# Privacy Preserving Subgraph Matching on Large Graphs in Cloud

Zhao Chang[*,#], Lei Zou[*], Feifei Li[#]
[*]Peking University, China; [#]University of Utah, USA;
{changzhao,zoulei}@pku.edu.cn; {zchang,lifeifei}@cs.utah.edu

## ABSTRACT

The wide presence of large graph data and the increasing popularity of storing data in the cloud drive the needs for graph query processing on a remote cloud. But a fundamental challenge is to process user queries without compromising sensitive information. This work focuses on privacy preserving subgraph matching in a cloud server. The goal is to minimize the overhead on both cloud and client sides for subgraph matching, without compromising users' sensitive information. To that end, we transform an original graph $G$ into a privacy preserving graph $G^k$, which meets the requirement of an existing privacy model known as *k-automorphism*. By making use of the symmetry in a $k$-automorphic graph, a subgraph matching query can be efficiently answered using a graph $G^o$, a small subset of $G^k$. This approach saves both space and query cost in the cloud server. We also anonymize the query graphs to protect their label information using label generalization technique. To reduce the search space for a subgraph matching query, we propose a cost model to select the more effective label combinations. The effectiveness and efficiency of our method are demonstrated through extensive experimental results on real datasets.

## 1. INTRODUCTION

Owing to the rich semantic and structure information represented by a graph, structured graph data is used in numerous applications, such as social networks, web graphs, biological networks, transportation networks, knowledge base, and RDF graphs. Many emerging applications rely on large graphs to satisfy their query needs, such as Google's knowledge graph and Facebook's graph search. Thus, graph data management has attracted significant attention, and efficient graph query processing on large graph is an important subject of study. In this paper, we focus on subgraph matching query [25, 18, 12], which is a key component of numerous applications. For example, answering SPARQL query $Q$ is equal to finding subgraph matches of $Q$ on an RDF graph $G$ [15]. Some graph databases also provide query language that is based on subgraph matching semantics, such as Cypher in Neo4j.

Meanwhile, increasingly, companies choose the "cloud" as their IT infrastructure platform. Using the cloud allows users to avoid expensive upfront infrastructure costs, and focus on projects that differentiate their businesses instead of on infrastructure [2]. Many public cloud services are available, such as Amazon cloud and Microsoft Azure. Some graph systems (such as GraphLab [14] and Neo4j [1]) offer SaaS (software-as-a-service)-style cloud services. In other words, they allow users to upload their graph data to their cloud platforms and provide cloud-based computing services over the outsourced graph data.

However, while utilizing cloud services for building graph applications is a cost-effective solution, the potential risk of compromising sensitive information is a serious problem.

One serious privacy leakage is the "identity disclosure" problem [20, 13]. Assume that an adversary can locate the target entity $t$ as a vertex $v$ of a social network graph $G$ with a high probability. We say that the identity of $t$ is disclosed. A naïve anonymization solution is to remove all identifiable personal information before publishing the network, such as names and social security numbers. However, even when a network is published without any identity information, it is still possible to locate the target with a high probability based on some structural information around the target [10].

For example, if an adversary knows the local graph structure (such as degree, 1-neighbor graph) around the target $t$, he/she may issue a subgraph query representing the local graph structure to find the matching position. If there are only few matches in graph $G$, the target will be identified with a high probability. Once it is determined that a vertex $v$ corresponds to the target $t$, all sensitive attributes associated with $v$ will be compromised. These are called *structural attacks*; see details in [13, 24, 10].

To address these threats, many privacy preserving graph data publishing techniques have been proposed [26, 6, 22]. A typical solution to protect graph privacy is based on the symmetry of the released graph data [26, 6, 22]. Specifically, given a graph $G$, we transform $G$ into a $k$-automorphic graph $G^k$ by introducing some noise edges, where each vertex has at least $(k-1)$ other symmetric vertices. It means there are no structural differences between $v$ and each of its $(k-1)$ symmetric vertices. Thus it is impossible to distinguish $v$ from the other $(k-1)$ symmetric vertices. In other words, the $k$-automorphism strategy [26] can defend against any structure-based attack.

These privacy preserving graph data publishing techniques seem like perfect solutions to our problem, but they *compromise the utility of query results*. In particular, they may return false positives regarding subgraph matching queries, due to the noise edges and vertices that were added. However, in a cloud setting, it is possible to offload most query processing costs to the cloud server and ask the clients to execute a simple and efficient filtering step to remove false positives and find the exact answers.

EXAMPLE 1. *Consider a professional social network in Figure 1 that is modeled as a graph G. Each vertex in G represents an*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA*
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882956

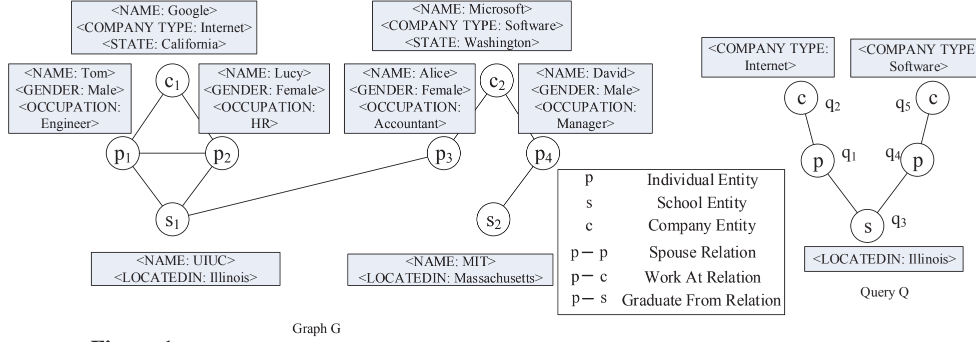



**Figure 1: An original data graph and a query graph on a professional social network graph.**

*entity, such as an individual ($p_i$), a company ($c_j$), or a school ($s_\ell$). Each edge in G represents a relation between two entities, such as "spouse" relation, "work at" relation and "graduate from" relation. Each entity has some attributes. For example, the attributes associated with individuals are "gender", "occupation" and so on.*

*A user wants to find "two individuals satisfying (1) both of them graduated from the same university located in Illinois, and (2) one of them is working at a software company and the other one is working at an Internet company". The user issues a subgraph pattern matching query Q over G, as shown in Figure 1.*

*Sending the original graph G to the cloud will leak user privacy in G. Thus, we resort to the privacy preserving graph publishing techniques. For example, we can use the k-automorphism model [26]; the k-automorphic graph $G^k$ for G and $k = 2$ is shown in Figure 2, where noise edges are shown by red dashed lines. Clearly, $G^k$ protects the structure privacy of the graph G. To protect label privacy, we also use label anonymization where each vertex label in $G^k$ is replaced by a label group (i.e., a generalized label). We use Label Correspondence Table (LCT) (in Figure 2) to represent the mapping between label groups and vertex labels. For example, vertex $p_1$ in $G^k$ has label groups C and E in Figure 2.*

*A straightforward method is to upload $G^k$ to the cloud and perform graph queries over $G^k$. Although the approach does not leak any sensitive information, the answers to query Q over graph $G^k$ ($R(Q, G^k)$) are different from those over G ($R(Q, G)$). For example, there are only two subgraph matches of Q over the original graph G. However, due to the noise edges introduced into the k-automorphic graph $G^k$ and the label anonymization, there are eight matches over $G^k$. Nevertheless, when the cloud returns $R(Q, G^k)$ with eight matches, the client can efficiently filter out false positives based on G to derive $R(Q, G)$.*

Note that the filtering step at the client side is *much cheaper than asking the client to run the subgraph matching query Q over G* (which is known to be expensive especially over large graphs). In other words, the cloud server did most of the work in this process, making the cloud service worthwhile and attractive for the client. We do assume that the client is the data owner who has access to $G$ for the filtering step. For the general case, a query client (who is trusted and authenticated by the data owner) can ask for the data owner to execute the (very lightweight) filtering step.

However, the example above shows some major limitations of directly applying the existing graph data publishing techniques [26, 6, 22]. Firstly, most of these techniques that focus on structural attacks do not protect label privacy at the same time. But more importantly, to achieve higher privacy, they need to add a lot of noise edges and/or vertices (e.g., a large $k$ value in the case of the $k$-automorphism model) to the original graph $G$. This results in a much larger graph (than the original graph) on the cloud side, which leads to much more expensive storage cost, much larger communication overhead, and much higher query costs for both the cloud server and the query client.

This work focuses on *reducing these overheads without compromising either data and query graphs' privacy or the correctness of the final query results at the client side*.

**Solution overview.** In order to achieve that, we first propose a basic solution. We transform the data graph $G$ into an outsourced graph $G^k$ using existing privacy-preserving graph publication techniques, such as the $k$-automorphism model [26]; and then upload $G^k$ to the cloud. At query time, we propose a *two-phase query evaluation strategy*. First, given a query graph $Q$, we transform $Q$ into an outsourced query graph $Q^o$. In the cloud, we evaluate subgraph query $Q^o$ over $G^k$ to obtain query results $R(Q^o, G^k)$. Then, in the client side, we filter out false positives in $R(Q^o, G^k)$ based on $G$ to find the final, correct results $R(Q, G)$, i.e., subgraph matches of $Q$ over the original graph $G$. To improve the query performance, we propose a solution that only uploads a small part of $G^k$ to the cloud by leveraging the symmetry of the $k$-automorphic graph $G^k$. This method saves both space cost and query processing time in the cloud significantly. Although only a (small) part of $G^k$ is uploaded to the cloud, our method still guarantees the correctness of the query results. Furthermore, we also propose a cost model based label combination/generalization strategy to reduce the search space for subgraph matching queries in the cloud.

