT * FROM Cellphone t2 WHERE t2.display = '6.3'
AND t2.price < 800 AND t1.storage > t2.storage
```

	Display	Storage	Price	RAM
t_1	6.2	x_1	-	-
t_2	6.3	x_2	x_3	-

Relevant/desired matches:

	Display	Storage	Price	RAM
P_3(S9+)	6.2	256	790	6
P_4(Note8)	6.3	64	790	6
P_5(S8+)	6.2	128	850	6

Irrelevant mathces:

	Display	Storage	Price	RAM
P_1(S9+)	6.2	64	840	4
P_2(Note8)	6.3	64	900	4

Figure 1: Product Search using Exemplars.

Graph pattern queries have been used to search for entities in *e.g.*, social networks and knowledge bases [8, 15], Web exploration, and characterization of data dependencies [7]. A pattern query Q can be modeled as a graph pattern with a designated *focus* entity u_o to specify its instances of interests [6, 24, 38]. Given a graph G, it aims to compute the *answers* $Q(G)$ as the set of entities in G that match u_o with desirable values and topologies. To identify desired answers, existing query-response paradigms [5, 15] require explicitly specified topological and search predicates.

A useful but missing feature in conventional query-response paradigm is the ability to clarify the unexpected or missing entities identified by graph pattern queries. Two common questions users naturally ask are:

- *Why* some unexpected entities are included in $Q(G)$?
- *Why-Not*: Why some relevant entities are missing?

While writing queries are hard, it is often easier for users to characterize answers by providing "example" entities. It is desirable to suggest *query rewrites* that can identify answers closer to the desirable entities characterized by the examples. Consider the following real-world example.

Example 1.1. Fig. 1 illustrates a real-world search scenario from an e-commerce stakeholder. A user issues a graph pattern query Q to find "Samsung" cellphones with carriers and a sensor within its two hops in a product knowledge graph G (partly shown in Fig. 2). The system returns three CellPhones that match the focus Cellphone in the pattern query Q, including $S9+$ (P_1), $Note8$ (P_2) and $S8+$ (P_5).

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06...$15.00
https://doi.org/10.1145/3299869.3319890

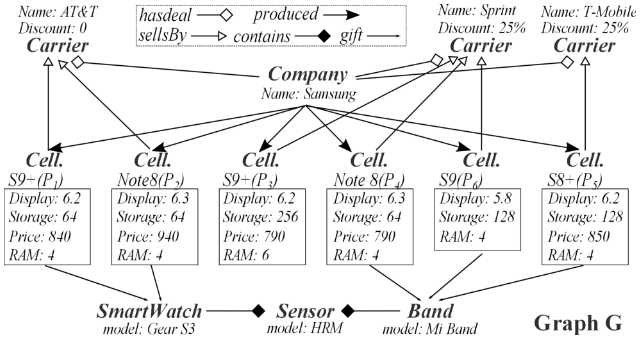


Figure 2: A product knowledge graph.

The user was not fully satisfied with the answer, and provided examples that describe two types of desired cellphones as pattern tuples: (1) t_2 with $6.3'$ display and costs less than \$800 ($x_3 < \$800$), or (2) t_1 with $6.2'$ display and larger storage than at least a match for t_2. Such constraints can be conveniently expressed by a SQL statement (Fig. 1), or directly specified as relevant entities from G.

The system should discover two missing matches *S9+* (P_3) and *Note8* (P_4) that match the exemplar. Moreover, as neither *S9+* (P_1) nor *Note8* (P_2) satisfies the constraints posed by the exemplar, none should be included in the answer. Moreover, lineage information should be provided to clarify how Q should be altered to obtain these desirable answers.

Why-provenance [4, 31] describes the origins of answers and operators by which it arrived in a relational database. A common approach is to use *query rewriting* over materialized views induced by a designated component (*e.g.,* a set of missing or unexpected tuples). The answer of a Why-question is a query rewrite that returns the component. *Can we extend query rewriting to answer Why-questions characterized by user-specified exemplars for graph exploration?*

Example 1.2. Tracing the difference between Q' and Q (Fig. 1), it also clarifies: (1) P_3 was not in $Q(G)$ since it has no wearable sensors; (2) the price constraints should be relaxed to find cheaper cellphones $\{P_3, P_4\}$; and (3) carriers that provided 25% discount do not sell cellphones $\{P_1, P_2\}$. Note that simply adding the constraints in $\mathcal{E}$ to Q does not ensure desired answers.

The above example illustrates the need of effective query rewriting *w.r.t.* specified exemplars. We study a new problem called *answering Why-questions by Exemplars* (WQE):

- **Input**: graph G, pattern query Q, answer $Q(G)$, and an "exemplar" $\mathcal{E}$ with examples and *optional* constraints;
- **Output**: a query rewrite Q' such that $Q'(G)$ is closest to $\mathcal{E}$, under a closeness measure.

Understanding the result of graph pattern queries and efficient query processing are equally important. Solutions for WQE naturally integrate the two processes into a single iterative process (illustrated in Fig. 3). The workflow, in turn, enables exploratory graph search guided by examples.

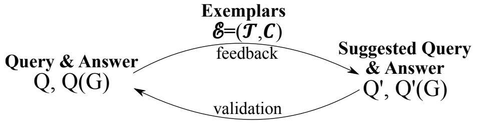


Figure 3: Exploratory graph search with Why-questions

The problem is nevertheless nontrivial: evaluating Q is already intractable (*e.g.,* NP-hard for subgraph isomorphism). This calls for practical cost models and practical algorithms for Why-questions by exemplars.

Contributions. This paper introduces the first framework for answering Why-questions by exemplars for graph pattern queries. We shall refer to graph pattern query simply as *"query"* in the rest of the paper.

(1) We formalize *answering Why-questions by exemplars* (WQE) (Section 3). Given a query Q, graph G, answers $Q(G)$, and exemplars $\mathcal{E}$ specified by a set of pattern tuples $\mathcal{T}$ and *optional* constraints C, it is to compute a query rewrite Q' such that the answers $Q'(G)$ is as close as to the exemplars $\mathcal{E}$ and satisfy C (if any), under a closeness measure. The problem is NP-hard, and is hard to approximate (not in APX).

(2) Despite the hardness, we show that answering Why-questions is within reach in practice for large G.

(a) We formalize WQE with Q-Chase, a Chase process [1] defined on pattern queries *w.r.t.* $\mathcal{E}$ and C. Queries are chased by constraints enforced on its answers, in accordance with atomic updates that in turn ensure the satisfiability.

(b) We develop feasible Q-Chase-based algorithms to compute optimal query rewrites (Section 5). These algorithms efficiently Q-Chase with *anytime* process, and guarantees *bounded delay* to compute picky operators that improve the closeness and update Q, with cost determined by the size of Q and the neighbors of desired nodes up to a bounded hop.

We develop optimization strategies to further reduce the cost of Q-Chase. Our strategies generate and exploit a class of *star views* at run time. These star patterns reduce query processing cost, and suggest useful query fragments that are responsible for missing entities. In addition, we provide a faster heuristic with tunable memory cost.

(3) We provide practical variants of WQE, including *Why-Many* and *Why-Empty*, which seek for query rewrites that can reduce irrelevant matches, and guarantees relevant matches for $\mathcal{E}$, respectively. We show these variants are either fixed-parameter approximable or solvable in PTIME.

(4) Using real-world graphs and benchmark queries, we experimentally verify the effectiveness and efficiency of our algorithms (Section 7). Our Q-Chase-based algorithms are feasible. For example, they converge to near-optimal

rewrites in 4 seconds over DBpedia with 19.7M entities and edges. The optimization techniques improve Q-Chase by 3.41 times. Our user study also verifies that they suggest useful explanations for real-world knowledge search.

Related work. We categorize the related work as follows.

Why-Questions. Why-provenance has been studied for relational data [4, 19] to identify component of queries and data that are responsible for unexpected results. Beyond relational data, special cases have been studied for querying labeled graphs with subgraph isomorphism [11, 21, 28, 33]. An answer is typically a subgraph that is isomorphic to the query graph. A common approach is to modify query topology to obtain desired ones. For example, [11] applies edge insertion/deletions only to include the missing subgraphs. Queries are refined to cover the original answer with diversified counterparts [21]. Maximal common subgraphs between queries and graphs are heuristically computed based on their impact to results [32, 33], to explain "too many" or "too few" answers in graph databases.

Our work differs from these work as follows. (1) We consider both why [4, 12] and why-not [11, 34] questions in a unified workflow rather than separate processes [28]. (2) We consider queries with search predicates and edge to path matching for flexible entity searching, beyond subgraph isomorphism. (3) We explore query rewriting with both relaxation and refinement operators. (4) Our methods only require a set of examples from users, instead of requiring explicitly defined unexpected or desired ones [28]. (5) We develop feasible algorithms with provable optimality guarantees and time cost. These are not addressed in prior work.

Query by example. Query-by-Example (QBE) aims to allow users to express their search intention with examples, thus reduces the manual effort of query construction. QBE has been recently extended to search graphs [14, 23]. The focus is to find answers similar to user-specified examples, rather than query reformulation for Why-provenance. For example, GQBE [14] computes top-k triples with highest similarity scores to user-defined triples, by deriving intermediate subgraphs that can heuristically connect the involved entities. Exemplar queries [23] follows a similar paradigm with subgraphs as examples. Both enforce matches with relationships that are same as their counterparts from the examples.

The Why-questions studied in this work are more involved. (1) We allow users to pose exemplars characterized by tuple patterns as a general representation system. (2) Our algorithms identify both query rewrites and their answers close to exemplars, with provable optimality guarantees. Query rewrites are not discussed by [14, 23].

Graph exploratory search. There has been emerging need in enabling exploratory search with *approximate pattern matching* [13, 25, 26, 29, 35]. Approximate matching relaxes query constraints to find more answers in labeled graphs [26, 29]. The goal is to rectify potential mistakes in query formulation under strict subgraph isomorphism. These methods do not incorporate exemplars as instance level constraints, and typically apply relaxation only. [13, 25] aim to minimize system response time between search sessions caused by both query suggestion and query processing [25]. Instead of automatically compute query rewrite, these methods let the users formulate the query manually by suggesting edges [13]. We present feasible algorithms that adapt queries with both refinement and relaxation operators for search predicates and edge-to-path matching, and ensure the quality of query rewrites measured by closeness to exemplars. These indicate the application of our methods in interactive search and user-friendly access to complex attributed graphs. We are not aware of prior work that addresses these challenges.

2 WHY-QUESTIONS: A SPECIFICATION

2.1 Graph Pattern Queries Revisited

Graphs. We consider a directed, attributed graph $G = (V, E, L, f_A)$, where V is a set of nodes, and $E \subseteq V \times V$ is a set of edges. Each node $v \in V$ (resp. edge $e \in E$) has a label $L(v)$ (resp. $L(e)$). For each node $v \in V$, a function f_A assigns a *tuple* $f_A(v)$ to v. The tuple $f_A(v)$ is a sequence of attribute-value pairs $\{(v.A_1, a_1), \ldots (v.A_n, a_n)\}$. Each pair $(v.A_i, a_i)$ $(i \in [1, n])$ states that the node attribute $v.A_i$, from a finite attribute set $\mathcal{A}$, has a constant value a_i.

We denote the diameter of G as $D(G)$. The active domain adom(A, G) refers to the finite set of values of A from G.

Graph pattern queries. A *graph pattern query* Q is a graph $(V_Q, E_Q, L_Q, F_Q, u_o)$. V_Q (resp. $E_Q \subseteq V_Q \times V_Q$) is a set of query nodes (resp. query edges). Specifically, there is a designated *focus* $u_o \in V_Q$ to specify an entity of interests to be identified by Q. Each node $u \in V_Q$ is associated with (1) a label $L_Q(u)$ (which can be '$\perp$'), and (2) a predicate $F_Q(u)$, which is a set of literals. Each literal $l \in F_Q(u)$ is a constant literal $u.A$ op c, where op is a comparison operator from the set $\{>, \geq, =, \leq, <\}$, and c is a constant. Each edge $e \in E_Q$ has an edge bound $L_Q(e)$, all bounded by an integer b_m $(L_Q(e) \leq b_m)$.

Pattern matching. For each node u in a query Q, a node v in G is a *candidate* of u if (1) $L(v) = L_Q(u)$ ($L(v)$ matches '$\perp$' by default), and (2) for each l= $(u.A$ op $c) \in F_Q(u)$, $(v.A, a) \in F_A(v)$ and a op c. We denote the candidates of u in G as V_u.

We extend *P-homomorphism* [5] to enable approximate edge to path mapping. A *valuation* of Q in G is an *injective* function $h \subseteq V_Q \times V$, such that

- for each node $u \in V_Q$, $h(u) \in V_u$ (candidates), and
- for each edge $e = (u, u')$ in Q, dist$(h(u), h(u')) \leq L_Q(e)$.

Here, $\text{dist}(h(u), h(u'))$ is the distance (the length of the shortest path) from $h(u)$ to $h(u')$ in G. Note that subgraph isomorphism is a special case of the valuation h when b_m=1 [5].

The *matches* of a node $u \in V_Q$, denoted as $Q(u, G)$, refers to all its matches via some valuation $h(u)$ from Q to G. We define the *answer* of Q in G, simply denoted as $Q(G)$, as the matches of the focus u_o of Q in G.

Example 2.1. Fig 2 illustrates a fraction of an attributed knowledge graph G about products of an online store. Each entity carries a type *e.g.,* Cellphone, and a list of attributes (*e.g.,* Display) with corresponding values (*e.g.,* "6.2"). A pattern query Q has focus u_o = Cellphone, with a pattern edge e=(Cellphone, Sensor) carrying edge bound 2. This allows e to be matched by any path between a cellphone and a sensor with length 2 or shorter in G. One can verify that the answer $Q(\text{Cellphone}, G)$ of Q in G is $\{P_1, P_2, P_5\}$.

Remarks. We clarify the following. (1) We consider queries that support edge-to-path mapping beyond subgraph isomorphism, which can identify more relevant matches in graphs with *e.g.,* missing edges [29, 37]. For example, the query Q in Fig. 1 allows identifying desirable cellphones having sensors within 2 hops. (2) Queries with focus nodes (*e.g.,* SPARQL with variables) are commonly used to identify entities in *e.g.,* social networks [6, 24] and knowledge graphs [38]. We focus on single focus, while our techniques easily extend to multiple focus nodes (see Appendix).