**Contributions.** To the best of our knowledge, this is the first work that supports privacy preserving subgraph matching queries over a large graph in the cloud while protecting privacy without undermining query results. In this paper, we consider the cloud server "honest-but-curious", which is consistent with most related works in the literature. In other words, the cloud server always offers correct computations without cheating. However, the cloud server is "curious" to learn the graph data, its index structure, and user queries so as to gain sensitive information if he can. Our main contributions are summarized as follows.

- We propose an effective strategy to provide exact subgraph matching query services in public cloud while preserving private information in the data graph and query graphs.
- To save both space cost and query processing cost, we only upload a small subset of the anonymized data graph to the cloud. We answer subgraph matching queries efficiently in the cloud by making use of the symmetry of the $k$-automorphic graph. These techniques contribute to saving cost while minimizing the overhead in the client side.
- In order to reduce the search space in answering subgraph matching queries, we design a novel cost model to select effective vertex label combinations for anonymizing labels in the data graph and query graphs.
- We study the effectiveness and efficiency of our method through extensive experiments over several large real graphs.

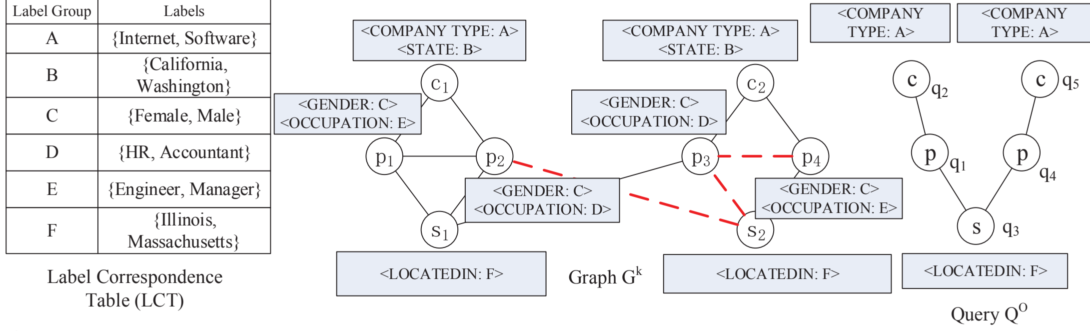



**Figure 2: A $k$-automorphic graph and an anonymized query graph on the same professional social network graph for $k = 2$.**

**Table 1: Notations**

| | |
|---|---|
| $G$ | The original data graph |
| $G^k$ | The data graph released by $k$-automorphism algorithm |
| $G^o$ | The outsourced data graph |
| $Q$ | The original query graph |
| $Q^o$ | The outsourced query graph |
| $S_i$ | The $i$-th star of $Q^o$ generated by query decomposition |
| $R_{in}$ | The set of candidate matching results of $Q^o$ generated in cloud |
| $R(Q,G)$ | The set of subgraph matches of $Q$ on $G$ |

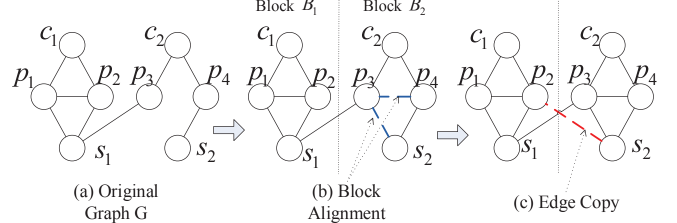



**Figure 3: An example of the $k$-automorphism algorithm [26].**

# 2. BACKGROUND

## 2.1 Preliminary

Traditional subgraph matching methods always adopt the vertex labeled graph model [23], where each vertex has a single label. But this model is not desirable to model complex graph data, such as social networks and RDF graphs. In this work, we adopt the *attributed graph model*, where each vertex has a rich data structure including vertex type, vertex attributes and vertex labels (i.e., attribute values). For simplicity, we only consider rich data structures on vertices and ignore those on edges, although handling the more general case is not more complicated. For example, we can introduce an imaginary vertex to represent an edge of interest and assign the rich data structure on the edge to the new vertex. Formally, our graph model is defined as follows. Table 1 lists some frequently-used notations in this paper.

**Definition** 1. ***Attributed Graph Model.*** *An attributed graph is defined as $G = \{V(G), E(G), T, \Gamma, L\}$, where (1) $V(G)$ is a set of vertices; (2) $E(G) \subseteq V(G) \times V(G)$ is a set of undirected edges; (3) $T$ is a set of vertex types, where each vertex has and only has one vertex type; (4) $\Gamma$ is a set of vertex attributes, where each vertex type has one or more vertex attributes and different vertex types have different vertex attributes; and (5) $L$ is a set of vertex labels, where each vertex attribute has one or more vertex labels.*

The vertex type, vertex attributes and vertex labels of vertex $v$ are denoted as $T(v)$, $\Gamma(v)$, $L(v)$, respectively. For any two different vertices $v_1$ and $v_2$, if and only if $T(v_1) = T(v_2)$, then $\Gamma(v_1) = \Gamma(v_2)$. For one vertex attribute $A \in \Gamma(v)$, $\langle A, \{a_1, \cdots, a_n\}\rangle$ represents the vertex attribute values on attribute $A$ which are the vertex labels of $A$. Note that one vertex attribute may have multiple vertex labels (i.e., attribute values). For example, the "Company Type" of one company can be "Internet", "Software", and so on.

The data graph $G$ and the query graphs $Q$ in the running example follow the attributed graph model.

**Definition** 2. ***Subgraph Match.*** *Given a data graph $G = \{V(G), E(G), T, \Gamma, L\}$ and a query graph $Q = \{V(Q), E(Q), T, \Gamma, L\}$, $Q$ is* subgraph isomorphic *to $G$, if and only if there exists at least one injective function $g : V(Q) \rightarrow V(G)$ such that*
*1) $\forall q_i \in V(Q)$, $g(q_i) \in V(G) \Rightarrow L(q_i) \subseteq L(g(q_i))$; and*
*2) $\forall q_i, q_j \in V(Q)$, edge $\overline{q_i q_j} \in E(Q) \Rightarrow$ edge $\overline{g(q_i)g(q_j)} \in E(G)$.*

*If $Q$ is* subgraph isomorphic *to $G$, a* subgraph match *$M$ of $Q$ is a subgraph of $G$, represented as $\langle g(q_1), \cdots, g(q_n)\rangle$.*

*The set of* subgraph matches *of $Q$ on $G$ is denoted as $R(Q,G)$.*

Our problem can now be formalized as follows:

**[Problem Definition]** *Given a data graph $G$ and a query graph $Q$, our problem is to find all subgraph matches of $Q$ over $G$, denoted as $R(Q,G)$, through a cloud server, while preserving private information in data graph $G$ and query graph $Q$.*

Note that the goal of our problem is to protect the privacy of data graph and query graph against the cloud, where clients who ask a subgraph query have the ability to de-anonymize, filter and verify the results returned from the cloud based on the original graph.

## 2.2 K-Automorphism

As explained in Section 1 (Example 1 and its discussion), it is possible to apply existing privacy preserving graph publication techniques in constructing a baseline solution. However, they suffer from major efficiency limitations. Furthermore, most of these techniques that focus on structural attacks do not protect label privacy at the same time. Nevertheless, we can use these techniques as a preprocessing step (e.g., $k$-automorphism [26]) towards constructing more effective and efficient solutions. That said, we will briefly review $k$-automorphism [26] next.

The basic idea of $k$-automorphism is as follows. Given a graph $G$ to be published, we transform it into $G^k$ by introducing more vertices and edges, where $G^k$ satisfies the $k$-automorphic graph model (Definition 3). It means that any vertex $v$ in graph $G^k$ cannot distinguish itself from each of its symmetric $(k-1)$ vertices in $G^k$. Therefore, $G^k$ is safe to publish. Here, it is assumed that the graph is unlabeled. Figure 3 shows an example of the algorithm in [26].

**Definition** 3. ***K-Automorphic Graph [26].*** *A $k$-automorphic graph $G^k$ is defined as $G^k = \{V(G^k), E(G^k)\}$, where $V(G^k)$ can be divided into $k$ blocks and each block has $\left\lceil \frac{|V(G^k)|}{k} \right\rceil$ vertices. Any vertex $v$ has $k-1$* symmetric vertices *$v'$* [1] *in the other $k-1$ blocks.*

The $k$-automorphic graph is generated with the help of the $k$-automorphic function defined as follows.

**Definition** 4. ***K-Automorphic Function [26].*** *Given a vertex $v$ in a $k$-automorphic graph $G^k$, $v$ and its corresponding $k-1$* symmetric vertices *form an* alignment vertex instance *(AVI).*

[1] "symmetric" means swapping $v$ and $v'$ gives an isomorphic graph.

| $p_1$ | $p_4$ |
|---|---|
| $p_2$ | $p_3$ |
| $s_1$ | $s_2$ |
| $c_1$ | $c_2$ |

(a) Alignment Vertex Table

$F_1(p_1) = p_4; F_1(p_4) = p_1$
$F_1(p_2) = p_3; F_1(p_3) = p_2$
$F_1(s_1) = s_2; F_1(s_2) = s_1$
$F_1(c_1) = c_2; F_1(c_2) = p_1$

(b) Automorphic Function $F_1$

**Figure 4: AVT and automorphic functions.**

*All AVIs are stored in an Alignment Vertex Table (AVT). Each AVI I in AVT is represented as a circularly-linked list. Specifically, I.at(i) ($i = 0, \cdots, k-1$) denotes the i-th vertex in instance I. Suppose $v = I.at(i)$. If $i \le k-2$, $v.next = I.at(i+1)$; else $v.next = I.at(0)$. For each vertex v in an AVI, we can define k automorphic functions $F_i$ ($i = 0, \cdots, k-1$) in $G^k$ based on the AVI, where*

- $F_0(v) = v$;
- $F_i(v) = F_{i-1}(v).next$, *for* $1 \le i \le k-1$.

*If M is a subgraph of $G^k$, each $F_i(M)$ ($i = 0, \cdots, k-1$) is defined as a mapping graph under function $F_i$, where:*

- $V(F_i(M)) = \{u \mid \exists v \in V(M), u = F_i(v)\}$;
- $E(F_i(M)) = \{\overline{u_1u_2} \mid \exists\ \overline{v_1v_2} \in E(M), u_1 = F_i(v_1) \wedge u_2 = F_i(v_2)\}$.

A $k$-automorphic graph $G^k$ (for $k = 2$) generated from $G$ (in Figure 1) is given in Figure 2. The corresponding Alignment Vertex Table (AVT) is given in Figure 4(a). Each row of AVT shows $k$ symmetric vertices. For example, the first row of AVT contains $p_1$ and $p_4$, meaning $p_1$ and $p_4$ are symmetric vertices. In other words, swapping vertices $p_1$ and $p_4$ will generate another isomorphic graph. Each column of AVT shows all vertices in one block of the $k$-automorphic graph. For example, there are two blocks $\{c_1, p_1, p_2, s_1\}$ and $\{c_2, p_4, p_3, s_2\}$ in the 2-automorphic graph in Figure 3. Based on AVT, we also show the automorphic function $F_1$ of the running example in Figure 4(b).

Given a graph $G$, we generate a $k$-automorphic graph $G^k$ from $G$ as follows. First, we adopt the METIS algorithm [11] to partition $G$ into $k$ blocks. For example, to generate the 2-automorphic graph for the graph $G$ in Figure 1, we obtain two blocks $B_1\{c_1, p_1, p_2, s_1\}$ and $B_2\{c_2, p_4, p_3, s_2\}$ by partitioning $G$. If $|V(G)|$ cannot be divided by $k$, we will introduce some noise vertices to guarantee that every block has exactly $\left\lceil \frac{|V(G^k)|}{k} \right\rceil$ vertices. There is an efficient method to build the Alignment Vertex Table (AVT) (e.g., Figure 4) [26]. Based on the AVT, it is easy to define the $k$-automorphic functions. Finally, based on the $k$-automorphic functions, we build $G^k$ by performing *graph alignment* and *edge copy*. For example, we perform graph alignment on $B_1$ and $B_2$ to obtain two alignment blocks $B_1'$ and $B_2'$ (adding edge $(p_3, p_4)$ and edge $(p_3, s_2)$ in Figure 3(b)), where $B_1'$ and $B_2'$ are isomorphic to each other. Lastly, the "edge copy" technique is used to deal with the crossing edges between different blocks (adding edge $(p_2, s_2)$ that corresponds to edge $(p_3, s_1)$ in Figure 3(c)). Readers can refer to [26] for further details. Furthermore, it is easy to show that any vertex in the $k$-automorphic graph $G^k$ cannot distinguish itself from each of its $(k-1)$ symmetric vertices. It means that any adversary cannot identify any target vertex with a probability higher than $\frac{1}{k}$. [2]

## 2.3 Our Framework and Analysis Overview

In order to provide the correct cloud-based subgraph query services while preserving privacy in data graph and query graphs, our solution works as follows. We transform $G$ into an outsourced graph $G^o$ and upload $G^o$ to the cloud. We guarantee that $G^o$ does not leak any private information in the original graph $G$. Specifically, we require that any adversary, who sees $G^o$, cannot identify any target vertex in graph $G$ with a probability higher than $\frac{1}{k}$. We adopt the *k-automorphism* model [26] to achieve this objective, by generating $G^o$ based on $G^k$. At query time, given a query graph $Q$, we transform $Q$ into an outsourced query graph $Q^o$. Leveraging the symmetry of the $k$-automorphic graph, the cloud server answers subgraph matching query $Q^o$ based on $G^o$ and returns $R_{in}$ (a small subset of $R(Q^o, G^k)$, see Section 4.2.1) to the client. Based on $R_{in}$, the client can efficiently recover $R(Q^o, G^k)$ and finally find the final results $R(Q, G)$. We highlight several key requirements.

Firstly, we need to define what sensitive information is in data graph and query graphs. We do not consider the vertex types $T$ and vertex attributes $\Gamma$ as sensitive information, since they do not compromise users' privacy. For any vertex $v$ in $G$ or $Q$, we consider vertex labels $L(v)$ (i.e., attribute values) as private information. Thus, the "*label privacy*" considered in this paper refers to the privacy of the attribute values of each vertex in the graph. For example, given an "individual" vertex in graph $G$ in Figure 1, the values of gender and occupation are users' privacy. Subgraph matching queries will reveal vertex labels in original data graph $G$, if we do not anonymize the vertex labels (i.e., attribute values) in query graphs. Thus, the vertex labels in query graph $Q$ are also considered as sensitive information.

Secondly, we must reduce expenses for the cloud server; that is to say, we need to save query time and storage space in the cloud and reduce communication overhead between the client and the cloud. To protect data privacy, we have to introduce noise edges into the data graph and anonymize attribute values of vertices. Doing so will lead to larger search space in subgraph matching. Therefore, in Section 5, we propose a cost model to guide how to select a good graph anonymization strategy. We also study how to reduce the storage space and communication overhead by leveraging the symmetry of the anonymized graph in Section 4, which also helps reduce the cloud's query processing cost.

Thirdly, we should minimize the overhead in the client side. Subgraph matching is an expensive task in terms of both time and space complexity. In our framework, the client only needs to filter out false positives using a hash index in linear time complexity (in terms of the number of candidate results generated by the cloud).

# 3. A BASELINE SOLUTION

We first describe a simple scheme to help illustrate the basic principle of our approach. For a graph $G$, we first generate the $k$-automorphic graph $G^k$. A baseline solution is to upload $G^k$ to the cloud directly. Note that $k$-automorphism [26] assumes that the graph is unlabeled. To protect the vertex labels in data graph and query graphs, we resort to the generalization technique, which is also used in $k$-anonymity [17, 19]. Specifically, each vertex label in data graph and query graphs is represented by a generalized label. We refer to a generalized vertex label as a "label group" in the following discussion. Here, we assume that each label group contains not less than $\theta$ distinct labels, where $\theta$ is a user-specified parameter. For example, a label group "$A$" in Figure 2 includes $\theta = 2$ labels {Internet, Software}.

The *Label Correspondence Table* (LCT) (in Figure 2) shows all label groups in the running example. Obviously, the label generalization strategy will affect the query performance. We assume that the label groups are given until Section 5, where we will propose a cost model to find a good label generalization strategy.

Using label generalization on a data graph $G$, we obtain a graph $G'$ whose vertices hide vertex labels by label groups. Next, using the $k$-automorphism algorithm in [26], we obtain a $k$-automorphic data graph $G^k$ and the corresponding AVT. Each vertex $v$ in $G^k$ has

[2] The proof was given in Theorem 4.4 of [26].

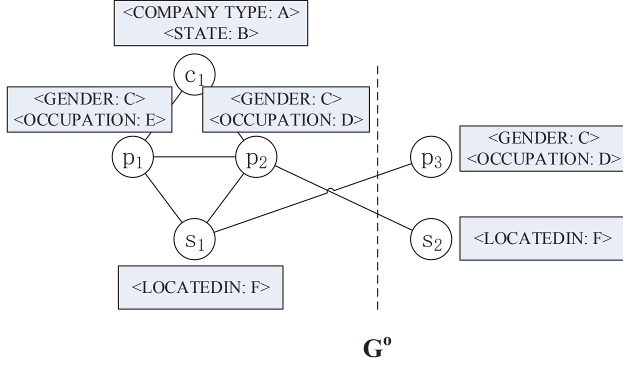



**Figure 5: The outsourced data graph $G^o$ for $G$ in Example 1.**

$(k-1)$ symmetric vertices $F_i(v)$ under automorphic functions $F_i$ for $i = 1, \cdots, k-1$. The $k$ vertices, i.e., $v$ together with $F_i(v)$ for $i = 1, \cdots, k-1$, form a symmetric vertex group, which corresponds to a row in AVT (an AVI). For each symmetric vertex group, we ensure that all vertices in it have the same label groups, i.e., a vertex $v$'s label gropus are $L(v) \cup L(F_1(v)) \cup \cdots \cup L(F_{k-1}(v))$.