2.2 Why-questions with Exemplars

We characterize Why-questions with several notions.

Query rewrites. We extend primitive operators that define graph edit distance [9] to rewrite queries as "graphs" with predicates and edge-to-path matching. We consider eight types of *atomic operators*, including four *refinement* operators AddL, AddE, RfL, and RfE; and four *relaxation* operators RmE, RmL, RxL, and RxE (shown in Table 1). We also introduce an "empty" operator $\emptyset$ for the convenience of formalizing the construction of query rewrites (see Section 4).

We say an atomic operator o is *applicable* to Q, if Q'=$Q \oplus \{o\}$ ($\oplus$ means "apply") is a pattern query, and $Q' \neq Q$ if $o \neq \emptyset$. For example, a relaxation operator RxL(Cellphone.Price $\geq$ 840, Cellphone.Price $\geq$ 790) is applicable to Q in Fig 1. Operator $\text{RmL}(u, l)$ is not applicable if $l \notin F_Q(u)$.

A *query rewrite* Q' of Q is a query obtained by applying a finite sequence of atomic operators $O = \{o_1, \ldots, o_m\}$ to Q (denoted as $Q' = Q \oplus O$), such that (1) $Q_i = Q_{i-1} \oplus \{o_i\}$ (Q_0 = Q), and (2) o_i is applicable to Q_{i-1} ($i \in [1, m]$).

Exemplars. Given a graph G with node attributes $\mathcal{A}$, an *exemplar* $\mathcal{E}$ is a pair $(\mathcal{T}, C)$ defined as follows.

(1) $\mathcal{T}$ is a table with all the attributes from $\mathcal{A}$. Each tuple $t_i \in \mathcal{T}$ is a *tuple pattern.* For each $t_i.A_j$ ($A_j \in \mathcal{A}$), the value of $t_i.A_j$ can be a constant c_{i_j}, a variable x_{i_j}, or a wildcard '_'.

Atomic Operator Classes	Description	Type	Unit Cost
$\text{RmL}(u, l)$	Remove literal $l \in F_Q(u)$	Relax	1
$\text{RmE}((u, u'), b)$	Remove edge e with bound b	Relax	$1 + \frac{b}{D(G)}$
$\text{RxL}(u.A \text{ op } c, u.A \text{ op' } c')$	Relax literal with constant c'	Relax	$1 + \frac{\lvert c'-c \rvert}{\text{range}(A)}$
$\text{RxE}((u, u'), b, b')$	Relax edge bound b	Relax	$1 + \frac{\lvert b-b' \rvert}{D(G)}$
$\text{AddL}(u.A \text{ op } c)$	Add literal l to $F_Q(u)$	Refine	1
$\text{AddE}((u, u'), b)$	Add edge with bound b	Refine	$1 + \frac{b}{D(G)}$
$\text{RfL}(u.A \text{ op } c, u.A \text{ op' } c')$	Refine literal with constant c'	Refine	$1 + \frac{\lvert c'-c \rvert}{\text{range}(A)}$
$\text{RfE}((u, u'), b, b')$	Refine edge bound b	Refine	$1 + \frac{\lvert b-b' \rvert}{D(G)}$

Table 1: Atomic Operators for Pattern Queries

(2) C is a conjunction of literals in the form of x_{i_j} op $x_{i'_{j'}}$ (variable literal), or x_{i_j} op c_{i_j} (constant literal), where op $\in \{>, \geq, =, \leq, <\}$, and c_{i_j} is a constant. It enforces value constraints on the node tuples that match the tuple patterns.

Representation of $\mathcal{E}$. We consider a PTIME-computable predicate vsim that asserts whether a node $v \in V$ of G *matches* a tuple pattern t_i (denoted as $v \sim t_i$). For example, vsim may assert $v \sim t_i$ if for every $t_i.A_j \in \mathcal{T}$, $v.A_j = c_{i_j}$ if $t_i.A_j = c_{i_j}$.

We say a set of nodes $V_C \subseteq V$ *satisfies* an exemplar $\mathcal{E}$=$(\mathcal{T}, C)$, denoted as $V_C \models \mathcal{E}$, if $V_C \models \mathcal{T}$, *i.e.,* for each tuple pattern $t_i \in \mathcal{T}$, there exists a node $v \in V_C$ such that $v \sim t_i$, and $V_C \models C$, *i.e.,* for each literal $l \in C$, $V_C \models l$ if:

- l is in the form of $x_{i_j} = x_{i'_{j'}}$, and for any pair of nodes $v \sim t_i$ and $v' \sim t'_i$ in V_C, $v.A_j = v'.A'_j$;
- l is in the form of x_{i_j} op $x_{i'_{j'}}$ with op $\in \{>, \geq, \leq, <\}$, and for any $v \sim t_i$ (resp. $v' \sim t'_i$) in V_C, there exists $v' \sim t'_i$ (resp.$v \sim t_i$) in V_C, such that $v.A_j$ op $v'.A'_j$; or
- l has form x_{i_j} op c, and $v.A_j$ op c for any $v \sim t_i$ in V_C.

We say $V_C \subseteq V$ is the *representation* of $\mathcal{E}$ in G, denoted as $\text{rep}(\mathcal{E}, V)$, if there is no larger node set $V' \subseteq V$, such that $V_C \subset V'$ and $V' \models \mathcal{E}$. V satisfies $\mathcal{E}$, denoted by $V \models \mathcal{E}$, if $\text{rep}(\mathcal{E}, V) \neq \emptyset$. We have the following result.

Lemma 2.2. *Given graph G and $\mathcal{E}$, it is in $O(|V||\mathcal{E}| + |V|^2)$ time to verify whether $V \models \mathcal{E}$.*

Given a query Q with focus u_o, $\text{rep}(\mathcal{E}, V)$ naturally specifies desired entities given $\mathcal{E}$ by the following table:

	$v \in \text{rep}(\mathcal{E}, V)$	$v \notin \text{rep}(\mathcal{E}, V)$
$v \in Q(G)$	RM ("*relevant match*")	IM ("*irrelevant match*")
$v \in V_{u_o} \backslash Q(G)$	RC ("*relevant cand.*")	IC ("*irrelevant cand.*")

We define the *relevant matches* (resp. *irrelevant matches*) of Q *w.r.t.* $\mathcal{E}$, denoted as $\text{RM}(\mathcal{E}, Q)$ (resp. $\text{IM}(\mathcal{E}, Q)$), as the set $\text{rep}(\mathcal{E}, V) \cap Q(G)$ (resp. $Q(G) \setminus \text{rep}(\mathcal{E}, Q)$), *i.e.,* the matches that satisfy (resp. do not satisfy) the constraints posed by $\mathcal{E}$. The *relevant candidates* $\text{RC}(\mathcal{E}, Q)$ and *irrelevant candidates* $\text{IC}(\mathcal{E}, Q)$ can be defined similarly. Intuitively, a query rewrite Q' should introduce RC as matches, preserve RM, and exclude IM from $Q'(G)$.

We shall consider "nontrivial" exemplars where $\text{rep}(\mathcal{E}, V) \neq \emptyset$. Indeed, any query can identify desired answers for trivial exemplars. Note that it is in $O(|V||\mathcal{E}|+|V|^2)$ time to decide whether $\mathcal{E}$ is nontrivial, following Lemma 2.2.

Remarks. Exemplar $\mathcal{E}$ only enforces value constraints to describe desired entities instead of requiring complex topological constraints, which are harder for users to specify. In practice, it can be (1) directly designated as a set of entities from G by non-expert users [22], or (2) declared as SQL statements over node tables by expert users, which is already supported by *e.g.*, relational graph storage systems [30]. Note that simply adding the constraints in $\mathcal{E}$ to original queries does not necessarily guarantee desired answers.

Why-question. Given a graph G, a *Why-question* W is a tuple $(Q(u_o), \mathcal{E})$, where Q is an *original* query with a focus u_o, and $\mathcal{E}$ is an exemplar. An *answer* of a Why-question $W(Q(u_o), \mathcal{E})$ is a rewrite Q', such that $Q'(G) \models \mathcal{E}$.

Example 2.3. The exemplar $\mathcal{E}$ shown in Fig 1 has $\mathcal{T}$ with two tuple patterns $t_1 = \langle 6.2, x_1, - \rangle$ and $t_2 = \langle 6.3, x_2, x_3 \rangle$, and the constraint C that enforces $c_1 : t_2.x_3 < 800$ and $c_2 : t_1.x_1 > t_2.x_2$. Constraint c_1 enforces that for any cellphone that matches tuple t_2, its price must be below \$800. c_2 poses a selection condition to enforce a comparison between their storage. Given $\mathcal{E}$, query rewrite Q' is an answer to the Why-question $W(Q(u_o), \mathcal{E})$, as $Q'(G)=\{P_3, P_4, P_5\} \models \mathcal{E}$. Given Q' and $\mathcal{E}$: RM = $Q'(G)$, IM = $\emptyset$, RC = $\emptyset$, and IC = $\{P_1, P_2\}$.

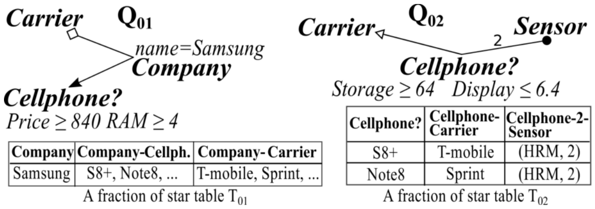
2.3 Star Views: A Primitive Structure

Query evaluation is a cornerstone step in our framework. To reduce query processing cost, we introduce a primitive class of *star views* and their materialized counterpart.

Star views. Given a query $Q = (V_Q, E_Q, L_Q, F_Q, u_o)$, a *star view* $Q.\mathcal{S} = \{Q_1, \ldots, Q_n\}$ of Q is a set of star queries, where each star query Q_i with a center $u_i \in V_Q$ is a subgraph of Q induced by u_i and its neighbors in Q. In addition, it contains an "augmented" edge (u_i, u_o) if u_o is not a neighbor of u_i, with a label as the distance between u_i and u_o in Q. Every node and edge in Q is covered by at least a star query in $\mathcal{S}$.

Star tables. A *materialization* of a star view $Q.\mathcal{S}(G) = \{T_1(G), \ldots T_n(G)\}$ contains a *star table* $T_i(G)$ for each star query Q_i $(i \in [1, n])$, which is a compact encoding of the matches $Q_i(G)$ and their relevance to $\mathcal{E}$. Specifically, (1) $T_i[j][u_i] = v_j$ $(j \in [1, m])$, and v_j is the jth match of the center $Q_i(u_i, G)$; and (2) For each node u in Q_i $(u \neq u_i)$, $T_i[j][u]$ is a set of pairs $(v, \text{dist}(u_i, v))$, one for each match v of u that is a neighbor of v_j. (3) Specifically, for each node $v \in V_{u_o}$ and its occurrence in a star table $T_i[j][u_o]$, it maintains a flag v.rev to bookkeep its relevance to $\mathcal{E}$, as characterized by RM, RC, IM, IC sets.

Example 2.4. Query Q in Fig. 1 can be decomposed into two star views $Q.\mathcal{S} = \{Q_{01}, Q_{02}\}$. Fig 4 shows the views and a fraction of their corresponding star tables T_{01} and T_{02}, respectively. Both views contain the focus node and thus indicate the status of relevant/irrelevant matches/candidates. Instead of evaluating from scratch for upcoming query rewrite Q' that may contain any of these stars, one can incrementally verify the candidates of these views are in $Q'(u_o, G)$.



Company	Company-Cellph.	Company- Carrier
Samsung	S8+, Note8, ...	T-mobile, Sprint, ...

A fraction of star table T_{01}

Cellphone?	Cellphone-Carrier	Cellphone-2-Sensor
S8+	T-mobile	(HRM, 2)
Note8	Sprint	(HRM, 2)

A fraction of star table T_{02}

Figure 4: Star views and star tables.

The star views support fast query evaluation, help to choose promising query rewrite, and facilitate the tracking of lineage information. We provide the details in Section 5.2.

3 ANSWERING WHY-QUESTION

We next introduce measures to quantify "good" query rewrites. Given a query rewrite Q', we introduce two measures, to quantify the closeness of $Q'(G)$ to rep$(\mathcal{E}, V)$, and the cost of updating original query Q' to Q, respectively.

Answer closeness. We introduce a closeness measure between $Q(G)$ and representation of $\mathcal{E}$ in G.

We start with a closeness measure cl(v, t) that measures how "good" a node v matches a tuple pattern t in $\mathcal{E}$=$(\mathcal{T}, C)$. Given a threshold θ, the predicate vsim asserts $v \sim t$ if cl$(v, t) \geq \theta$. Denote as $\mathcal{A}(t)$ the attributes of t. The measure cl(v, t) is defined as

$$\text{cl}(v, t) = \frac{\sum_{A \in \mathcal{A}(t)} \text{cl}(v.A, t.A)}{|\mathcal{A}(t)|}$$

Here, cl$(v.A, t.A)$ is 1 if $t.A$='_' or a variable. Otherwise, for each $t.A = c$ in $\mathcal{E}$, cl$(v.A, t.A) \in [0, 1]$ is a similarity score computed by established metrics. The closeness of a node v and $\mathcal{E}$=$(\mathcal{T}, C)$ is aggregated as cl$(v, \mathcal{E})$ = $\max_{t \in \mathcal{T}, v \sim t}$ cl(v, t). This resembles the well-established tuple similarity measures in clustering [27], where two data points are closer if they have more similar values on more attributes.

Given $Q(G)$ such that $Q(G) \models \mathcal{E}$, the *closeness* between $Q(G)$ and $\mathcal{E}$ is then computed as

$$\text{cl}(Q(G), \mathcal{E}) = \frac{\sum_{v \in \text{RM}(\mathcal{E}, Q)} \text{cl}(v, \mathcal{E}) - \lambda |\text{IM}(\mathcal{E}, Q)|}{|V_{u_o}|}$$

The closeness is rewarded by the total closeness of the relevant matches in RM$(\mathcal{E}, Q)$, quantified by the first term. It is penalized by the size of irrelevant matches IM$(\mathcal{E}, Q)$ introduced by Q, tuned by a factor λ. Both terms are normalized by $|V_{u_o}|$, the size of the candidates of focus u_o (Section 2). The larger cl$(Q(G), \mathcal{E})$ is, the better Q should be.