EXAMPLE 2. *Given the original data graph G in Figure 1, the k-automorphic graph $G^k$ together with its label correspondence table (LCT) is given in Figure 2. A naïve solution is to upload $G^k$ to the cloud directly.*

Given a query graph $Q$, we anonymize $Q$ by representing each vertex label using its corresponding label group. The anonymized query graph is denoted as $Q^o$, which is submitted to the cloud. The cloud server answers subgraph query $Q^o$ over the $k$-automorphic graph $G^k$. We know that $R(Q,G) \subseteq R(Q^o, G^k)$. Intuitively, we introduce more edges and vertices into $G$ to form $G^k$, and $Q^o$ and $G^k$'s vertex labels are anonymized using the same LCT. Formally,

**Theorem** 1. *For graph G and query graph Q, $R(Q,G) \subseteq R(Q^o, G^k)$.*

Finally, $R(Q^o, G^k)$ are sent to the client and the client filters out false positives in $R(Q^o, G^k)$ based on $G$ to obtain $R(Q,G)$.

The baseline solution suffers from the following limitations. First, we need to upload the $k$-automorphic graph $G^k$ to the cloud. Since the size of $G^k$ can be significantly larger than $|G|$, this introduces communication overhead and higher storage cost. Secondly, a larger graph $G^k$ naturally leads to a larger search space for subgraph matching, which leads to higher query cost, especially as $k$ increases.

Nevertheless, the baseline solution already saves the client from executing the very expensive subgraph matching query herself.

# 4. THE OPTIMIZED METHOD

## 4.1 Outsourced Graph

According to Definition 3, $G^k$ is a $k$-automorphic graph consisting of $k$ blocks. Each vertex $v$ in $G^k$ has $(k-1)$ symmetric vertices in the other $(k-1)$ blocks. Intuitively, we only need to upload a block of $G^k$ together with $k$ automorphic functions $F_i$ $(i = 0, \cdots, k-1)$ to the cloud, since the cloud can recover the whole $G^k$ based on $G^o$ and the functions $F_i$. This is the motivation of defining the outsourced graph $G^o$ (Definition 5), which is exactly the first block of $G^k$ together with the 1-hop neighbors in $G^k$.

**Definition** 5. ***Outsourced Graph.*** *For a graph $G(V(G), E(G), T, \Gamma, L)$ and its k-automorphic graph $G^k(V(G^k), E(G^k), T, \Gamma, L)$, an outsourced graph of G is defined as $G^o = \{V(G^o), E(G^o), T, \Gamma, L\}$, where (1) $G^o$ follows the* attributed graph model*; (2) $V(G^o)$ is the union of vertices in the first block of $G^k$, denoted as $V(B_1)$, together with their one-hop neighbors in $G^k$ denoted as $V(N_1)$; and (3) $E(G^o)$ is the subset of undirected edges from $E(G^k)$ that connect vertices within $V(B_1)$ and vertics between $V(B_1)$ and $V(N_1)$.*

Given a $k$-automorphic graph $G^k$, according to Definition 5, we generate an outsourced graph $G^o$ and upload it to the cloud. For example, given $G$ and $G^k$ in Figure 1 and Figure 2 from Example 1 respectively, an outsourced graph $G^o$ is represented in Figure 5. $G^o$ contains the vertices in the first block of $G^k$ (i.e., the vertices in the first column of AVT) together with their 1-hop neighbors in $G^k$, and their corresponding edges in $G^k$. Note that the size of $G^o$ is much smaller than the size of $G^k$ (roughly an $1/k$ fraction of it).

Although $G^o$ is a part of $G^k$, according to the $k$ automorphic functions $F_i$ (see Definition 4), each vertex/edge in $G^k$ must have a counterpart in $G^o$. It means that we can easily recover $G^k$ based on $G^o$ and the $k$ automorphic functions $F_i$ $(i = 0, \cdots, k-1)$. This is the intuition that we only upload $G^o$ to the cloud without compromising the accuracy of query results. More sophisticated technical details are discussed in the following subsection.

## 4.2 Privacy Preserving Subgraph Query

Given a query $Q$ at the client side, we generalize its vertex labels to form $Q^o$. Specifically, for each vertex label in $Q$, we replace it with the corresponding label group according to the LCT (an example is shown in Figure 2). $Q^o$ has the same size as $Q$, although each vertex of $Q^o$ has a generalized label group. $Q^o$ ensures the privacy of the vertex label information in $Q$. It is a trade-off between the query privacy and the query performance. Thus, label generalization is an interesting challenge in its own and we assume for now that this is done and present the details in Section 5.

The client then sends $Q^o$ to the cloud. The cloud first finds subgraph matches of all *basic units* of $Q^o$ over $G^o$, which will be defined shortly. Using the symmetry of the $k$-automorphic graph, the cloud can then obtain $R(Q^o, G^k)$, *but without $G^k$ since it only has* $G^o$, by joining these intermediate results. Lastly, $R(Q^o, G^k)$ is sent back to the client, who can obtain $R(Q,G)$ by pruning the false positives in $R(Q^o, G^k)$. Theorem 1 guarantees the correctness of this framework.

### *4.2.1 Processing in the Cloud*

A $k$-automorphic graph $G^k$ consists of $k$ symmetric blocks ($B_1$, $\cdots$, $B_k$), while the outsourced graph $G^o$ contains only one block together with the first-hop neighbors of its boundary vertices. *The challenge* is how to find subgraph matches *crossing* multiple blocks of $G^k$ without accessing $G^k$, since only $G^o$ resides in the cloud.

We adopt the query decomposition method. Given a query $Q^o$, the cloud server decompose $Q^o$ into a set of stars $\{S_i\}$, $i = 1, ..., n$, where a star is a root vertex together with its adjacent edges and neighbors in $Q^o$. Then, it finds $R(S_i, G^o)$ for each star graph $S_i$ for $i = 1, \cdots, n$. Leveraging the symmetry of $G^k$, we can then obtain $R(S_i, G^k)$ based on $R(S_i, G^o)$. Finally, we obtain $R(Q^o, G^k)$ by joining the matching results for these stars.

**Query Decomposition.** We first discuss how to decompose $Q^o$ into a set of stars and how to find the optimal query decomposition.

Consider a query $Q^o$ in Figure 2. The stars rooted at vertices with Type $P$ are shown in Figure 6, where a shaded circle denotes the *center (aka root)* of a star.

Intuitively, a query decomposition is a set of stars that collectively cover the outsourced query graph $Q^o$. Figure 6 shows the query decomposition $\{S_1, S_2\}$ over query $Q^o$. The set of subgraph matches of a star graph $S_i$ with respect to the outsourced graph $G^o$ is $R(S_i, G^o)$, or simply $R(S_i)$ when the context is clear. To reduce the number of intermediate results, we design a *cost model* to estimate the number of matches, $|R(S_i)|$, for each star $S_i$. We will discuss the technical details of the cost model in Section 5. Here, we assume

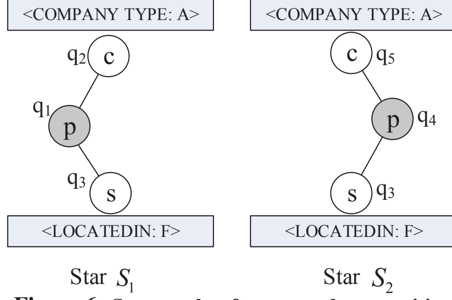



**Figure 6: Star graphs after query decomposition.**

| Label Group | Vertex Type | | |
|---|---|---|---|
| | C <$c_1$> | P <$p_1$, $p_2$> | S <$s_1$> |
| A | <1> | | |
| B | <1> | | |
| C | | <1, 1> | |
| D | | <0, 1> | |
| E | | <1, 0> | |
| F | | | <1> |

Vertex Bit Vector Table (VBV)

| Vertex | Label Groups of the Corresponding Neighbor Vertices <A, B, C, D, E, F> |
|---|---|
| $c_1$ | <0, 0, 1, 1, 1, 0> |
| $p_1$ | <1, 1, 1, 1, 0, 1> |
| $p_2$ | <1, 1, 1, 0, 1, 1> |
| $s_1$ | <0, 0, 1, 1, 1, 0> |

Neighbor Label Bit Vector Table (LBV)

**Figure 7: Index structure.**

that each $|R(S_i)|$ $(i = 1, \cdots, n)$ is given. We define the cost of the query decomposition as follows.

**Definition** 6. ***The Cost of Query Decomposition****. Given a set of stars $\{S_1, \ldots, S_n\}$ that is a decomposition of an outsourced query graph $Q^o$, the cost of the query decomposition is:*

$$cost(Q^o) = \sum_{i=1}^{n} |R(S_i)|.$$

It is straightforward to reduce a minimum weighted vertex cover (a classical NP-hard problem) to the problem of finding the query decomposition with the minimum cost. Formally,

**Theorem** 2. *Given an outsourced query graph $Q^o$, finding the query decomposition with the minimum cost with respect to Definition 6 is an NP-hard problem.*

That said, we formulate finding the optimal query decomposition as the following ILP (integer linear programming) problem and use an available ILP tool, e.g, the Gurobi ILP solver, to solve this problem. Although ILP is still an NP-hard problem, $Q^o$ is always a small-size graph. Thus in practice, it is actually very efficient to find the optimal query decomposition through this formulation. Solving this ILP also gives us the value $n$, the number of stars, in the query decomposition.

$$minimize \sum_{\text{all } v_i \in V(Q^o)} |R(S(v_i))| x_{v_i}$$
$$subject\ to \quad x_{v_i} + x_{v_j} \geq 1 \quad \text{for all } \overline{v_i v_j} \in E(Q^o)$$

*/*Each edge is contained in at least one star.*/*

$$x_{v_i} \in \{0, 1\} \text{ for all } v_i \in V(Q^o)$$

*/*A star rooted at each vertex $v_i$ is either selected ($x_{v_i} = 1$) into the query decomposition or not ($x_{v_i} = 0$).*/*

**Star Matching.** Next, we present our star matching algorithm. The algorithm is designed to find the subgraph matches for each star $S_i$, in the query decomposition of $Q^o$, over $G^o$, i.e., $R(S_i, G^o)$. The cloud server builds a query index structure offline to improve the query efficiency.

The index can be considered as two parts: one is regarding the vertex label, and the other one is regarding the neighbourhood structure. Thus, there are two components in the index structure, as shown in Figure 7 which are constructed based on LCT (label correspondence table) and $G^o$ (see Figure 2 and Figure 5 respectively). They are the *Vertex Bit Vector (VBV) Table* and the *Neighbor Label Bit Vector (LBV) Table*, respectively. For convenience, the first block of $G^k$ is denoted as $B_1$. Obviously, $B_1$ is a subgraph of $G^o$.

Each VBV corresponds to a label group, where the corresponding bit in the VBV for a vertex $v \in B_1$ is set to 1 iff $v$ contains that label group. For example, $p_2$ contains $D$ but not $E$ in $B_1$. Thus the two corresponding bits in VBV are 1 and 0, respectively.

In LBV, for each vertex $v$ in $B_1$, the corresponding bit for a label group $L$ is set to 1 if and only if $L$ is contained in at least one label set for $v$'s neighbor vertices. For example, $D$ is contained in the label set of $p_2$ that is one of $p_1$'s neighbors. However, $E$ is not contained in the label set of any of $p_1$'s neighbors. Thus the two corresponding bits for $p_1$ in LBV are 1 and 0, respectively.

---

**Algorithm 1** Star Matching Algorithm

**Require:** Input: $G^o$ (the outsourced data graph) and $S^*$ (the set of stars that contains star $S_i$ with center $v_i$, for $1 \leq i \leq n$).
Output: $RS$ ($R(S_i, G^o)$ for $1 \leq i \leq n$).
1: Initialize $RS := \phi$.
2: **for** $i := 1$ to $n$ **do**
3: Initialize $RS_i := \phi$.
4: Set $\alpha := VBV(L(v_i, 1)) \wedge VBV(L(v_i, 2)) \wedge \cdots \wedge VBV(L(v_i, |L(v_i)|))$.
5: **for** each vertex $v_a$ that corresponds to a non-zero bit in $\alpha$ **do**
6: **if** $LBV(v_a) \wedge LBV(v_i) = LBV(v_i)$ **then**
7: Generate the set of matches of $S_i$ with center $v_a$, denoted as $RS_{temp}$.
8: $RS_i := RS_i \cup RS_{temp}$.
9: $RS := RS \cup RS_i$.
10: Return $RS$.

---

The *Star Matching Algorithm* is presented in Algorithm 1. Without loss of generality, assume that the center of a star $S_i$ is vertex $v_i$. For each star $S_i$ for $1 \leq i \leq n$, we first find each *Vertex Bit Vector* that corresponds to every label group of $v_i$; e.g. VBV($C$) = (1, 1) in our example. Recall $L(v)$ is the set of label groups for vertex $v$. We use $L(v, j)$ to denote the $j$-th label group of $v$, for $1 \leq j \leq |L(v)|$.

Hence, the first step is to find $VBV(L(v_i, j))$ for $1 \leq j \leq |L(v_i)|$. We perform a *bitwise AND* operation on these $|L(v_i)|$ bit vectors to obtain the result vector $\alpha$ (Line 4). Each vertex $v_a$ that corresponds to a non-zero bit in $\alpha$ is a candidate match of $v_i$. If each label group of $v_i$'s neighbors can be found in the label sets of $v_a$'s neighbors (Line 6), some of $v_a$'s neighbors can be candidate matches of $v_i$'s neighbors. By enumerating candidate vertex combinations of $v_a$ and its neighbor vertices, we generate matches of $S_i$ and add them to the result set (Line 7-Line 8).

**Result Join.** The next challenge is how to compute $R(Q^o, G^k)$ based on the star matching results $R(S_i, G^o)$, for $i = 1, ..., n$. A straightforward solution works as follows. First, using the $k$-automorphic function $F_j$ $(j = 0, \cdots, k-1)$, we compute $R(S_i, G^k)$ based on $R(S_i, G^o)$. Then we compute $R(Q^o, G^k)$ by joining $R(S_i, G^k)$'s, i.e., $R(Q^o, G^k) = R(S_1, G^k) \bowtie R(S_2, G^k) \bowtie \cdots \bowtie R(S_n, G^k)$. Obviously, the cost of the join is $\prod_{i=1}^{n} |R(S_i, G^k)|$.

Figure 8 demonstrates the above process using the running example. In our example, $Q^o$ is decomposed into two stars $S_1$ and $S_2$ (in Figure 6). Given the graph $G^o$ (in Figure 5) on the cloud, we perform subgraph matching to obtain $R(S_1, G^o)$ and $R(S_2, G^o)$ (as shown in Figure 8), i.e., the subgraph matches of $S_1$ and $S_2$ over graph $G^o$, respectively. According to the automorphic function $F_1$ (defined in Figure 4), we can derive $R(S_1, G^2)$ and $R(S_2, G^2)$. Specifically, for each match $M$ in $R(S_1, G^o)$, using the automorphic function $F_1$, $F_1(M)$ derives another match $M'$ in $R(S_1, G^2)$. For example, $(p_1, c_1, s_1)$ is a match of $S_1$ over $G^o$; thus, $(F_1(p_1), F_1(c_1), F_1(s_1))$ gives another match $(p_4, c_2, s_2)$. Thus, we get $R(S_1, G^k) = R(S_1, G^o) \cup F_1(R(S_1, G^o)) \cup \cdots \cup F_{k-1}(R(S_1, G^o))$. We can obtain

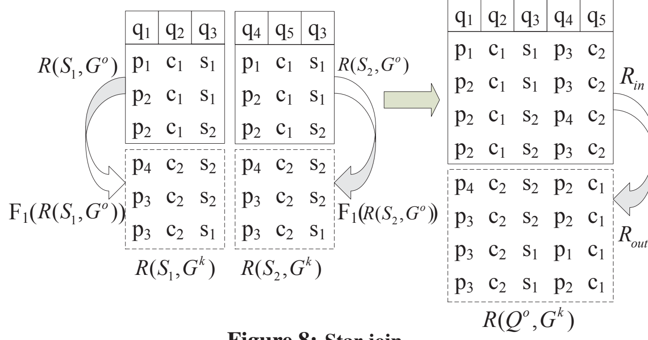



**Figure 8:** **Star join.**

$R(S_2, G^k)$ in a similar fashion. Figure 8 illustrates this process for our running example where $k = 2$.

Finally, we need to join $R(S_1, G^k)$ and $R(S_2, G^k)$ based on the topology relation between $S_1$ and $S_2$. Since $S_1$ and $S_2$ shares only one common vertex $q_3$ in this case, the join condition is that $R(S_1, G^k).q_3 = R(S_2, G^k).q_3$ but the other two columns of $R(S_1, G^k)$ and $R(S_2, G^k)$ do not match. The join results are $R(Q^o, G^k)$.

In this subsection, we propose an efficient technique to speed up the above process by leveraging the symmetry of the $k$-automorphic graph $G^k$. We conceptually divide $R(Q^o, G^k)$ into two parts: $R_{in}$ and $R_{out}$. We only compute the subgraph matches in $R_{in}$ through the above join process. The subgraph matches in $R_{out}$ can be easily computed based on the automorphic functions (such as in Figure 8) without going through the expensive join process. For example, in Figure 8, we only expand $R(S_2, G^o)$ to $R(S_2, G^k)$ using the automorphic function. Then, we join $R(S_1, G^o)$ with $R(S_2, G^k)$ to obtain $R_{in}$. All other matches can be obtained by applying the automorphic function over matches in $R_{in}$. For $n$ stars, we can apply this idea recursively over two star matches at a time, which can be done in parallel as well.

Consider a vertex $q_1$ in $Q^o$. For any match $M'$ of $Q^o$ over $G^k$, let $v'$ denote the matching vertex of $q_1$ in this match. There are only two cases: $v'$ is in $B_1$ (the first block of $G^k$) or not. If $v' \notin B_1$, there must exist another match $M$ under automorphic function $F_1$ (i.e., $M' = F_1(M)$), where $v$ is the matching vertex of $q_1$ in match $M$ and $v$ is in $B_1$. For example, in match $(p_1, c_1, s_1, p_3, c_2)$ in Figure 8, $p_1$ corresponds to $q_1$, where $p_1$ is in block $B_1$. In match $(p_4, c_2, s_2, p_2, c_1)$, $p_4$ corresponds to $q_1$ but $p_4$ is not in block $B_1$; but we know that $(p_4, c_2, s_2, p_2, c_1) = F_1(p_1, c_1, s_1, p_3, c_2)$, where $F_1$ is an automorphic function. In the latter match, $p_1$ matches $q_1$ and $p_1$ is in block $B_1$. Formally, we have the following theorem.