Query updating cost. Our second measure quantifies the editing cost between Q and Q' by extending weighted graph edit distance [9]. Given $Q' = Q \oplus O$, we assign a cost $c(o) \in [1, 2]$ for each atomic operator $o \in O$ (see Table 1), which counts the unit cost 1 for all operators, and additional weight on "relative" difference, normalized by the range of active domain of A for literal modifications and diameter $D(G)$ for edge bound updates, respectively. The query updating cost $c(O)$ is computed as $c(O) = \sum_{o \in O} c(o)$.

Example 3.1. Consider the operators shown in Fig. 1, the cost is 1 for AddL discount, and $1 + \frac{840-790}{150} = 1.33$ for RxL($\mathit{Cellphone.Price} \geq 840$, $\mathit{Cellphone.Price} \geq 790$). A set of operators and their costs are shown below.

	operator	cost
o_1	AddL(Carrier.Discount = 25%)	1
o_2	RmE((Cellphone, Sensor),2)	$1\frac{2}{3}$
o_3	RxL(Cellphone.Price ≥ 840, Cellphone.Price ≥ 790)	$1\frac{1}{3}$
o_4	RxL(Cellphone.Price ≥ 840, Cellphone.Price ≥ 750)	1.6
o_5	RfL(Cellphone.RAM ≥ 4, Cellphone.RAM ≥ 6)	2
o_6	RmL(Cellphone.Display)	1
o_7	AddL(Cellphone.Display)	1

For the candidate nodes in $V_{\mathsf{Cellphone}}$ in Fig. 2, we have $\mathsf{cl}(P_3, t_1) = 1$, $\mathsf{cl}(P_4, t_2) = 1$, and $\mathsf{cl}(P_5, t_1) = 1$. Thus, nodes $\{P_3, P_4, P_5\}$ are considered to be relevant candidates, and the rest ones are irrelevant to $\mathcal{E}$. By applying operators o_1, o_2, and o_3, query rewrite Q' achieves closeness $\mathsf{cl}(Q'(G), \mathcal{E}) = \frac{1}{2}$ when $\lambda = 1$, with updating cost 4.

We now formalize the problem of answering Why-questions with exemplars and budgeted query rewriting. We assume the predicates vsim is given.

Problem statement. Given a graph G, a Why-question $W(Q(u_o), \mathcal{E})$, answer $Q(G)$, and a budget B, the problem of *answering why-question by exemplar* (WQE) is to compute an optimal query rewrite $Q' = Q \oplus O^*$ such that

$$O^* = \underset{O:c(O) \leq B}{\arg\max}\, \mathsf{cl}(Q'(G), \mathcal{E})$$

This problem is nontrivial. The hardness remains robust even when the matches can be computed in PTIME.

Theorem 3.2. *The problem* WQE *is (1)* NP*-hard, and (2) not in* APX *(not approximable in constant ratio), even when Q, query rewrites, and G are constrained to be trees.*

Proof sketch: We show the NP-hardness by a reduction from subgraph isomorphism. To show the inapproximability, we construct a reduction from minimum 3DNF satisfiability, an optimization problem is known to be not in APX [10]. □

Example 3.3. Recall the Why-questions in Fig 1. Query rewrite Q' with closeness $\mathsf{cl}(Q'(G), \mathcal{E}) = \frac{1}{2}$ and cost 4 is the optimal query rewrite under cost budget $B = 4$. Another query rewrite $Q''{=}Q \oplus \{o_1, o_5, o_6\}$ with the same cost, yet has closeness $\mathsf{cl}(Q''(G), \mathcal{E}) = \frac{1}{6}$, thus is not optimal for B.

Remarks. Ideally, an optimal query rewrite should be the "ground truth" query in users' mind. Nevertheless, given that users are not aware of such queries from scratch but can easily provide exemplar $\mathcal{E}$, we compute optimal queries that can "cover" relevant matches and exclude irrelevant ones from $\mathcal{E}$. Indeed, a ground truth query should have a *theoretically optimal closeness* $\mathsf{cl}^* = \frac{|\mathsf{rep}(\mathcal{E}, V)|}{|V_{u_o}|}$. We thus align the optimal query rewrite Q' with the "ground truth" queries by optimizing the answer closeness.

4 Q-CHASE: A CHARACTERIZATION

Before we study feasible algorithms for answering Why-questions with exemplars, we first introduce Q-Chase, a new technique that revises Chase [1] to characterize the computation of pattern queries under exemplar constraints.

Q-Chase. Chase is a well established technique for deciding logical consequence of data constraints. Unlike Chase that enforces constraints on database instances, Q-Chase enforces constraints posed by exemplars to graph queries, and verifies query rewrites by their answer closeness.

Q-Chase step. We start with an initial pair $(Q_0, \mathcal{E}_0)$, where $Q_0 = Q$, $\mathcal{E}_0{=}(\mathcal{T}_0, C_0)$, and $\mathcal{T}_0{=}C_0{=}\emptyset$. Each Q-Chase step i extends $(Q_i, \mathcal{E}_i)$ to $(Q_{i+1}, \mathcal{E}_{i+1})$ by applying an atomic operator o (can be $\emptyset$; *i.e.*, $Q_{i+1}{=}Q_i$) and enforcing constraints in $\mathcal{E}$ over $Q_{i+1}(G)$. We define a Q-Chase *step* of Q by $\mathcal{E}_i$ at $(Q_i, \mathcal{E}_i)$ as

$$(Q_i, \mathcal{E}_i) \xrightarrow{v,t,l} (Q_{i+1}, \mathcal{E}_{i+1})$$

where $Q_{i+1} = Q_i \oplus \{o\}$, and o is an atomic operator (may be $\emptyset$), t is a pattern tuple in $\mathcal{T}$, l is a literal in C, and $v \in V_{u_o}$. Specifically, a Q-Chase step satisfies the following rules:

- If o is a relaxation operator, then (a) if $v \notin Q_i(u_o)$ and v is a new match of u_o in Q_{i+1}, set $Q_{i+1}(G){=}Q_i(G) \cup \{v\}$; (b) if $v \sim t$, add t to $\mathcal{T}_{i+1}$; and (c) if $Q_i(G) \not\models l$ and $Q_{i+1}(G) \models l$, add l to C_{i+1};
- If o is a refinement operator, then (a) if $v \in Q_i(u_o)$ and $v \notin Q_{i+1}(G)$, set $Q_{i+1}(G) = Q_i(G) \setminus \{v\}$; (b) if $v \sim t$ and $Q_{i+1}(G) \setminus \{v\} \not\models \mathcal{T}$, set $\mathcal{T}_{i+1} = \mathcal{T}_i \setminus \{t\}$; and (c) if $Q_{i+1}(G) \not\models l$, set $C_{i+1}{=}C_i \setminus \{l\}$.

That is, the query Q_i is chased with the constraints (t, l), yielding a new query Q_{i+1} with an updated relevant matches $\mathsf{RM}(\mathcal{E}_{i+1}, Q_{i+1})$. The step is *valid* if (1) o is either empty or *applicable* to Q_i (Section 2), and (2) $Q_{i+1}(G) \models \mathcal{E}_{i+1}$.

A Q-Chase *sequence* $\rho{=}\{(Q_0, \mathcal{E}_0), \ldots (Q_k, \mathcal{E}_k)\}$ of Q by $\mathcal{E}$ is a sequence of valid Q-Chase steps. A valid Q-Chase sequence ρ is *canonical*, if for any $i \in [0, k-1]$, and any constant literal $l{=}v.A$ op c (resp. edge e) in Q_i, there exists no operators o_j and o_k ($i \leq j \leq k \in [0, k-1]$), such that l (resp. e) is relaxed or removed by o_j and refined or added by o_k, or vice versa. Indeed, o_j and o_k "cancel out" each other and can be effectively replaced by a single operator. We consider valid, canonical Q-Chase sequences in the rest sections.

Termination of Q-Chase. Denote as $\rho(O)$ the set of operators enforced to Q by a Q-Chase sequence ρ=$\{(Q_0, \mathcal{E}_0),\ldots(Q_k, \mathcal{E}_k)\}$, we define the cost of ρ as $c(\rho)$=$c(O)$. The Q-Chase ρ is *terminal* if (1) there exists no (v, t, l) such that $(Q_k, \mathcal{E}_k) \xrightarrow{v,t,l} (Q_{k+1}, \mathcal{E}_{k+1})$ is a valid Q-Chase step, otherwise (2) $\forall o$, $c(\rho) + c(o) \geq B$. The pair $(Q_k, \mathcal{E}_k)$ is the *result* of a terminal Q-Chase sequence ρ.

We now introduce Q-Chase tree, a symbolic characterization of the computational space for optimal query rewrites.

Normal form. A canonical Q-Chase sequence ρ = $\{(Q_0, \mathcal{E}_0),\ldots(Q_k, \mathcal{E}_k)\}$ is in a *normal form*, if there exists a number $j \in [0, k-1]$, such that the sub-sequence ρ_1=$\{(Q_0, \mathcal{E}_0),\ldots(Q_j, \mathcal{E}_j)\}$ only enforces relaxation or empty operators, and the sub-sequence ρ_2=$\{(Q_{j+1}, \mathcal{E}_{j+1}),\ldots(Q_k, \mathcal{E}_k)\}$ only enforces refinement or empty operators. That is, ρ_1 keeps "relaxing" Q, and ρ_2 keeps "refining" Q_j and follow-up query rewrites.

We say two Q-Chase sequences are *equivalent* if they both start with pair $(Q_0, \mathcal{E}_0)$ and yields same pair $(Q_k, \mathcal{E}_k)$. The result below shows that the normal form is an equivalent encoding of canonical Q-Chase sequences.

Lemma 4.1. *For any valid canonical* Q-Chase *sequence* ρ = $\{(Q_0, \mathcal{E}_0),\ldots(Q_k, \mathcal{E}_k)\}$*, there exists an equivalent* Q-Chase *sequence* ρ' *in a normal form.*

We provide a constructive proof that transforms canonical sequences to equivalent normal forms (see Appendix).

Example 4.2. Recall the queries in Fig. 1. A Q-Chase step from Q_0 to Q_1 is $(Q_0, \mathcal{E}_0) \xrightarrow{P_4,t_2,c_1} (Q_1, \mathcal{E}_1)$ by applying o_3. Since o_3 is a relaxation operator and node P_4 is a new match of u_o in G, we add P_4 to $Q_1(u_o, G)$. Similarly, we add t_2 to $\mathcal{T}_1$ and c_1 to C_1. For simplicity, we show a sequence of operators to represent a corresponding Q-Chase sequence. Consider three sequences $\rho_1 = \langle o_1, o_3, o_6, o_2, o_7\rangle$, $\rho_2 = \langle o_1, o_3, o_2\rangle$, and $\rho_3 = \langle o_3, o_2, o_1\rangle$. ρ_1 is neither canonical nor in a normal form, since o_6 and o_7 "cancel out" each other, and refinement o_1 is performed before relaxation o_2. ρ_2 is canonical, and ρ_3 is its equivalent sequence in normal form.

Q-Chase tree. Given G, exemplar $\mathcal{E}$ and a budget B, a *canonical* Q-Chase *tree* of Q with G for Why-question $W(Q(u_o), \mathcal{E})$, denoted as Q-Chase (W, G, B), is a tree whose root is $(Q, \mathcal{E}_0)$. There exists an edge from $(Q_i, \mathcal{E}_i)$ to $(Q_{i+1}, \mathcal{E}_{i+1})$ with label (v, t, l), if $(Q_i, \mathcal{E}_i) \xrightarrow{v,t,l} (Q_{i+1}, \mathcal{E}_{i+1})$ is a valid Q-Chase step. Any path from the root to a leaf $(Q_k, \mathcal{E}_k)$ is a terminal canonical sequence ρ in normal form, with $(Q_k, \mathcal{E}_k)$ as its *result*.

Theorem 4.3. *Given a Why-question* $W(Q(u_o), \mathcal{E})$*, graph* G *and budget* B*, (1) a query rewrite* Q_k *is an answer of* $W(Q(u_o), \mathcal{E})$ *if and only if there is a path* ρ *from the root with a result* $(Q_k, \mathcal{E})$ *in* Q-Chase (W, G, B)*, and (2) it is optimal if no other path* ρ' *has a result* $(Q'_k, \mathcal{E})$ *such that* $\mathsf{cl}(Q'_k(G), \mathcal{E}) > \mathsf{cl}(Q_k(G), \mathcal{E})$.

The above result can be verified by the invariant that each edge in Q-Chase (W, G, B) encodes a valid Q-Chase step i, such that $Q_{i+1}(G) \models \mathcal{E}_{i+1}$. Thus $Q_k(G) \models \mathcal{E}$, and ρ has cost $c(\rho) \leq B$. Hence, Q_k is an answer to $W(Q(u_o), \mathcal{E})$ by definition, and is optimal if ρ ensures a maximum closeness.

The computation of optimal answers for a Why-question is now equivalent to finding the optimal terminal sequence that maximizes answer closeness in Q-Chase (W, G, B).

Interpretation of Q-Chase. Beyond a symbolic encoding for computing query rewrites, Q-Chase also characterizes practical search scenarios. (1) It provides useful lineage information to support exploratory search. At each Q-Chase step $(Q_i, \mathcal{E}_i) \xrightarrow{v,t,l} (Q_{i+1}, \mathcal{E}_{i+1})$, and a match (resp. candidate) v that is removed from (resp. added to) $Q_{i+1}(G)$, it can easily trace (a) the operator o that is responsible for the change, or (b) the affected tuple pattern t and conditions l in exemplars, for provenance purpose. (2) Each Q-Chase step may correspond to a *search session* in exploratory search, and its time cost reflects *system response time* [25, 29]. We verify the application of Q-Chase in these scenarios (see Section 7).

5 Q-CHASE BASED ALGORITHMS

Given Theorem 4.3, an immediate yet naive Q-Chase implementation exhaustively enumerates every sequence in Q-Chase (W, G, B). This is clearly not feasible for large G. We next develop an efficient implementation of Q-Chase.