**Theorem** 3. *Given $Q^o$ with $m$ vertices $q_a$ ($a = 1, \cdots, m$), $M'$ with $m$ vertices $v'_a$ ($a = 1, \cdots, m$) is a subgraph match of $Q^o$ over graph $G^k$, where $v'_a$ matches $q_a$. Consider a vertex $q_a$ in $Q^o$. If vertex $v'_a$ matching $q_a$ is not in $B_1$ (the first block of $G^k$), we can find another match $M$ that contains a vertex $v_a$ matching $q_a$, where $v_a \in B_1$ and $v'_a = F_j(v_a)$ under some automorphic function $F_j$. Furthermore, $M' = F_j(M)$, where $F_j(M)$ is a mapping graph of $M$ under an automorphic function $F_j$, which is defined in Definition 4.*

Let us consider a vertex $q_a$ in $Q^o$. Given $q_a$, the set of subgraph matches $R(Q^o, G^k)$ can be divided into the following two parts:

1. $R_{in}$=$\{M | M \in R(Q^o, G^k) \wedge (\exists v_a \in B_1, v_a \in M \wedge v_a \leftrightarrow q_a)\}$
2. $R_{out}$=$\{M | M \in R(Q^o, G^k) \wedge (\exists v_a \notin B_1, v_a \in M \wedge v_a \leftrightarrow q_a)\}$

where $v_a \leftrightarrow q_a$ means that vertex $v_a$ matches $q_a$.

We only need to find subgraph matches in $R_{in}$. Then based on the automorphic functions, the client can obtain $R_{out}$. Note that the cloud server can do this step too if the goal is to minimize the processing cost at the client side, but with a higher communication overhead. Finally, $R(Q^o, G^k) = R_{in} \cup R_{out}$.

Algorithm 2 presents the join algorithm. Let us consider a query decomposition $\{S_1, \ldots, S_n\}$ over $Q^o$. Suppose that $|R(S_a)|$ ($1 \le a \le n$) is the minimum over all stars, and without loss of generality star $S_a$ roots at vertex $q_a$. Initially, we set answer set $R_{in} = R(S_a, G^o)$. We start with $S_a$ and find another star $S_i$ ($1 \le i \ne a \le n$), where $S_i$ overlaps with $S_a$ and $|R(S_i)|$ is minimum over all such overlapping stars. Then, we compute $R(S_i, G^k)$ according to the automorphic functions $F_j$ ($j = 0, \cdots, k-1$) (Line 5-Line 8). We perform the natural join $R_{in} = R_{in} \bowtie R(S_i, G^k)$ (Line 9). Then we delete all the matches that contain duplicate vertices (Line 10-Line 12), since two query vertices cannot match the same vertex in the data graph, according to the definition of subgraph isomorphism. We iterate the above process until all stars in the query decomposition have been processed.

**Algorithm 2** Result Join Algorithm

**Require:** Input: $RS$ (the set of $R(S_i, G^o)$, $1 \le i \le n$) and Alignment Vertex Table (AVT).
Output: $R_{in}$.
1: Initialize $R_{in} := R(S_a, G^o)$, where $|R(S_a)|$ is minimum over all stars.
2: $RS := RS - R_{in}$.
3: **while** $RS$.size() > 0 **do**
4: Initialize a set $R_{next} := R(S_i, G^o)$ ($1 \le i \ne a \le n$), where $S_i$ overlaps with the part of query graph that corresponds to current matches in $R_{in}$, and $|R(S_i)|$ is minimum over all such overlapping stars.
5: Initialize a set $R^k_{next} := \phi$.
6: **for** $m := 0$ to $k - 1$ **do**
7: $R^k_{next} := R^k_{next} \cup F_m(R_{next})$.
8: /* $F_m(R_{next}) = \{M | \exists M' \in R_{next}, M = F_m(M')\}$. */
9: $R_{in} := R_{in} \bowtie R^k_{next}$.
10: **for** each match $M \in R_{in}$ **do**
11: **if** $M$ contains duplicate vertices **then**
12: $R_{in} := R_{in} - \{M\}$.
13: $RS := RS - R_{next}$.
14: Return $R_{in}$.

#### 4.2.2 Processing in the Client Side

According to Algorithm 2, the cloud obtains the set $R_{in}$ and transmits it to the client. There are two stages in the client side processing. The pseudocode is given in Algorithm 3.

First, we need to compute $R_{out}$ according to the automorphic functions $F_j$ ($j = 1, \cdots, k-1$) (Line 1-Line 4 in Algorithm 3). For each match $M$ in $R_{in}$, we compute $F_j(M)$ ($j = 1, \cdots, k-1$) and put them into $R_{out}$. Obviously, $R(Q^o, G^k) = R_{in} \cup R_{out}$ (Line 5). Note that this step can also be done by the cloud.

Secondly, the client computes the final result set $R(Q, G)$ by filtering out the false positives in $R(Q^o, G^k)$ (Line 6-Line 23). The filtering process also has two steps. In the first step, we remove some matches in $R(Q^o, G^k)$ that contain vertices or edges that do not exist in the original graph $G$ (Line 8-Line 20). Also note that we anonymize the vertex labels in the query graph by using vertex label groups. Thus in the second step, we need to filter out matches that contain vertices whose labels cannot match those of the corresponding vertices in the original query graph $Q$ (Line 21-Line 22).

It is straightforward to see that the time complexity of the client processing is linear with the number of matches in $R(Q^o, G^k)$. Furthermore, it is easy to design some hashing techniques to speed up the filtering processing.

## 5. COST MODEL

In order to reduce the search space of subgraph matching, we need a cost model for both label generalization and query decomposition. Different label combinations lead to different search s-

**Algorithm 3** Result Processing Algorithm

**Require:** Input: $R_{in}$ (the set of candidate matching results generated in cloud), Alignment Vertex Table (AVT).
Output: $R$ (the set of final matching results).
1: Initialize $R_{out} := \phi$.
2: **for** $m := 1$ to $k-1$ **do**
3: $R_{out} := R_{out} \cup F_m(R_{in})$.
4: /* $F_m(R_{in}) = \{M | \exists M' \in R_{in}, M = F_m(M')\}$. */
5: Set $R^k := R_{in} \cup R_{out}$.
6: Initialize $R := \phi$.
7: **for** each match $M \in R^k$ **do**
8: find := **true;**
9: **for** each vertex $v \in V(M)$ **do**
10: **if** $v \notin V(G)$ **then**
11: find := **false;**
12: **break;**
13: **if** find = **false then**
14: **continue;**
15: **for** each edge $e \in E(M)$ **do**
16: **if** $e \notin E(G)$ **then**
17: find := **false;**
18: **break;**
19: **if** find = **false then**
20: **continue;**
21: **if** $M$ contains any vertices whose labels do not match those of the corresponding vertices on query $Q$ **then**
22: **continue;**
23: $R := R \cup \{M\}$.
24: Return $R$.

paces in subgraph matching. Since our subgraph matching algorithm is based on joining star matches, we require that the number of star matches ($|R(S)|$) should be as small as possible. Furthermore, the query decomposition method in Section 4.2.1 also relies on estimating $|R(S)|$. Thus, we propose a cost model for estimating $|R(S)|$.

## 5.1 Estimating $|R(S)|$

Given a star query $S$ with center $q$, it matches a star with the center $v$ in $B_1$ (i.e., the first block of $G^k$) if and only if the following two conditions hold. Therefore, we should consider the two factors while estimating $|R(S)|$.

- center $q$ should match $v$;
- each neighbor of $q$ should match one of $v$'s neighbors.

**First Factor.** The first factor measures the number of candidate matching vertices of the center $q$ of the star query $S$. Given an outsourced graph $G^o$, the vertices matching the center $q$ of $S$ only locate in the first block of $G^k$, i.e., block $B_1$. That is to say, we need to find the number of such candidate vertices $v$ in $B_1$, where each $v$ and $q$ share the same vertex type and each such candidate $v$ contains $q$'s vertex label group. $\frac{|V(G^k)|}{k}$ is the number of vertices in $B_1$. Given a center $q$, we should estimate the probability that a vertex $v \in B_1$ has the same vertex type with $q$ and $v$ contains $q$'s vertex label group.

Due to the symmetry in $G^k$, the first block $B_1$ has the same vertex label distribution with $G^k$. Thus, we use $G^k$ to derive the probability. Let $V(G^k, j)$, $V_l(G^k, (j,i))$, and $V_g(G^k, (j,i))$ denote the set of vertices with the $j$-th vertex type, label $l_{j,\,i}$ ($i$th label of $j$th vertex type), and label group $L_{j,\,i}$ respectively. Then, we define:

$$F_{G^k}(j) = \frac{|V(G^k, j)|}{|V(G^k)|}, F^l_{G^k}(j,i) = \frac{|V_l(G^k, (j,i))|}{|V(G^k, j)|}, F^g_{G^k}(j,i) = \frac{|V_g(G^k, (j,i))|}{|V(G^k, j)|} \quad (1)$$

for the graph $G^k$.

Intuitively, $F_{G^k}(j)$ estimates the probability of a vertex being the $j$th vertex type; $F^l_{G^k}(j,i)$ estimates the probability of a vertex that's the $j$th vertex type having an $i$th label; lastly, $F^g_{G^k}(j,i)$ estimates the probability a vertex that's the $j$th vertex type having an $i$th label group *after the label generalization*.

Similarly, we can also define $F_S(j)$, $F^l_S(j,i)$ and $F^g_S(j,i)$ for a star query graph $S$, replacing $G^k$ by star $S$ in the equation.