5.1 Computing Optimal Query Rewrite

We first introduce an *anytime* implementation of Q-Chase. The algorithm can be interrupted at any time to provide a query rewrite along with matches that are close to $\mathcal{E}$. Moreover, it has a *bounded delay time* (the cost to update query rewrite) to improve the answer, and early terminates whenever possible, without enumerating all Q-Chase sequences.

Algorithm. The algorithm, denoted as AnsW and illustrated in Fig. 5, adopts a best first search strategy with backtracking to simulate the traversal of Q-Chase (W, G, B), and reports the best query rewrite upon request.

Auxiliary structures. Algorithm AnsW dynamically maintains a primary priority queue $\mathcal{P}$ to record query rewrites and their (estimated) matches, ranked by their closeness. Each entry $\mathcal{P}[Q']$ records (1) a pair $\langle Q', Q'(G)\rangle$, where Q'=$Q \oplus O$ is encoded with Q and operators O, and (2) a secondary priority queue $Q'.O$, that records applicable updates for Q, ranked by their *pickiness* as a measure of their likelihood to improve closeness (see **"Picky Operators"**).

```
Algorithm AnsW
Input: graph $G$, Why-question $W$=($Q, \mathcal{E}$), budget $B$.
Output: the optimal query rewrite $Q^*$ that answers $W$.
1.  set $R(u_o)$ := rep($\mathcal{E}, V$); $I(u_o)$:=$V_{u_o} \setminus R(u_o)$; set cl* = $\frac{|R(u_o)|}{|V_{u_o}|}$;
2.  $Q(G)$ := Match($Q, G$); $Q_0 = Q^*$:=$Q$;
3.  initializes priority queue $\mathcal{P}$:={⟨$Q_0, Q_0(G)$⟩};
    /* best-first Q-Chase with backtracking */
4.  while $\mathcal{P}$ is not empty do
5.    pair ⟨$Q, Q(G)$⟩:= $\mathcal{P}$.peek();
    /* decides to "relax" or "refine" and finds next best operator */
6.    operator o := NextOp (⟨$Q, Q(G)$⟩, $R(u_o)$, $I(u_o)$, $B$, cl($Q^*$), cl*);
7.    if o := ∅ then $\mathcal{P}$.poll(); continue ; /* Backtracking */
    /* simulating Q-Chase steps */
8.    $Q'$ := $Q \oplus$ o; $Q'(G)$ := Match($Q', G$); update cl($Q'$);
    /* prune non-promising Q-Chase sequences at $Q'$ */
9.    if Prune ($Q', Q'(G)$) then continue ;
10.   if $Q'(G) \models \mathcal{E}$ and cl($Q'$) > cl($Q^*$) then
11.     $Q^*$ := $Q'$; cl($Q^*$) = cl($Q'$);
12.     return $Q^*$ upon request; /* anytime response */
13.   if cl($Q^*$) = cl* then break ; /* theoretically optimal */
14.   $\mathcal{P}$ := $\mathcal{P}$.enqueue (⟨$Q', Q'(G)$⟩);
15. return $Q^*$;
```

Figure 5: Algorithm AnsW**: main framework**

Given a Why-question $W(Q(u_o), \mathcal{E})$, AnsW computes optimal answer Q^* with the following steps.

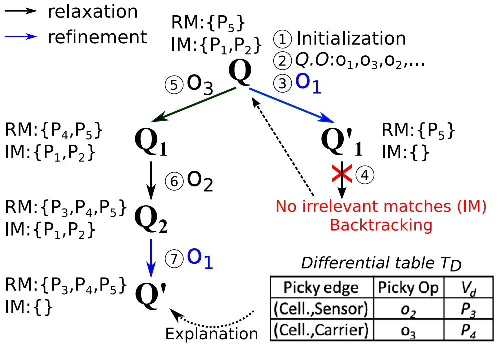
(1) AnsW first initializes a set $R(u_o)$ = rep($\mathcal{E}, V$), the projection of $\mathcal{E}$ in V ("relevant set"), and $I(u_o) = V_{u_o} \setminus R(u_o)$ ("irrelevant set"), and computes a theoretical upper bound of the closeness cl* = $\frac{|R(u_o)|}{|V_{u_o}|}$ (line 1). Indeed, cl* is the maximum closeness a query rewrite can achieve via Q-Chase. It then computes the matches $Q(G)$ by invoking Match (see "**Query Processing**"), and initializes the primary queue $\mathcal{P}$ with Q (as Q_0) and $Q(G)$ (as $Q_0(G)$) (lines 2-3). This simulates the creation of the root of Q-Chase (W, G, B).

(2) AnsW then performs best first search (lines 3-14).

(a) It starts with a verified query Q to be extended with new operators (encoding a node $(Q, \mathcal{E})$ in Q-Chase (W, G, B) tree), and invokes a procedure NextOp to produce operators (lines 5-6). Specifically, procedure NextOp decides

- whether to relax or refine Q, thus enforces a normal form of the underlying canonical Q-Chase sequence that it simulates (see **"Query Pruning"**); and
- dynamically produces a set of "picky" operators $Q.\mathcal{O}$ that are likely to improve the closeness (see **"Picky Operators"**), under a certain generation mode.

(b) It then constructs a new query rewrite with the best operator and evaluate the closeness (simulating a single Q-Chase step), and updates the best query rewrite Q^* so far accordingly (lines 8-11). It also prunes unpromising Q-Chase sequences (procedure Prune; line 9) for early termination.



Picky edge	Picky Op	V_d
(Cell.,Sensor)	o_2	P_3
(Cell.,Carrier)	o_3	P_4

Figure 6: Algorithm AnsW**: Simulation of** Q-Chase**.**

The above process is controlled by priority queue $\mathcal{P}$. When there is no applicable operator (line 6), it completes a terminal Q-Chase sequence and backtracks to a new sequence. The anytime process (early) terminates when no new query rewrite is extendable under budget B (line 4), or the theoretically optimal query rewrite Q^* is identified (line 13); and returns Q^* by demand (line 12).

Example 5.1. We show a step-by-step overview of AnsW in simulating a Q-Chase sequence that chase from Q in Fig. 1 to Q_2. (1) AnsW first computes $Q(G)$ (assume it is not known) with procedure Match, and initializes the primary priority queue $\mathcal{P}$ with Q and $Q(G)$. (2) Procedure NextOp then generate a set of "picky" atomic operators from Q that help in improving its closeness (to be discussed). It returns o_1 as the best operator. (3) Setting Q'_1=$Q \oplus \{o_1\}$, it updates cl(Q'_1). This completes a single Q-Chase step from Q to Q'_1. As the current Q-Chase sequence has no "relaxation" phase, Q is continuously refined. (4) As no irrelevant matches can be removed, cl(Q'_1) can no longer be improved.

Steps (1)-(4) completes the simulation of a terminal Q-Chase sequence and backtracks from Q'_1 to Q. In steps (5)-(7), it simulates another terminal Q-Chase sequence in normal form, where Q is first relaxed to Q_2 with operator o_2. As all the relevant matches are included, AnsW terminates relaxation phase and triggers refinement to remove irrelevant matches. This refines Q_2 to Q', and terminates Q-Chase due to that no irrelevant matches can be removed. As cl(Q') ≥ cl(Q'_1), it returns Q' as the optimal query rewrite.

We next introduce the three procedures NextOp, Prune, and Match, all access the star views structures (Section 2.3).

5.2 Match: Querying with Star Views

Given a query Q, Procedure Match constructs star views $Q.\mathcal{S}$ by edge decomposition and materializes $Q.\mathcal{S}$ if necessary. It then computes $Q(G)$ over the star tables as materialized views, by only verifying the candidates $v \in V_{u_o}$ that is a match of u_o in star views. The verification optimizes the computation of P-homomorphism [5] with a Threshold-Algorithm based strategy that joins the matches of stars that

```
Procedure NextOp
Input: queue entry $\{\langle Q, Q(G)\rangle\}$ of $\mathcal{P}$, relevant set $R(u_o)$, irrelevant
       set $I(u_o)$, cost bound $B$
Output: a best operator o to be applied on $Q$.
if $Q.O := \emptyset$ then /* $Q$ is first visited; generate operators */
   initialize $Q.O$; extract $O$ from $Q$ ($Q$=$Q_0 \oplus O$);
  /* enforce "refinement-only" to current Q-Chase */
  if RefineCond ($Q.\mathcal{S}(G)$, cl($Q$), cl($Q^*$)) then
     construct picky edges $E_P \subseteq E_Q$;
     for each picky edge $(u, u') \in E_P$ do
        $Q.O$.enqueue(GenRf($Q.\mathcal{S}(G), E_P, B$));
  /* generate relaxation operators for current Q-Chase*/
  if RelaxCond ($O$, cl($Q$), cl$^*$) then
     construct picky edges $E'_P \subseteq E_Q$;
     for each picky edge $(u, u') \in E'_P$ do
        $Q.O$.enqueue(GenRx($Q.\mathcal{S}(G), E'_P, B$));
  /* $Q$ is revisited; returns next operator with best pickiness */
return $Q.O$.poll ();
```

Figure 7: Procedure NextOp

cover Q at runtime [36]. Match terminates the verification as soon as v is verified to be a match. (3) It updates the relevance flag v.stat (Section 2.3) in $Q.\mathcal{S}(G)$ with $Q(G)$, which will be consulted by NextOp for operation generation.

Caching the Stars. Q-Chase sequences tend to produce highly similar queries, with minor changes in predicates or edge bounds. AnsW exploits this property and dynamically maintains a global cache $\mathcal{V}$ of materialized star views, which can be accessed by Match at runtime. It adopts a replacement policy for each cached star view, by (a) counting a "hit" number from query rewrites, incremented each time it is used for incremental evaluation, and decayed by a time factor, and (b) applying a least-hit policy to replace the star view with minimum hit rate.

5.3 NextOp: Generating Picky Operators

Given a query rewrite Q to be extended, procedure NextOp generates a set of applicable operators $Q.O$, and chooses a best one $o \in Q.O$, such that Q'=$Q \oplus \{o\}$ *may* improve the closeness cl(Q) (line 6 in AnsW). We call such o "*picky operator*", and apply a *dichotomy* strategy that consults star views $Q.\mathcal{S}$ (computed by Match) to compute picky operators.

Generate Picky Relaxation. Relaxation operators (RxL, RxE, RmE, and RmL) only introduce new matches from candidates. Thus, a picky relaxation operator should introduce matches from relevant candidates ("new relevant matches", denoted by RC(o)), and avoid introducing matches from IC nodes. Given this intuition, NextOp performs the following.

(1) It first identifies a set of *picky edges* E_p from $Q.\mathcal{S}$ over which the associated operators will be relaxed. The idea is to collect pattern edges (u_i, u_o) (resp. (u_o, u_i)) in each star Q_i, such that there exists at least an RC node v_o (v_o.stat ='RC') reachable to (from) a candidate of u_i. Indeed, it suggests by relaxing edge and node condition on such picky edges in E_p will likely to introduce u_o as a new match.

(2) For each picky edge *w.l.o.g.* $e = (u_o, u')$, it invokes a procedure GenRx to produce $Q.O$, by relaxing literals or edge label, guided by the candidates of u_o and u', and $Q.\mathcal{S}(G)$. For each operator o, it also computes a *pickiness* score as $p(o) = \frac{\sum_{v \in \overline{\text{RC}}(o)} \text{cl}(v, \mathcal{E})}{|V_{u_o}|}$, where $\overline{\text{RC}}(o)$ is an overestimation of RC(o), and includes all RC nodes of u_o induced by the candidates over picky edges.

Lemma 5.2. *For any applicable relaxation operator o of Q, $p(o) \geq \text{cl}(Q \oplus \{o\}, \mathcal{E}) - \text{cl}(Q, \mathcal{E})$.*

That is, the pickiness $p(o)$ estimates the gain of closeness if o is applied. The operators are sorted by $p(\cdot)$ in $Q.O$. The best one is returned to simulate a Q-Chase step.

Procedure GenRx. We show how GenRx generates RmE, RmL and RxL. We defer the discussion of RxE to Appendix.

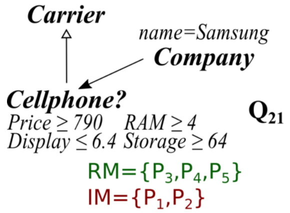
Generating RmE *and* RmL. Given a picky edge $e = (u_o, u')$, GenRx simply adds RmE(e) and for all literals $l \in F_Q(u_o)$ (resp. $F_Q(u')$), GenRx adds RmL(u_o, l) (resp. RmL(u', l)).

Generating RxL. Let adom(A, E_P) be the set of distinct values of $v.A$ with all $v \in$ RC that may become matches upon relaxing E_P. For end node u of a picky edge e=(u, u') and literals $l = (u.A_i$ op $c)$ of u, GenRx sorts and *discretizes* adom(A, E_P) that has values satisfying relaxed literal, and generates an atomic RxL with two rules below. (1) If op $\in \{<, \leq, =\}$, adds RxL(l, $u.A_i \leq a$) where a is the *smallest* value in adom(A, E_P) and $a > c$; (2) if op $\in \{>, \geq, =\}$, adds RxL(l, $u.A_i \geq a$), where a is the *largest* value in adom(A, E_P) and $a < c$. It generates RxL similarly for node u'.

The above rule applies to numerical attributes. For categorical attributes where attribute values are not comparable ($l = (u.A_i = c)$), it simply adds RmL(l) and leaves Q to be refined by AddL (to be discussed).

Example 5.3. Consider the relaxation operator o_3 shown in Fig 6. Since it can potentially identify $\{P_3, P_4\}$ as relevant matches, $\overline{\text{RC}}(o_3) = 2$. Similarly, o_2 can only identify P_3 due to a corresponding picky edge (Cellphone, Sensor), $\overline{\text{RC}}(o_2) = 1$. Thus o_3 is preferred to chase Q rather than o_2.