Without loss of generality, assume that for the $j$-th ($1 \le j \le t$) vertex type, there are $\theta h_j$ different labels ($l_{j,1}, l_{j,2}, \cdots, l_{j,\theta h_j}$). These labels can be combined into $h_j$ label groups ($L_{j,1}, L_{j,2}, \cdots, L_{j,h_j}$). The $i$-th group $L_{j,i}$ contains $\theta$ different labels, which are $l_{j,\, p_{\theta(i-1)+1}}, l_{j,\, p_{\theta(i-1)+2}}, \cdots, l_{j,\, p_{\theta i}}$ for $1 \le i \le h_j$. Note that $\langle p_1, p_2, \cdots, p_{\theta h_j}\rangle$ forms a permutation of $\{1, 2, \cdots, \theta h_j\}$.

$F_{G^k}(j) F_S(j)$ is the probability that a vertex $v$ in $G^k$ has the same vertex type with the star center $q$. Consider each possible vertex type. For the $j$-th vertex type, $\sum_{i=1}^{h_j} F^g_{G^k}(j,i) F^g_S(j,i)$ denotes the probability that vertex $v$ and the query center has the same label group. Therefore, the first factor estimating the number of vertices that can match the center of the star query is as follows.

$$\frac{|V(G^k)|}{k} \sum_{j=1}^{t} \left[ F_{G^k}(j) F_S(j) \sum_{i=1}^{h_j} F^g_{G^k}(j,i) F^g_S(j,i) \right] \quad (2)$$

**Second Factor.** The second factor measures the search space of checking whether each of $q$'s neighbors can find its matching vertex in $v$'s neighbors. The estimation here is similar with that of the first factor. The difference is that the candidate matching vertex $v$ of the center $q$ of $S$ has been given. Thus, to match the first vertex of $q$'s neighbors, the number of candidate vertices that we should search is the degree of vertex $v$ rather than $\frac{|V(G^k)|}{k}$. Since there are several candidate vertices $v$, we use the average degree of vertices in $G^k$ to estimate the degree of vertex $v$. Here, we denote it as $D(G^k)$. Then, to match the second vertex of $v$'s neighbors, the potential search space is $D(G^k) - 1$. The rest can be done in the same manner. Suppose the center of the star query $S$ has $D^c(S)$ neighbors. Thus, this part of the search space can be estimated as $D(G^k) \cdot (D(G^k) - 1) \cdots (D(G^k) - D^c(S) + 1)$. For the sake of simplicity, we can estimate it as $D(G^k)^{D^c(S)}$. As with the estimation of the first factor, we should also consider the probability of sharing the same vertex type and containing the corresponding vertex label group.

Thus, we can define the second factor that estimates the search space of matching the star center $q$'s neighbors as follows.

$$\left\{ D(G^k) \sum_{j=1}^{t} \left[ F_{G^k}(j) F_S(j) \sum_{i=1}^{h_j} F^g_{G^k}(j,i) F^g_S(j,i) \right] \right\}^{D^c(S)} \quad (3)$$

**The Cost Model.** Our cost model is given by Expression 4, which is simply the product of factor 1 and factor 2 from (2) and (3).

It does assume independence among the label distributions for the neighbors of a vertex. This may not always hold in practice. But our experimental results show that our assumption is acceptable in three real large graphs and our cost model is very effective. We also note that $|V_g(G^k, (j,i))| \le [1 + \delta(k)] \cdot \sum_{m=1}^{\theta} V_l(G^k, (j, p_{\theta(i-1)+m}))$ for some constant $0 \le \delta(k) \le k-1$, i.e., the number of vertices from the $j$th vertex type having the $i$th label group is at most a constant factor of the total number of vertices with $j$th vertex type having a label that was generalized into this label group. Immediately, this implies that $F^g_{G^k}(j,i) \le [1 + \delta(k)] \cdot \sum_{m=1}^{\theta} F^l_G(j, p_{\theta(i-1)+m})$. Intuitively, each vertex $u$ in $G^k$ has $(k-1)$ symmetric vertices. To ensure that they have the same vertex groups, we require that $u$ should have a union of all its symmetric vertices' label groups. In the worst case, $F^g_{G^k}(j,i)$ will increase by a factor of $(k-1)$. In fact, $\delta(k)$ can be much less than $(k-1)$, and given $G$ and the corresponding $k$-automorphic graph $G^k$, we can give a much tighter bound on the parameter $\delta(k)$. That is to say, if we do not introduce any unnecessary label groups during label generalization (just like the example in Figure 2), $\delta(k)$

will be 0. In practice, $\delta(k)$ is far less than 1 when $k$ is small, as demonstrated in our experiments.

$$
\begin{aligned}
&|R(S)| \\
&= \frac{|V(G^k)|}{k} \sum_{j=1}^{t} \left[ F_{G^k}(j) F_S(j) \sum_{i=1}^{h_j} F^g_{G^k}(j,i) F^g_S(j,i) \right] \\
&\quad \cdot \left\{ D(G^k) \sum_{j=1}^{t} \left[ F_{G^k}(j) F_S(j) \sum_{i=1}^{h_j} F^g_{G^k}(j,i) F^g_S(j,i) \right] \right\}^{D^c(S)} \\
&= \left\{ \sum_{j=1}^{t} \left[ F_{G^k}(j) F_S(j) \sum_{i=1}^{h_j} F^g_{G^k}(j,i) F^g_S(j,i) \right] \right\}^{D^c(S)+1} \cdot \frac{|V(G^k)| D(G^k)^{D^c(S)}}{k} \\
&\leq \left\{ \sum_{j=1}^{t} \left\{ F_{G^k}(j) F_S(j) \sum_{i=1}^{h_j} \left[ \sum_{m=1}^{\theta} F^l_G(j, p_{\theta(i-1)+m}) \right] \left[ \sum_{m=1}^{\theta} F^l_S(j, p_{\theta(i-1)+m}) \right] \right\} \right\}^{D^c(S)+1} \\
&\quad \cdot \frac{|V(G^k)| D(G^k)^{D^c(S)} [1+\delta(k)]^{D^c(S)+1}}{k}
\end{aligned}
\tag{4}
$$

## 5.2 Label Combination

The cost model based query decomposition has been studied in Section 4.2.1. Hence, we only need to show how to do label generalization (label combination) based on the cost model above. Our solution works as follows. Let us consider the $j$-th vertex type and its $\theta h_j$ different vertex labels $(l_{i,1}, l_{i,2}, \cdots, l_{i,\theta h_j})$. Assume that $P = \langle p_1, p_2, \cdots, p_{\theta h_j} \rangle$ is a permutation of $\{1, 2, \cdots, \theta h_j\}$. We divide $P$ sequentially into $h_j$ groups, where each group has $\theta$ vertex labels. Each permutation corresponds to one possible label combination. The goal is to find the optimal permutation, which leads to the minimum cost in star matching.

In the derivation of Expression 4, we only estimate the search space of a single star query $S$. However, to perform the label combination of the $k$-automorphic graph $G^k$, we need to estimate the search space of star queries in the average case. The average case can be estimated by exploring different query patterns. Suppose $S_{set}$ is the set of all possible star queries. Similar to the definitions in Equation 1 in Section 5.1, we can define

$$
F_{S_{avg}}(j) = \frac{\sum_{s \in S_{set}} \frac{|V(s,j)|}{|V(s)|}}{|S_{set}|}, \; F^l_{S_{avg}}(j,i) = \frac{\sum_{s \in S_{set}} \frac{|V_l(s,(j,i))|}{|V(s,j)|}}{|S_{set}|}, \; F^g_{S_{avg}}(j,i) = \frac{\sum_{s \in S_{set}} \frac{|V_g(s,(j,i))|}{|V(s,j)|}}{|S_{set}|}
$$

for star queries. These values rely on all possible star queries rather than any one single star query. Similarly, let $D^c(S_{avg}) = \frac{\sum_{s \in S_{set}} D^c(s)}{|S_{set}|}$.

Now similar to the derivation of Expression 4, we can estimate the search space of star queries in the average case, denoted as $|R(S_{avg})|$, in Expression 5 (the derivation follows the same steps as that in Expression 4). According to Expression 5, the component that concerns the label combination is given in Expression 6.

$$
\begin{aligned}
&|R(S_{avg})| \\
&\leq \left\{ \sum_{j=1}^{t} \left\{ F_{G^k}(j) F_{S_{avg}}(j) \sum_{i=1}^{h_j} \left[ \sum_{m=1}^{\theta} F^l_G(j, p_{\theta(i-1)+m}) \right] \left[ \sum_{m=1}^{\theta} F^l_{S_{avg}}(j, p_{\theta(i-1)+m}) \right] \right\} \right\}^{D^c(S_{avg})+1} \\
&\quad \cdot \frac{|V(G^k)| D(G^k)^{D^c(S_{avg})} [1+\delta(k)]^{D^c(S_{avg})+1}}{k}
\end{aligned}
\tag{5}
$$

Therefore, we define the label combination cost as follows.