Generate Picky Refinement. Procedure NextOp identifies picky edges from $Q.\mathcal{S}$ as those with matches involving RM nodes in $Q.\mathcal{S}(G)$, and invokes a procedure GenRf to generate refinement operators ranked by their (estimated) pickiness. Similarly, as the relaxation counterpart, NextOp chooses refinement operator o that tends to maximally remove irrelevant matches (denoted as IM(o)) while minimally remove the relevant ones (denoted as RM(o)). It ranks the operator o by its pickiness $p'(o) = \frac{\lambda|\overline{\text{IM}}(o)| - \sum_{v \in \underline{\text{RM}}(o)} \text{cl}(v, \mathcal{E})}{|V_{u_o}|}$.



Star Table $T_{21}(G)$

	Cellphone-Carrier	Cellphone-Company
P_1	AT&T	Samsung
P_2	AT&T	Samsung
P_3	Sprint	Samsung
P_4	Sprint	Samsung
P_5	T-Mobile	Samsung

AddL(Carrier.discount=25%)

Figure 8: Picky operator generation by views

Here $\overline{\mathsf{IM}}(o)$ overestimates $\mathsf{IM}(o)$; and $\underline{\mathsf{RM}}(o)$ underestimates $\mathsf{RM}(o)$ by only counting RM nodes that are no longer candidates. One can verify that $p'(o) \geq \mathsf{cl}(Q \oplus \{o\}, \mathcal{E}) - \mathsf{cl}(Q, \mathcal{E})$.

Procedure GenRf. We show how GenRf generates AddL and RfE, and show the cases of AddE and RfL in Appendix.

Generating AddL. For each node u and match $v \in Q(u, G)$ reachable by some RM node v_o, and each attribute-value pairs $(A_i, a_i) \in f_A(v)$ that $A_i \notin F_Q(u)$, it enforces a single $\mathsf{AddL}(u, A = a_i)$ with the hope that irrelevant matches do not satisfy this constraint.

Generating RfE. For each edge $e \in E_Q$ with $L_Q(e) > 1$, GenRf introduces $\mathsf{RfE}(e, L_Q(e), L_Q(e) - 1)$.

Example 5.4. We illustrate how the star views and star tables are used to generate AddL in Fig. 8. GenRf generates AddL(Carrier, Discount=25%) since (Carrier.Discount, 25%) ∈ f_A(Sprint) and is not specified in F_Q(Carrier). As IM nodes $\{P_1, P_2\}$ no longer satisfy this new constraint, they will be excluded if AddL(Carrier, Discount=25%) is posed.

Completeness guarantee. We present a *completeness* property for optimal query rewrite, *i.e.*, it suffices to chase Q with the picky operators generated from GenRx and GenRf only, without the enumeration of Q-Chase (see Appendix).

5.4 Prune: Pruning Q-Chase Sequences

We finally introduce a major pruning strategy (Prune) guaranteed by Q-Chase in normal form. Observe that $\mathsf{cl}^+(Q, \mathcal{E}) = \frac{\sum_{v \in \mathsf{rep}(\mathcal{E}, Q)} \mathsf{cl}(v, \mathcal{E})}{|V_{u_o}|}$, *i.e.*, the closeness contributed by RM nodes of Q, is a valid upperbound for $\mathsf{cl}(Q, \mathcal{E})$. Moreover, for any Q-Chase sequence ρ in normal form, with "relaxation-only" subsequence ρ_1 followed by "refinement-only" subsequence ρ_2 we have the following.

Lemma 5.5. *For any query rewrite Q_1 and Q_2=$Q_1 \oplus O$, (1) if Q_2 is obtained in ρ_1 from Q_1, and $\mathsf{cl}^+(Q_2, \mathcal{E}) = \mathsf{cl}^*$ (theoretically optimal closeness), then $\mathsf{cl}(Q_2) \geq \mathsf{cl}(Q_3)$ for any Q_3=$Q_2 \oplus O'$, and O' is any set of relaxation operators; (2) if Q_2 is obtained in ρ_2 from Q_1, then $\mathsf{cl}^+(Q_1, \mathcal{E}) \geq \mathsf{cl}(Q_2, \mathcal{E})$.*

The result can be verified by definition of relaxation and refinement operators. It leads to pruning strategies below.

"Relaxation" or "Refinement"? Procedure NextOp reduces unnecessary Q-Chase extension by checking two conditions. Given Q=$Q_0 \oplus O$ and Q is visited in Q-Chase for the first time, the condition RefineCond (line 3) verifies whether: (1) there exists IM nodes in $Q(G)$ to be removed, and (2) $\mathsf{cl}^+(Q, \mathcal{E}) > \mathsf{cl}(Q^*)$. Putting together, it generates refinement operators at Q only when $\mathsf{cl}(Q, \mathcal{E})$ can be possibly improved by removing IM nodes (Lemma 5.5(2)). Similarly, it generates relaxation operators only when Q *is not refined earlier* and $\mathsf{cl}^+(Q) < cl^*$ (thus $\mathsf{cl}^+(Q)$ can be improved by relaxation (Lemma 5.5(1)), asserted by RelaxCond (line 7).

Early termination. Procedure Prune safely prunes the entire subtree of $\mathsf{Chase}(W, G, B)$ at node Q, whenever Q is at "refinement-only" sequences and $\mathsf{cl}^+(Q, \mathcal{E}) \leq \mathsf{cl}(Q^*)$. By Lemma 5.5(2), no query rewrite refined from Q has larger closeness than discovered $\mathsf{cl}(Q^*)$. Moreover, it terminates whenever $\mathsf{cl}(Q)$=cl^*, *i.e.*, theoretically optimal closeness.

Correctness and complexity. The correctness of AnsW follows from that it simulates the computation of Q-Chase (W, G, B) and by Theorem 4.3. Moreover, it correctly prunes query rewrites that are not optimal by Lemma 5.5.

AnsW has an anytime performance: it returns the best query rewrite identified so far upon request, and finds optimal answer given enough time. Given a verified query Q, the *delay time* it takes to simulate a Q-Chase step and updates Q is in $O(|Q.\mathcal{S}(G)|_{b_m}^{|Q|+1})$ time, where b_m is the largest number posed by a RxE, and $|Q.\mathcal{S}(G)|_{b_m}$ is the b_m-hop neighbors of the edges in star views in G. We found that AnsW has fast response time over large G (see Section 7). This also indicates feasible application of AnsW in exploratory graph search.

Generating Explanations. The lineage information associated to query rewrites produced by AnsW can be easily tracked by a structure called *differential table*. Given Q'=$Q \oplus O$ following a Q-Chase sequence, a differential table T_D is a set of triples $\langle e, o, V_d \rangle$, such that (1) e is a picky edge from NextOp, (2) o is a picky operator induced by e, and (3) $V_d \in Q'(G) \setminus Q(G)$ is either in $\mathsf{RM}(o)$ or $\mathsf{IM}(o)$. The table T_D can be easily maintained by tracing star tables.

Differential tables provide additional explanations for the suggested queries. Fig 6 illustrates a differential table T_D. For example, the first tuple tells us that by applying RmE on edge e=(Cellphone, Sensor), P_3 becomes a relevant match, due to the removal of e. This helps users to identify new exemplars, closing the loop of query-response-suggestion workflow in exploratory search (Fig. 3).

5.5 A Faster Heuristic

We next outline a variant of AnsW, which preserves anytime performance but with tunable performance.

Q-Chase with Beam Search. The faster and tunable heuristic, denoted as AnsHeu, performs a breadth-first *beam* *search* with beam size k. It explores at most k query rewrites extended from a query, induced by top-k operators ranked by their pickiness. At any time, NextOp generates at most k operators for each operator class (at most $8k$ operators) for each query rewrite. Moreover, each time a newly evaluated query rewrite Q' is added to $\mathcal{P}$, AnsHeu evicts other queries with lower closeness than Q' from $\mathcal{P}$, keeping $|\mathcal{P}| \leq k$.

AnsHeu does not return optimal answer, yet is a fast variant that often returns desirable query rewrites due to the effectiveness of picky operator generation. We found that it can improve the efficiency of AnsW by 4.87 times with 34% loss of answer quality compared to AnsW (Section 7).

6 EXTENSIONS

We next show that Q-Chase-based framework readily yield approximation algorithms for special cases of Why-questions, and extends to suggest top-k queries.

6.1 Approximating Q-Chase

Approximating Why-Many. Why-many question is a variant of Why-question [32], where the goal is *refining* Q to remove as many as possible matches from $Q(G)$ that are irrelevant to $\mathcal{E}$. An equivalent interpretation is to chase Q with "refinement-only" operators in Q-Chase. More formally, it is formulated as follows.

- **Input**: a graph G, a Why-Many question $W_M(Q, \mathcal{E})$ with original query Q, exemplar $\mathcal{E}$, and budget B;
- **Output**: an answer $Q'{=}Q \oplus O$ for $W_M(Q, \mathcal{E})$ such that (1) O contains refinement operators only and $c(O) \leq B$; (2) $Q'(G) \models \mathcal{E}$, and $\mathsf{cl}(Q', \mathcal{E})$ is maximized.

Fixed-parameter approximability. While WQE is hard to approximate (Theorem 3.2), we study its fixed-parameter approximability [18]. Real-world pattern queries are usually small [8]. For example, 97.2% of SWDF and 67% of DBPedia queries contain a single triple pattern. Under this practical assumption, we parameterize Why-Many questions with two parameters. (1) $k_1 = |Q|$: including its attribute size $|\mathcal{A}|$ and edge size; and (2) k_2: maximum size of active domain of an attribute in $\mathsf{rep}(\mathcal{E}, V)$. We have the following result.

Theorem 6.1. *Answering Why-Many question is fixed-parameter* $\frac{1}{2}(1 - \frac{1}{e})$*-approximable with fixed* k_1 *and* k_2.

We provide a constructive proof for Theorem 6.1 by presenting an approximation algorithm, denoted as ApxWhyM.

Approximation algorithm. The algorithm ApxWhyM (illustrated in Fig. 9) computes a refined query Q' without simulating a complete Q-Chase tree. It has two steps.

(1) Given query answers $Q(G)$ and its irrelevant matches $I(u_o)$ (initialized in line 1), it invokes a procedure SeedRf to identify a "seed" set of picky refinement operators O_s (line 2). For each seed operator o, it keeps track of a set of irrelevant matches $\mathsf{IM}(o)$ "covered" by o. That is, it ensures the removal of $\mathsf{IM}(o)$ from $Q'(G)$ for any Q' obtained by a Q-Chase sequence with o from Q.

```
Algorithm ApxWhyM
Input: a graph $G$, a Why-many question $W_M(Q, \mathcal{E})$, budget $B$,
Output: a query rewrite $Q'$ that answers $W_M$.
1.  $Q(G)$:= Match($Q, G$); set $I(u_o) = Q(G) \setminus \mathsf{rep}(\mathcal{E}, Q)$;
2.  set $O_s$ := SeedRf($Q, I(u_o), G, B$); $O_1 := \emptyset$;
3.  set $O_2 := \arg\max\{w(\{o\}) : o \in O_s, c(o) \leq B\}$;
    /*greedy selection of refinement operators*/
4.  while $O_s \neq \emptyset$ do
5.      $o^* := \arg\max\{\frac{w(O_1 \cup \{o\}) - w(O_1)}{c(o)} : o \in O_s\}$;
6.      if $c(O_1) + c(o^*) \leq B$ then
7.      $O_1 := O_1 \cup \{o^*\}$; $O_s := O_s \setminus \{o^*\}$;
8.      if $c(O_1) \geq B$ then break ;
9.  $O' = \arg\max_{i \in [1,2]} \mathsf{cl}(O_i)$;
10. construct query $Q' = Q \oplus O'$;
11. return $Q'$;
```

Figure 9: Algorithm ApxWhyM

(2) Once O_s is identified, it solves a budgeted weighted set cover problem, treating each operator o as a weighted set that contains $\mathsf{IM}(o) \subseteq I(u_o)$. To approximate optimal closeness, it assigns a weight to each o for $Q' = Q \oplus \{o\}$ as $w(o) = \mathsf{cl}(Q'(G), \mathcal{E}) - \mathsf{cl}(Q(G), \mathcal{E})$ to quantify the gain of closeness by rewarding the removal of $\mathsf{IM}(o)$ and penalizing the elimination of $\mathsf{RM}(o)$.

The rest of ApxWhyM follows a greedy selection strategy, which iteratively updates set O_1 with an operator having maximized closeness gain, compared with set O_2 initialized by the first best operator. The process repeats until O_1 exceeds the cost bound (line 9) or all the seed operators are processed (line 4). The query rewrite Q' is constructed with the one having better closeness (lines 9-11).

Due to the limited space, we defer procedure SeedRf and performance analysis in the Appendix.

Answering Why-Empty Questions. Given a query Q with *no relevant matches* to exemplar $\mathcal{E}$, a *Why-Empty question* (or differential query [32]) aims to revise Q such that at least a relevant match is identified. The answer to Why-Empty questions helps bridge missing answers and user search intention. We focus on clarifying empty answers for star queries using pragmatic operators including RmL and RmE. Such queries are quite common. For example, 99.7% of DBPedia and SWDF queries are star-shaped [8].

Lemma 6.2. *Answering removal-only Why-Empty questions is in* $O(|Q||\mathsf{rep}(\mathcal{E}, V)||V|)$ *time.*

Algorithm. We outline the algorithm, denoted as AnsWE. The main idea is to perform a *guided* Q-Chase from Q, with atomic operators whose removal introduce new matches that are either directly in $\mathsf{rep}(\mathcal{E}, V)$, or are on a path from rep($\mathcal{E}, V$) to the node matches $Q(u, G)$ for some pattern node u. To this end, it (1) generates a set of RmL operators that relax the predicates at u_o, (2) produces a set of picky edges adjacent to u_o, and (3) generates a set of paths (with at most two edges) between u_o to a leaf node in Q. It then generates a set of RmL and RmE operators over picky edges and picky paths, along with their associated RC sets. Finally, it finds a relevant candidate that is associated with at most B generated relaxation operators (see details in Appendix).

6.2 Top-k Query Suggestion

The algorithm AnsW can be readily extended to return top-k query rewrites, preserving the optimality guarantee. To this end, one only need to modify its early termination condition (procedure Prune). Instead of comparing cl(Q^*) (the current optimal closeness) with the estimated closeness cl$^+(Q, \mathcal{E})$ at refinement-only stage, we maintain Q^*_k as the k^{th} highest closeness, and prune the entire sequence at refinement-only stage, whenever cl$^+(Q, \mathcal{E}) \leq$ cl(Q^*_k).

7 EXPERIMENTAL EVALUATION

Using real-life attributed graphs, we conduct experiments to evaluate the effectiveness and efficiency of our algorithms.

Experimental setting. We use the following setting.

Datasets. We use three real-life data graphs and a benchmark dataset. (1) DBpedia[1] consists of 4.86M entities and 15M edges, where each entity carries one of the 676 labels (*e.g.*, Person, Building), and on average 9 attributes, (2) IMDB[2], contains 1.7M nodes (*e.g.*, movies) and 5.2M edges (*e.g.*, acted in, rating). Each node has on average 6 attributes (*e.g.*, rating, co-actors), and (3) Offshore[3] contains 839K offshore entities (*e.g.*, companies, countries, person), 3.6M relationships (*e.g.*, establish, close) and 433 labels. It has on average 4 attributes and covers 40 years of offshore entities and financial activities. We also use a benchmark synthetic knowledge graph denoted as WatDiv [3] with 521K nodes and 9.1M edges. It contains e-commerce information *e.g.*, Purchase and Retailer.

"Ground truth" Queries. We adopt two query benchmarks. For DBpedia, we use DBPSB benchmark [20] that simulates real-world query workload. We first identify 40 query templates from DBPSB, where each DBPSB template carries a focus node originally carried from its SPARQL format. We then instantiate each template by assigning node labels from the candidates of its focus node, and up to 3 search predicates per node, including those from the benchmark queries. Similarly, we instantiate 20 templates from WatDiv [3].

For IMDB and Offshore, we implement a query generator that generates diversified queries, following WatDiv query generator [3]. For each query, we ensure that it has isomorphic answer in G (except for Why-Empty questions) and randomly select a focus node.

We consider each query Q^* instantiated from the benchmark as a "ground truth" query. Each such query has a desired answer $Q^*(G)$ induced by subgraph isomorphism.

Generating Why-Questions. Given a ground truth query Q^* and answer $Q^*(G)$, we "disturb" Q^* by injecting up to 5 atomic operators to create a query Q. We then set $\mathcal{T}$ = $Q^*(G) \setminus Q(G)$, and C=$\emptyset$ (unless declared by users; see Exp-5). We construct Why questions with input Q, $Q(G)$ and $\mathcal{E}$=($\mathcal{T}, C$). By default, we set budget B = 3. We observed that setting B close to real-world query size [8] suffices to obtain query rewrite with high quality, as verified by our tests.

Algorithms. We implemented Q-Chase-based algorithms, all in Java: (1) Exact (anytime) algorithms: AnsW (including caching and pruning strategies), with two variants: (a) AnsWnc, its variant without caching; and (b) AnsWb, by removing both pruning strategies and caching; (2) Heuristics: AnsHeu that applies tunable beam search, and AnsHeuB, a variant of AnsHeu that replace picky operator generation with random selection; (3) Tractable algorithms ApxWhyM and AnsWE, for Why-Many and Why-Empty questions, respectively. (4) While existing query relaxation methods do not consider examples, and query-by-example cannot suggest queries, we implemented FMAnsW, a frequent pattern mining algorithm that applies the methods in [21], to suggest queries as frequent subgraph patterns around V_{u_o}.

For a fair comparison, all the algorithms use a star-based query processing and access a fast distance index [2].

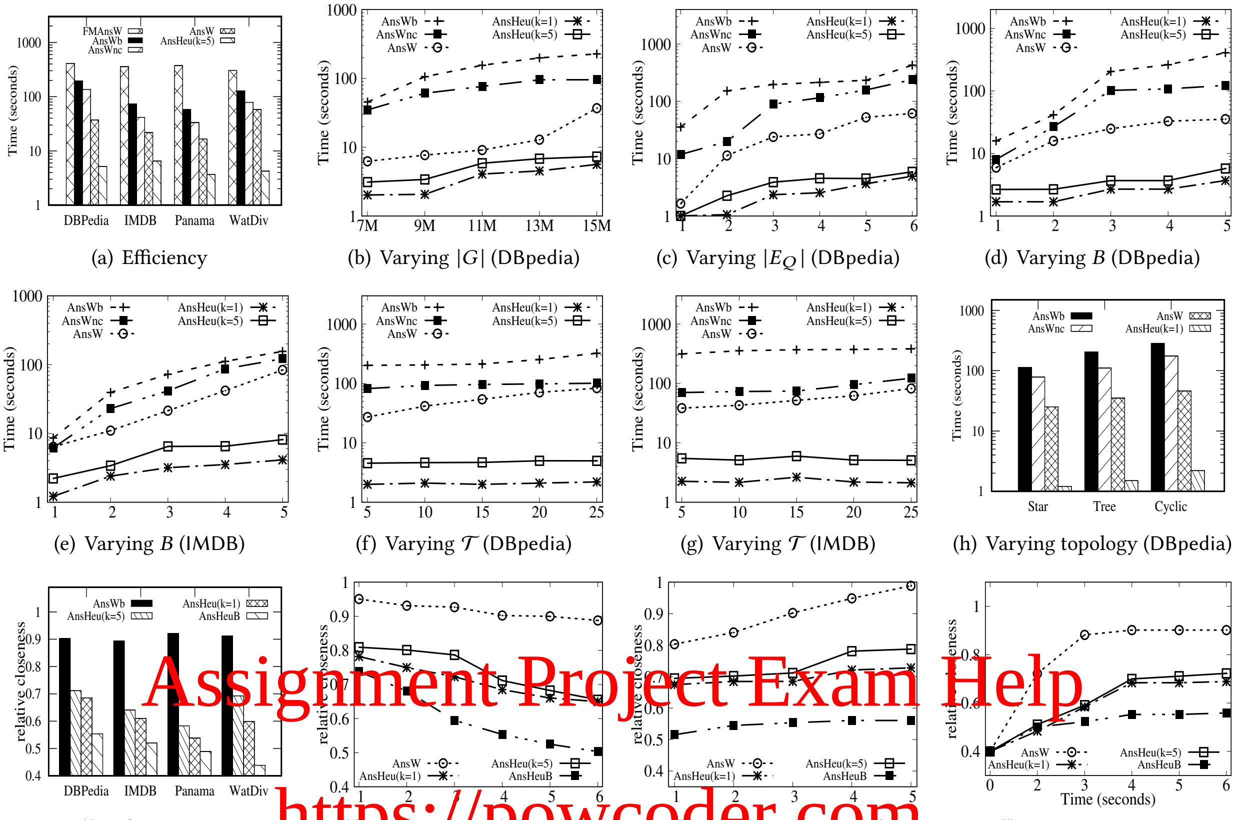
We deployed the algorithms on a machine powered by an Intel Xeon processor with 2.3GHz. Each test uses 50 queries and is repeated 5 times, and the average result is reported.

Experimental results. We next report our findings.

Exp-1: Efficiency. We first evaluate the efficiency of our algorithms, including AnsHeu, AnsW, and AnsWnc, compared with AnsWb and FMAnsW. Fig. 10(a) verifies the following. (1) It is quite feasible to answer Why-questions for large graphs. AnsW takes on average 37, 21, 16, and 57 seconds on DBpedia, IMDB, Offshore, and WatDiv, respectively. (2) The optimization techniques significantly improve the efficiency of the algorithms. On average, AnsW outperforms FMAnsW, AnsWb, and AnsWnc by 10.82, 3.41, and 2.1 times, respectively. We also observe that AnsHeu quickly converges to near-optimal solutions (on average 4 seconds), due to picky operator generation and pruning.

Scalability. We sample 5 versions of DBpedia with edge size varied from 7M to 15M. Fig. 10(b) verifies that both AnsW

[1] http://dbpedia.org
[2] https://www.imdb.com/interfaces/
[3] https://offshoreleaks.icij.org/



(a) Efficiency (b) Varying $|G|$ (DBpedia) (c) Varying $|E_Q|$ (DBpedia) (d) Varying B (DBpedia)

(e) Varying B (IMDB) (f) Varying $\mathcal{T}$ (DBpedia) (g) Varying $\mathcal{T}$ (IMDB) (h) Varying topology (DBpedia)

(i) Effectiveness (j) Varying $|E_Q|$ (DBpedia) (k) Varying B (DBpedia) (l) Time vs. Accuracy

Figure 10: Answering WQE: Efficiency and Effectiveness

and AnsHeu scale well with $|G|$, and are less sensitive compared with AnsWb, due to star view based optimizations.

Varying query size. Varying number of edges $|E_Q|$ from 1 to 6 (with fixed budget B=3), we report the efficiency of our algorithms over DBpedia in Fig. 10(c). While all the algorithms take more time to verify larger queries, AnsW and AnsHeu are less sensitive to the query size. This is because the star views reduce the cost of evaluating new query rewrite whenever possible. For example, AnsHeu returns query rewrites in only 5 seconds when $|E_Q|$=6.

Varying cost bound. Varying the budget B from 1 to 5, Fig. 10(d) and Fig. 10(e) show that all the algorithms take more time to consume a larger budget over DBpedia and IMDB, respectively. AnsHeu is the least sensitive algorithm, since it does not backtrack Q-Chase sequences as others.

Varying exemplars. We evaluate the impact of changing the size of $\mathcal{T}$ in Why-questions (which contains entities from G). We simulate exploratory search by adding more number of exemplars in each session such that $\mathcal{T}$ is varied from 5 to 25. As shown in Fig. 10(f) and Fig. 10(g), all the algorithms but AnsHeu require more time with larger $\mathcal{T}$ over both DBpedia and IMDB. Indeed, it needs to inspect more query rewrites triggered by picky operators from larger $\mathcal{T}$. AnsHeu is relatively less sensitive, as it spawns up to a fixed number of Q-Chase steps without backtracking.

Varying Topology. Fig. 10(h) tells us that it takes less time to answer Why-questions for star queries than for trees and cyclic ones. The latter two decompose to more stars in the procedure Match, and take more time for joining star views.

Exp-2: Effectiveness of answering Why-questions. Given a benchmark query Q^* and answer $Q^*(G)$ as "ground truth", and a query rewrite Q' reported by an algorithm, we define a *relative closeness* as $\delta(Q', Q^*) = \frac{\mathrm{cl}(Q^*(G),\mathcal{E})}{\mathrm{cl}(Q'(G),\mathcal{E})}$, which measures how well Q' "recovers" the true answer of the ground truth query. The higher δ is, the better. Interestingly, as the ground truth Q^* guarantees optimal closeness, $\delta(Q', Q^*)$ degrades to $\frac{|Q'(u_o)\cap Q^*(u_o)|}{|Q'(u_o)\cup Q^*(u_o)|}$, *i.e.,* the *Jaccard coefficient* of the answers, which justifies our metric. Note that δ is computable only when Q^* is known. Our algorithms optimize $\mathrm{cl}(Q'(G),\mathcal{E})$ as an absolute measure.

Fig. 10(i) reports the relative closeness achieved by the algorithms within cost bound B. (1) AnsW constantly achieves

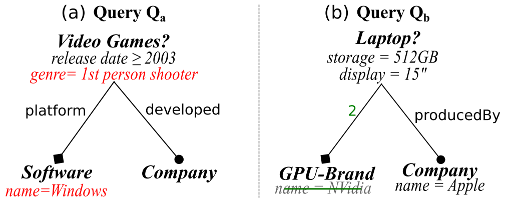


Figure 11: Case study: real-world queries

the maximum closeness among others, which also verifies our closeness measure cl(·). (2) Increasing the beam size for algorithm AnsHeu consistently improved the closeness over all the datasets. Moreover, it can suggest good query rewrites with relative closeness on average 66%.

Varying query size. We report the impact of $|E_Q|$ (varied from 1 to 6 by injecting more operators to ground truth queries) to the relative closeness in Fig. 10(j). All the algorithms achieve higher closeness for smaller queries. Indeed, the answers $Q(G)$ usually departed "further" from ground truth $Q^*(G)$ as more operators are injected to Q^*, making it more difficult to find good query rewrites under fixed budget. On the other hand, AnsW and AnsHeu are able to preserve 89% and 71% (with beam size 1) of relative closeness over all cases, respectively. Moreover, the relative closeness is improved to 80% with beam size up to 5 for AnsHeu. These indicate their ability in recovering true answers in practice.

Varying cost bound. Fig. 10(k) tells us that all algorithms achieve better relative closeness with larger budget B. The exact algorithm achieves optimal closeness when B=5. This is consistent with the changes we injected to ground truth.

Exp-3: Anytime performance. We measure $\delta_t = \delta(Q'_t, Q^*)$ where Q'_t is the query rewrite at time t discovered by an algorithm. Setting a time bound $t = 6$ seconds, Fig. 10(l) shows that AnsW converges to high relevant closeness fast. For example, it achieves δ_t above 90% in less than 4 seconds. AnsHeuB takes longer time to achieve the same relative closeness, mainly due to its lack of backtracking.

Exp-4: Answering Why-Empty and Why-Many. We also evaluate the efficiency and effectiveness of answering Why-Many and Why-Empty questions. Due to limited space, we report the results (Exp-4) in Appendix.

Exp-5: User Study. To further verify the practical application of Q-Chase algorithms, we conducted a user study that involves 5 professional users in graph databases/SQL search and 5 graduate students (end users), investigating search scenarios over DBpedia and a private dataset from our industry partner with real query logs. We revised AnsW to return top-k query rewrites. For each dataset, the users are requested to issue 3 initial queries. If not satisfied with the answer, they provide examples, and (1) re-rank the top-3 query rewrites from AnsW based on answer relevance, and (2) label the "desired matches" from the returned answers.

Effectiveness. We report the following. (1) The discounted cumulative gain at 3 (nDCG_3) from user's ranked list and the one returned by AnsW is on average 0.71. This demonstrates good consistency with the user's ranking in terms of answer relevance. (2) The precision, defined as the ratio of the manually labeled relevant entities in the answers of query rewrite, is on average 0.76. This indicates the good capacity of AnsW in recovering relevant entities for real queries.

Case analysis. We illustrate two real-world queries issued by users and suggested query rewrite in Fig. 11.

(1) The first query Q_a searches for video games released after 2003. While 161 video games were retrieved, he provided "call of duty" as an example. The best query rewrite (consistent with user feedback) Q'_a modified Q_a with additional constraints on genre and operating system (color-coded predicates). The answer $Q'_a(G)$ contains 5 games (*e.g.*, "Doom 3") that narrow the interested answers down to "first-person shooting" games on Windows.

(2) The second query Q_b from the query log of our industry partner searches for recent computer models (after 2018) with constraints on screen size, memory and GPU model, yet returns no answer. An end user in our study sets $\mathcal{T}$ = {MR942CH/A}, a model id she is aware of, and wonders why similar laptops are not among the results. AnsW suggested a query rewrite Q'_b that identifies 4 similar MacBook models such as (e.g., MR942LL/A) verified as desired products, by revising Q_b with RmL (name=NVidia) and RxE (Laptop,Brand,1,2). The differential table (not shown) further reveals that relaxing GPU and connectivity constraints finds desired laptops powered by either Intel or AMD GPU.

8 CONCLUSION

We have formulated the general problem of answering Why-questions for graph pattern queries. We made case for entity search with P-homomorphism, and approached query rewriting to clarify unexpected answers. We proposed Q-Chase, a symbolic representation that extends conventional Chase, and showed the problem is intractable and hard to approximate. We introduced Q-Chase-based algorithms to efficiently implement query chasing process, from fixed parameter approximations to feasible optimal solutions. We experimentally verified the efficiency and effectiveness of Q-Chase-based techniques and its application.

Acknowledgments. This work is supported in part by USDA/NIFA 2018-67007-28797, UI-ASSIST DE-IA0000025, and Siemens. A part of this work has been done when Mohammad and Shengqi were in JD AI Research.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[2] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks. In *SIGMOD.*

[3] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified stress testing of RDF data management systems. In *ISWC.*

[4] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *ICDT.*

[5] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. 2010. Graph homomorphism revisited for graph matching. *VLDB* (2010), 1161–1172.

[6] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Diversified top-k graph pattern matching. *VLDB* (2013), 1510–1521.

[7] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional dependencies for graphs. In *SIGMOD.* 1843–1857.

[8] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. 2011. An empirical study of real-world SPARQL queries. In *USEWOD workshop.*

[9] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. 2010. A survey of graph edit distance. *Pattern Analysis and applications* (2010), 113–129.

[10] Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.*

[11] Md Saiful Islam, Chengfei Liu, and Jianxin Li. 2015. Efficient answering of why-not questions in similar graph matching. *TKDE* (2015).

[12] Md Saiful Islam, Chengfei Liu, and Rui Zhou. 2012. User feedback based query refinement by exploiting skyline operator. In *ER.*

[13] Nandish Jayaram, Sidharth Goyal, and Chengkai Li. 2015. VIIQ: auto-suggestion enabled visual interface for interactive graph query formulation. *VLDB* (2015), 1940–1943.

[14] Nandish Jayaram, Arijit Khan, Chengkai Li, Xifeng Yan, and Ramez Elmasri. 2015. Querying knowledge graphs by example entity tuples. *TKDE* (2015), 2797–2811.

[15] Gjergji Kasneci, Fabian M Suchanek, Georgiana Ifrim, Maya Ramanath, and Gerhard Weikum. 2008. Naga: Searching and ranking knowledge. In *ICDE.*

[16] Samir Khuller, Anna Moss, and Joseph Seffi Naor. 1999. The budgeted maximum coverage problem. *Inform. Process. Lett.* 70, 1 (1999), 39–45.

[17] Phokion G Kolaitis and Madhukar N Thakur. 1994. Logical definability of NP optimization problems. *Information and Computation* (1994).

[18] Dániel Marx. 2008. Parameterized complexity and approximation algorithms. *Comput. J.* 51, 1 (2008), 60–78.

[19] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F Moore, and Dan Suciu. 2009. Why so? or why no? functional causality for explaining query answers. *arXiv preprint arXiv:0912.5340* (2009).

[20] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. 2011. DBpedia SPARQL benchmark–performance assessment with real queries on real data. In *ISWC.*

[21] Davide Mottin, Francesco Bonchi, and Francesco Gullo. 2015. Graph query reformulation with diversity. In *KDD.*

[22] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2014. Exemplar queries: Give me an example of what you need. *VLDB* (2014), 365–376.

[23] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2016. Exemplar queries: a new way of searching. *VLDB* (2016), 741–765.

[24] Mohammad Hossein Namaki, Yinghui Wu, Qi Song, Peng Lin, and Tingjian Ge. 2017. Discovering Graph Temporal Association Rules. In *CIKM.* 1697–1706.

[25] Mohammad Hossein Namaki, Yinghui Wu, and Xin Zhang. 2018. GExp: Cost-aware Graph Exploration with Keywords. In *SIGMOD.*

[26] Alexandra Poulovassilis. 2018. Applications of Flexible Querying to Graph Data. In *Graph Data Management.* 97–142.

[27] L Rocach and O Maimon. 2005. Clustering methods Data mining and knowledge discovery handbook. *Springer US* (2005), 321.

[28] Qi Song, Mohammad Hossein Namaki, and Yinghui Wu. 2019. Answering Why-Questions for Subgraph Queries in Multi-Attributed Graphs. In *ICDE.*

[29] Yinglong Song, Huey Eng Chua, Sourav S Bhowmick, Byron Choi, and Shuigeng Zhou. 2018. BOOMER: Blending Visual Formulation and Processing of P-Homomorphic Queries on Large Networks. In *SIGMOD.*

[30] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. Sqlgraph: An efficient relational-based property graph store. In *SIGMOD.*

[31] Quoc Trung Tran and Chee-Yong Chan. 2010. How to conquer why-not questions. In *SIGMOD.*

[32] Elena Vasilyeva, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner. 2016. Answering "why empty?" and "why so many?" queries in graph databases. *J. Comput. System Sci.* 82, 1 (2016).

[33] Elena Vasilyeva, Maik Thiele, Adrian Mocan, and Wolfgang Lehner. 2015. Relaxation of subgraph queries delivering empty results. In *SSDBM.*

[34] Meng Wang, Jun Liu, Bifan Wei, Siyu Yao, Hongwei Zeng, and Lei Shi. 2018. Answering why-not questions on SPARQL queries. *KAIS* (2018).

[35] Mohamed Yahya, Klaus Berberich, Maya Ramanath, and Gerhard Weikum. 2016. Exploratory querying of extended knowledge graphs. *VLDB* (2016), 1521–1524.

[36] Shengqi Yang, Fangqiu Han, Yinghui Wu, and Xifeng Yan. 2016. Fast top-k search in knowledge graphs. In *ICDE.*

[37] Shengqi Yang, Yinghui Wu, Huan Sun, and Xifeng Yan. 2014. Schemaless and structureless graph querying. *VLDB* (2014), 565–576.

[38] Lei Zou, M Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. 2014. gStore: a graph-based SPARQL query engine. *VLDBJ* 23, 4 (2014), 565–590.

APPENDIX A: PROOFS

Proof of Lemma 2.2. We describe a procedure to compute $\mathsf{rep}(\mathcal{E}, V)$ and verify whether $V \models \mathcal{E}$ as follows. Given an exemplar $\mathcal{E}$=$(\mathcal{T}, C)$, for each tuple pattern $t \in \mathcal{T}$, it computes $\mathsf{rep}(t, V)$ as the set of nodes v, where $\mathsf{vsim}(v, t)$ is true (*i.e.,* $v \sim t$), and sets $\mathsf{rep}(\mathcal{E}, V)$=$\bigcup_{t \in \mathcal{T}} \mathsf{rep}(t, V)$. It then enforces C over $\mathsf{rep}(\mathcal{E}, V)$ by processing a selection query defined on a table that consists of the node tuples from $\mathsf{rep}(\mathcal{E}, V)$, with a conjunct condition $\bigwedge_{l \in C} l$, and update $\mathsf{rep}(\mathcal{E}, V)$ with only those selected ones. By definition, $V \models \mathcal{E}$ if $\mathsf{rep}(\mathcal{E}, V) \neq \emptyset$. This can be processed in $O(|V||\mathcal{E}| + |V|^2)$ time.

Proof of Theorem 3.2. Given a Why question $W(Q, \mathcal{E})$, graph G, budget B and threshold γ, the decision version of WQE is to decide whether there exists a query rewrite $Q' = Q \oplus O$ such that $Q'(G) \models \mathcal{E}$, $c(O) \leq B$ and $\mathsf{cl}(Q') \geq \gamma$.

(1) NP-**hardness**. We show a strong result that WQE is already NP-hard when only relaxation operators are allowed, by a reduction from subgraph isomorphism. Consider two graphs G_1=(V_1, E_1, L_1) and $G_2 = (V_2, E_2, L_2)$, where each

node $v_1 \in V_1$ (resp. $v_2 \in V_2$) has a label $L_1(v_1)$ (resp. $L_2(v_2)$). The subgraph isomorphism is a bijective function from V_1 to a set $V_2' \subseteq V_2$ that preserves node label equality and edges. We construct a pattern query Q_3 with focus u_o that has (a) a pattern Q_3' that is isomorphic to G_1, where each pattern node u, corresponding to node v in V_1, carries a literal $u.A_L = L(v)$, each edge (u, u') has label 1, and there is a designated focus u_o in V_1; and (b) a pattern edge (u_o, u_s) with label 1 from u_o to a new pattern node u_s not in Q_3', which carries a literal $u_s.A_L = L'$, and L' is not seen in G_2. We obtain a graph G from G_2 by setting an attribute $v.A_L{=}L_2(v)$ for each node v in V_2. Let $\mathcal{E}$ be the node tuple t with $t[L_A]{=}L(u_o)$ and $C{=}\emptyset$, budget $B{=}1$, and $\gamma{=}\frac{1}{|V_2|}$. Then there exists an answer of Why-question $W(Q_3, \mathcal{E})$ exists (by removing edge (u_o, u_s)) if and only if there exists a subgraph isomorphism from G_1 to G_2. As subgraph isomorphism is NP-hard, WQE with RmE operators only is already NP-hard.

(2) **Approximation hardness**. We show that answering WQE is not constant-approximable unless $P = NP$. To prove this, we construct an approximation factor preserving (AFP) reduction from minimum 3DNF satisfiablity (MinSAT) problem [17] known to be not in APX. An instance I_1 of MinSAT is of the form $C_1 \vee \ldots \vee C_n$. Each clause C_j ($j \in [1, n]$) is of the form $y_{j_1} \wedge y_{j_2} \wedge y_{j_3}$, and each y_{j_i} ($i \in [1, 3]$) is a variable from a set $\{x_1, \ldots, x_m\}$. It is to find a truth assignment for $C_1 \vee \ldots \vee C_n$, such that a minimum number of clauses are satisfied. We provide the reduction below.

(1) Given an instance I_1 of MinSAT, function f outputs an instance I_2 of the WQE problem as follows. (a) It constructs Q as a single edge (u_o, u'), where $L(u_o){=}l_o$, $L(u') = l'$, and for each variable x_i ($i \in [1, m]$), it adds a literal $x_i = \perp$ to u_o, where $\perp$ can be matched by any value of x_i. (b) It adds a node $v_j \in G$ with $L(v_j) = l_o$ for each clause C_j ($j \in [1, n]$). Each node v_j has three attributes corresponding to the three variables of C_j. It adds another node v' to G with $L(v') = l'$, and connects it to all the nodes added to G. (c) It sets cost bound $B = m$ (number of variables), and define operator RfL to resolve the wildcard predicates of u_o to a truth value to be assigned to its variable. Let $\mathcal{E}$ be a tuple pattern such that $t[L]{=}l'' \neq l_o$. For Q to have maximized closeness, it needs to have as least irrelevant matches in G as possible.

(2) Given an answer Q' and its answer $Q'(G)$, a function g simply outputs the refined literals $F_Q'(u_o)$.

The construction (f, g) is an AFP-reduction from MinSAT to WQE. Moreover, we can verify that a clause C_j is satisfied by the truth assignment $F_{Q'}(u_o)$ if and only if its corresponding node v_j is in $Q'(u_o, G)$. Assume that for some $\epsilon > 0$, there is an ϵ-approximation algorithm for WQE, then A guarantees a constant-approximation for MinSAT. As MinSAT is not in APX, WQE is also not in APX.

Proof of Lemma 4.1. A procedure transforms canonical sequence to their equivalent normal forms by separating the relaxation and refinement operators into two disjoint sets O_1 and O_2, respectively. It then sorts the operators of O_1 by first putting RxL, RxE, and RmL followed by RmE's in a sequence ρ_1. This ensures that the operators are applicable to their previous step. Similarly, it sorts the operators O_2 as a sequence ρ_2 by first putting AddE followed by AddL, RfE, and RfL which ensures the applicability. It then returns a concatenation of ρ_1 followed by ρ_2 as a single sequence ρ'. This is doable since a valid canonical sequence ρ (1) does not contain *redundant operators e.g.,* multiple relaxation/refinement on the value/bound of the *same* literal/edge are presented by a single operator; and (2) does not contain *conflicting operators* on the *same* node/edge of the query that cancel-out each other *e.g.,* AddE (u, u', b), RmE (u, u', b); Thus, there exists an equivalent Q-Chase sequence ρ' in a normal form.

Proof of Theorem 4.3. We provide the following analysis.

(1) To see **If**, assume there is a terminal Q-Chase sequence from the root with result $(Q_k, \mathcal{E})$. As ρ is a sequence of valid Q-Chase steps, each step that chases Q from $(Q_i, \mathcal{E}_i)$ to $(Q_j, \mathcal{E}_j)$ ensures the invariant that $Q_j(G) \models \mathcal{E}_j$. Thus $Q_k(G) \models \mathcal{E}$ and $c(O){=}c(\rho) \leq B$ for $Q_k{=}Q \oplus O$. Thus Q_k is an answer for Why-question.

(2) The **Only If** states that for any answer Q_k of $W(Q, \mathcal{E})$, there exists a path ρ from the root with a result $(Q_k, \mathcal{E})$ in Q-Chase (W, G, B). We prove this by contradiction. Let $Q_k{=}Q \oplus O$, where O can be easily obtained by comparing Q and Q'. We construct a Q-Chase sequence ρ accordingly, and transform it to an equivalent normal form ρ'. Assume ρ' is not in Q-Chase (W, G, B). Then either ρ' is not valid, or it is not terminal. If ρ is not valid, then there exists a Q-Chase step from $(Q_i, \mathcal{E}_i)$ to $(Q_j, \mathcal{E}_j)$ for which $Q_j \not\models \mathcal{E}_j$ with operator o (which is known to be applicable). If it is not terminal, then $c(\rho) > B$. Either case leads to the contradiction that Q_k is an answer for $W(Q, \mathcal{E})$.

Given the above analysis, the optimality guarantees in Theorem 4.3 (2) follows.

Proof of Theorem 6.1: Approximability. We provide details of approximation ratio of Algorithm ApxWhyM by providing the reduction to budgeted maximum weighted coverage problem [16]. We consider each operator o generated from the seed set O_s as a subset of a universal set $U{=}V_{u_o}$, where each refinement operator o can "cover" a subset of U as $Q(u_o, G) \setminus (Q \oplus \{o\})(u_o, G)$ *i.e.,* the eliminated elements due to adding operator o. The weight of each element $v \in \mathsf{rep}(\mathcal{E}, Q)$ computed as $-\mathsf{cl}(v, \mathcal{E})$ and the element $v \in Q(u_o, G) \setminus \mathsf{rep}(\mathcal{E}, Q)$ assigned as $+\lambda$. Thus, the weight of a single o (as the subset) is computed as $\mathsf{cl}(Q'(G), \mathcal{E}) - \mathsf{cl}(Q(G), \mathcal{E})$ where $Q' = Q \oplus \{o\}$, and the cost of the subset is $c(o)$. Setting B as budget for subset selection, the problem is to find a collection O^* of operators to construct $Q^* = Q \oplus \{O^*\}$ such that it maximizes the total weight $\mathsf{cl}(Q^*(G), \mathcal{E}) - \mathsf{cl}(Q(G), \mathcal{E})$. Similar to the coverage problem where a subset is enough to cover an element, in here, an operator is enough to eliminate a candidate and thus removing its effect from $\mathsf{cl}(Q^*(G), \mathcal{E})$. Indeed, maximizing the constructed budgeted set cover problem is equivalent to maximizing the $\mathsf{cl}(Q^*(G), \mathcal{E}) - \mathsf{cl}(Q(G), \mathcal{E})$ wherein $\mathsf{cl}(Q(G), \mathcal{E})$ is constant and thus $\mathsf{cl}(Q^*(G), \mathcal{E})$ is maximized.

ApxWhyM simulates a greedy algorithm to ensure a $1 - \frac{1}{e}$ approximation [16] by greedily selecting operators with maximized $\frac{w(O \cup \{o\}) - w(O)}{c(o)}$. As the size of queries a fixed small parameter k_1 with small domain size k_2, which is a practical assumption for real-world queries [8], ApxWhyM ensures a fixed-parameter approximation.

Proof of Lemma 6.2. The algorithm AnsWE considers each literal and edge of the query Q as a single "atomic condition" that may be responsible for elimination of a node $v \in R(u_o)$ from $Q(u_o, G)$. To this end, it creates a set of smaller fragments and evaluates them as follows. (1) for each literal $l \in F_Q(u_o)$, it creates a single node u_o with a single predicate $l \in F_Q(u_o)$. (2) for each node $u \neq u_o$, it creates a single edge (u, u_o) with the edge bound as the distance between u and u_o in Q. (3) For each literal $l \in F_Q(u)$, it generates a copy of created single edge (u, u_o) and adds the single literal l. Clearly, each generated fragment represents an atomic condition either a literal (case 1 and 3) or an edge (case 2) that can be relaxed by an operator either RmL or RmE respectively. One can verify that (1) a relevant candidate of Q is not *RM* for *at least one* of the generated fragments; and (2) the relevant candidate does not become an *RM* in Q unless all such fragments are removed from the query (given RmL and RmE as the only relaxation operators). That is, for a $v \in RC$ to be a match, all of its picky conditions should be relaxed. Thus, AnsWE can associate each $v \in RC$ to a subset of generated fragments (and consequently their relax operators) that v is not *RM*. A node $v \in RC$ can be an *RM* for Q if the total cost of associated operators is within B.

The number of generated fragments are bounded by $|Q|$. Given a distance index that verifies the distance of two nodes, a fragment can be evaluated in at most $O(|\mathsf{rep}(\mathcal{E}, V)||V|)$. Thus, in total the complexity of AnsWE is $O(|Q||\mathsf{rep}(\mathcal{E}, V)||V|)$.

APPENDIX B: OPTIMIZATION

We provided examples of picky operator generation in Section 5. In here, we introduce more details of picky operator generation and illustrate remaining 3 operator types.

Generating Relaxation Operators. The goal of generating relaxation operators is to convert relevant candidates IC that are not currently in $Q(G)$ to RM. Thus, the picky operators refer to those that relax the conditions to cause the addition of at least a relevant match.

Identifying picky edges. NextOp first identifies picky edges (u, u'), which have verified matches or involve candidates that can reach any RC nodes, thus may bring new matches if the conditions it poses are relaxed. More specifically, for each pattern edge (u, u') in Q, NextOp consults the star tables $Q.\mathcal{S}(G)$ and decides whether (u, u') is a picky edge (maintained by a set E_P) as follows. (1) Assume *w.l.o.g.* u=u_o, and there exists a RC node not in any star table of $Q.\mathcal{S}(G)$ (not a match), but is reachable to any seen matches of u' in a table, NextOp adds (u, u') to E_P. (2) If neither u nor u' is u_o, but there is a star table with candidates of u or u' that can reach an RC node via candidates that follows a path $\{(u, u'), (u', u_o)\}$ or vice versa, NextOp adds (u, u') and (u', u_o) to E_P. This generates a set of RC nodes that may be introduced as matches, denoted as RC(E_P).

Generating picky operators. For each picky edge (u, u') in E_P, procedure GenRx generates four types of relaxation operators following predefined optimized rules. For example, it generates RxE operators as follows.

A maximum edge bound b_m for edge operator RxE is posed. Thus, RxE relaxes the edge bound $L_Q(e)$ up to b_m. Let $\mathsf{dist}(u, u')$ be the sum of the edge bounds on a shortest path in Q from u to u' that does not contain e. If $L_Q(e) < b_m$, RxE((u, u'), $L_Q(e)$, $L_Q(e) + 1$) is added.

Generating Refinement Operators. In contrast to generating relaxation operators, the goal of refinement is to remove irrelevant matches IM from $Q(G)$ as much as possible. Thus, the picky operators refer to those responsible of removing irrelevant matches.

Identifying picky edges. An edge (u, u') is picky for refinement conditions when it directly involves RM nodes, or has matches whose removal cause the removal of IM nodes. Thus, for each pattern edge (u, u') in Q, NextOp checks the following. (1) Assume *w.l.o.g.* u=u_o, and there exists a RM node in $Q.\mathcal{S}(G)$ seen in the entry for u, NextOp adds (u, u') to picky edge set E_P. (2) If neither u nor u' is u_o, but there is a star table with matches of u or u' that can reach an RM node via matches that follows a path $\{(u, u'), (u', u_o)\}$ or vice versa, NextOp adds (u, u') and (u', u_o) to E_P.

NextOp induces RM(E_P) and the picky domain adom(A, E_P) for involved attributes accordingly.

Generating AddE. GenRf adds two types of edges. (1) For each pair (u_o, u) in Q ($(u_o, u) \notin E_Q$), if there is a shortest path of length k between an RM node v and a match v' of a non-focus node u in Q (resp. if v and v' are not reachable), it adds AddE ((u, u'), $k - 1$) with $k - 1$ up to b (resp. AddE ((u, u'), 1)). (2) For each node u in Q, and any node v in G

within its b hop that can be reached by an RM node with a shortest path with length k (resp. not reachable), it adds AddE $((u,u'), k-1)$ (resp. AddE $((u,u'), k+1)$), where u' is a new pattern node, initialized with $L'_Q(u')$=$L(v)$).

Generating RfL. These rules refine existing (constant) literals in Q to reduce IM nodes. For each node u and match $v \in Q(u,G)$ reachable by some RM node, and for each literal l=$v.A$op$c$, it adds an RfL with value a only when at least a IM node fails the refined literal. (1) if c =⊥, add RfL(l, $u.A$ op a); (2) if op $\in \{<, \leq\}$ and $c \geq a$, add RfL(l, $u.A < a$); (3) if op $\in \{>, \geq\}$ and $c \leq a$, add RfL($l, u.A > a$); (4) if op is $'='$ and $a_i \neq c$, add RfL($l, u.A = a$). We remark that RfL is not generated for categorical-valued domains. Instead, the values are enumerated by an AddL.

Completeness of Operator Generation. Given the described procedure GenRf (resp. GenRx), it generates all the applicable refine (resp. relax) operators when there is a hope to reduce IM (resp. increase RM) by investigating the neighborhood of RM (resp. RC).

Proof of Lemma 5.2 To see Lemma 5.2(1), observe that $|\overline{\text{RC}}(o)| \geq |\text{RC}(o)|$. That is, for any relevant candidate $v \in$ RC(o) that eventually becomes matches by applying o, $v \in \overline{\text{RC}}(o)$, as the latter includes all the candidates (including v) induced by picky edges affected by o that can be possibly new matches. Thus $p(o) \geq \text{cl}(Q \oplus \{o\}, \mathcal{E}) - \text{cl}(Q, \mathcal{E})$.

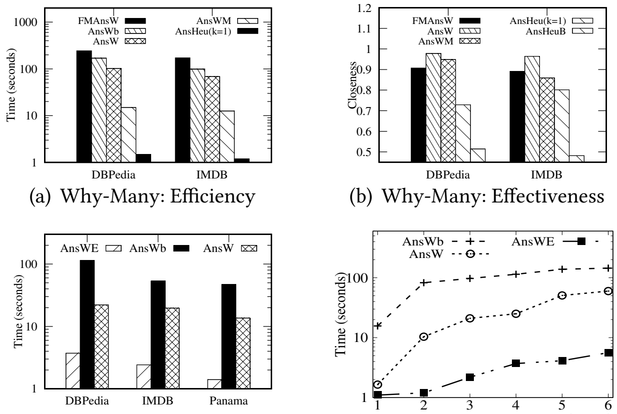
Proof of Lemma 5.5. Lemma 5.5 (1) can be shown by observing that when $\text{cl}^+(Q_2, \mathcal{E})$=cl*, for any Q_3=$Q_2 \oplus O$ with relaxation operators only, $\text{cl}^+(Q_3, \mathcal{E}) \leq \text{cl}^+(Q_2, \mathcal{E})$ and $Q_2(G) \setminus \text{rep}(\mathcal{E}, V) \subseteq Q_3(G) \setminus \text{rep}(\mathcal{E}, V)$; thus $\text{cl}(Q_3, \mathcal{E})$ is no larger than $\text{cl}(Q_2, G)$. This suggests an early switching from relaxation phase to refinement phase for any Q-Chase sequences. To see Lemma 5.5 (2), observe that Q_2 is refined from Q_1, thus $\text{cl}^+(Q_1, \mathcal{E}) \geq \text{cl}^+(Q_2, \mathcal{E}) \geq \text{cl}(Q_2, \mathcal{E})$. This pruning property helps to prune unpromising query rewrites at any refinement phase of Q-Chase sequences.

Queries with multiple focus nodes. Our algorithms can be readily extended to support answering Why-questions by exemplars that involve multiple focus nodes in Q. Indeed, for each focus node $u_i \in V_Q$ (1) $\mathcal{E}$ can be defined as the union of all $\mathcal{E}_i$ while rep($\mathcal{E}, V$) remains the same; (2) $Q(G)$ is extended to a set of sets $Q(u_i, G)$; and Q-Chase and all algorithms readily extended for this case.

APPENDIX C: EXTENSIONS AND ADDITIONAL EXPERIMENTS

Why-Many Questions. We provide the details of operator generation and analysis of answering Why-Many below.

Seed generation. Procedure SeedRf first "locally" refines operators in Q by choosing picky edges involving RM nodes, following its counterpart GenRf in algorithm AnsW. It then explores at most B hop neighbors of the matches $Q(u,G)$ that are connected to RM matches, for each node u in Q (denoted as $N_B(Q,G)$). Each time a node is visited, it enforces an AddE operator to O_s with a label matching the current distance, followed by generation of AddL and RfL, treating the newly inserted edge as a picky one. This process repeats until no new refinement operators can be generated.



(a) Why-Many: Efficiency (b) Why-Many: Effectiveness

(c) Why-Empty: Efficiency (d) Why-Empty: Varying $|E_Q|$

Figure 12: Why-Many and Why-Empty

Analysis of Why-Many. The fixed-parameter approximability can be verified by an approximation preserving reduction from answering Why-Many question to budgeted maximum weighted coverage [16], which is approximable within $\frac{1}{2} \cdot (1 - \frac{1}{e})$ by a greedy selection strategy. The above reduction holds when both k_1, including the query size and the number of attributes involved in Q and $\mathcal{E}$, and k_2, the maximum domain size of these attributes, are fixed. It takes $O(|N_B(Q,G)|k_1^2 \cdot k_2)$ time to identify O_s, and $O(|N_B(Q,G)|^{k_1})$ time to greedily select an operator. The total time is thus in $O(k_1^4 \cdot k_2^2 \cdot |N_B(Q,G)|^{k_1+2})$ time.

Exp-4: Answering Why-Empty and Why-Many. We evaluate the efficiency and effectiveness of answering Why-Many and Why-Empty questions in Fig. 12.

Performance for Why-Many. Fig. 12(a) and Fig. 12(b) reports the efficiency and effectiveness of answering Why-Many over DBpedia and IMDB. It tells us (1) ApxWhyM outperforms FMAnsW, AnsWb, and AnsW, by 15, 9.5, and 6 times. (2) On average, it takes 14 seconds for ApxWhyM to reduce the size of IM while it has approximate guarantees.

Performance for Why-Empty. We compare the PTIME algorithm AnsWE with the general algorithms AnsW and AnsWb/. Fig. 12(c) shows that AnsWE outperforms its counterparts AnsWb and AnsW by 7.91 and 3.79 times over all the datasets. Indeed, it only discovers atomic picky conditions and removes them within the cost bound.