**Definition** 7. ***Label Combination Cost.*** *Given the $j$-th vertex type and its $\theta h_j$ different vertex labels $(l_{j,1}, l_{j,2}, \cdots, l_{j,\theta h_j})$, we can form a list $\{(F^l_G(j,1), F^l_{S_{avg}}(j,1)), (F^l_G(j,2), F^l_{S_{avg}}(j,2)), \cdots, (F^l_G(j,\theta h_j), F^l_{S_{avg}}(j,\theta h_j))\}$, where $\sum_{i=1}^{\theta h_j} F^l_G(j,i) = 1$ and $\sum_{i=1}^{\theta h_j} F^l_{S_{avg}}(j,i) = 1$. Given $P = \langle p_1, p_2, \cdots, p_{\theta h_j} \rangle$ as a permutation of $\{1, 2, \cdots, \theta h_j\}$, the* label combination cost *of $P$ for the $j$-th vertex type is defined as follows:*

$$
cost(P) = \sum_{i=1}^{h_j} \left[ \sum_{m=1}^{\theta} F^l_G(j, p_{\theta(i-1)+m}) \right] \left[ \sum_{m=1}^{\theta} F^l_{S_{avg}}(j, p_{\theta(i-1)+m}) \right] \tag{6}
$$

*/*G refers to the original data graph in this definition.*/*

Iteration 1

| i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Label | $l_{j,1}$ | $l_{j,2}$ | $l_{j,3}$ | $l_{j,4}$ | $l_{j,5}$ | $l_{j,6}$ |
| $F_G{}^l(j, p_i)$ | 0.05 | 0.1 | 0.15 | 0.2 | 0.25 | 0.25 |
| $F_{Savg}{}^l(j, p_i)$ | 0.15 | 0.05 | 0.2 | 0.3 | 0.1 | 0.2 |

| i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Label | $l_{j,1}$ | $l_{j,4}$ | $l_{j,3}$ | $l_{j,2}$ | $l_{j,5}$ | $l_{j,6}$ |
| $F_G{}^l(j, p_i)$ | 0.05 | 0.2 | 0.15 | 0.1 | 0.25 | 0.25 |
| $F_{Savg}{}^l(j, p_i)$ | 0.15 | 0.3 | 0.2 | 0.05 | 0.1 | 0.2 |

| i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Label | $l_{j,1}$ | $l_{j,4}$ | $l_{j,3}$ | $l_{j,5}$ | $l_{j,2}$ | $l_{j,6}$ |
| $F_G{}^l(j, p_i)$ | 0.05 | 0.2 | 0.15 | 0.25 | 0.1 | 0.25 |
| $F_{Savg}{}^l(j, p_i)$ | 0.15 | 0.3 | 0.2 | 0.1 | 0.05 | 0.2 |

Iteration 2

| i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Label | $l_{j,1}$ | $l_{j,4}$ | $l_{j,3}$ | $l_{j,5}$ | $l_{j,2}$ | $l_{j,6}$ |
| $F_G{}^l(j, p_i)$ | 0.05 | 0.2 | 0.15 | 0.25 | 0.1 | 0.25 |
| $F_{Savg}{}^l(j, p_i)$ | 0.15 | 0.3 | 0.2 | 0.1 | 0.05 | 0.2 |

**Figure 9: An example of label anonymization algorithm.**

Thus we need to choose an effective permutation $P$ to decrease $cost(P)$ as much as possible. To achieve that, we propose a heuristic solution. It works in an iterative manner. Initially, we generate a random label combination. Then, in each iteration, we try to swap two labels in two label groups, respectively. If the swap leads to smaller cost (see Expression 6), we keep the swap; otherwise, we ignore that. For example, we keep the swap between $l_{j,2}$ and $l_{j,4}$ (see Figure 9) in the first iteration. We consider all possible swaps sequentially. Once we cannot find a swap that leads to smaller cost, the algorithm stops. Our experimental results show that for each of the three datasets in Section 6 and each vertex type, we generally need no more than 10 such iterations before the results converge. An example of label anonymization algorithm for $\theta = 2$ (two lables in each label group) is represented in Figure 9, using our running example.

# 6. EXPERIMENTAL RESULTS

## 6.1 Datasets and Setup

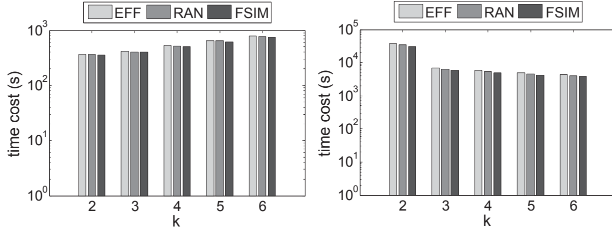
We evaluated our method on three real datasets in our experiments. Statistics on these graphs are given in Table 2.

**Table 2: Real graph datasets.**

| Dataset | $\|V\|$ | $\|E\|$ | # of Types | # of Attributes | # of Labels |
|---|---|---|---|---|---|
| Web-NotreDame | 325,729 | 109,0108 | 1 | 1 | 200 |
| DBpedia | 3,243,606 | 8,588,047 | 86 | 101 | 6300 |
| UK-2002 | 18,520,486 | 261,787,258 | 2500 | 2500 | 20,000 |

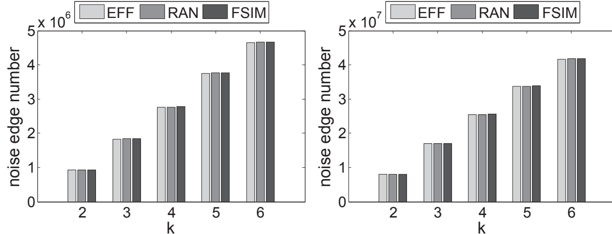
**Web-NotreDame**. Web-NotreDame is a web graph collected in 1999. Vertices represent pages from the University of Notre Dame and edges represent hyperlinks between them.

**DBpedia**. DBpedia is a crowd-sourced community effort to extract structured information from Wikipedia and generate a large semantic knowledge graph. Vertices correspond to entities in DBpedia and edges correspond to relationships among them. According to type information and property information from the dataset, we extract types, attributes and attribute values and add them for all the vertices.




(a) Web-NotreDame (b) DBpedia

**Figure 10: Time cost in generating $G^k$.**




(a) Web-NotreDame (b) DBpedia

**Figure 11: Number of noise edges in $G^k$.**

| Dataset | | k = 2 | k = 3 | k = 4 | k = 5 | k = 6 |
|---|---|---|---|---|---|---|
| Web-NotreDame | $\|E(G^o)\|$ | 1024152 | 1044054 | 1092048 | 1131386 | 1163919 |
| | $\|E(G^k)\|$ | 2013828 | 2923686 | 3850740 | 4848650 | 5743884 |
| DBpedia | $\|E(G^o)\|$ | 13000836 | 19106315 | 23923355 | 27285461 | 32653869 |
| | $\|E(G^k)\|$ | 16582848 | 25561791 | 33971336 | 42268695 | 50278758 |
| UK-2002 | $\|E(G^o)\|$ | 351912573 | 502482658 | 618590274 | 737819332 | 848864429 |
| | $\|E(G^k)\|$ | 487677404 | 713567550 | 952457696 | 1202931182 | 1460772185 |

**Figure 12: Number of edges in $G^o$ and $G^k$ using EFF.**

**UK-2002**. UK-2002 is obtained from a crawl of the .uk domain performed by UbiCrawler[3] in 2002.

We find that the frequencies of different vertex labels on these graphs all (roughly) obey Zipf's law of different skewness.

**SETUP.** We compare four methods EFF, BAS, FSIM and RAN, where EFF is our method with all optimizations discussed.

1. EFF applies both the "cost model based" label generalization and the graph size reduction (i.e., uploading $G^o$).

2. BAS applies the same cost model based label generalization approach with EFF; but BAS uploads the entire $G^k$ directly.

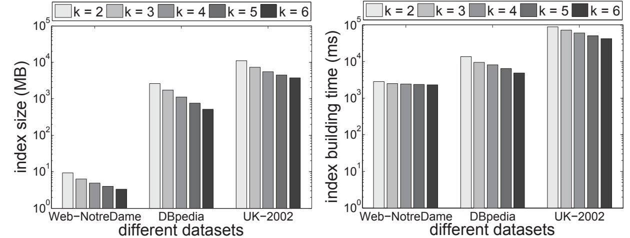
3. FSIM and RAN adopt two different label generalization approaches. FSIM combines vertex labels with similar frequencies in the data graph into the same label groups, while RAN randomly combines vertex labels into label groups. However, *both FSIM and RAN adopt the graph size reduction approach proposed in this paper*, i.e., they upload $G^o$ instead of $G^k$ to the cloud.

We use a Windows 7 PC with 3.0 GHz Intel Core 2 Duo CPU and 8 GB main memory as the client side. The cloud server is on a virtualized Windows machine within Microsoft Azure cloud, providing 64 GB main memory and four CPU cores. All methods are implemented in C++. Some additional experimental results are presented in Appendix B.

## 6.2 Cost of Generating $G^k$ and $G^o$

In this subsection, we evaluate the performance of generating $G^k$ and $G^o$. The default value of $\theta$ (the number of labels contained in each label group) is 2 in all the experiments. The main results of these experiments are as follows.

[3]http://law.di.unimi.it/webdata/uk-2002




(a) Index size. (b) Index construction time.

**Figure 13: Index cost of our algorithm.**

The cost model based label anonymization approach EFF is as efficient as simple strategies like RAN and FSIM when generating $G^k$, as shown in Figure 10. The cost on Web-NotreDame slightly increases when $k$ goes from 2 to 6, but the running time on DBpedia decreases as $k$ becomes larger. The reason is that there are a few high-degree vertices on DBpedia. Since we use the BFS strategy in graph alignment for generating $G^k$, when $k$ is small, such as 2, we have to do more explorations on these high-degree vertices. The experimental results on UK-2002 are similar with those on DBpedia; see Appendix B.

The number of noise edges ($|E(G^k)| - |E(G)|$) does not depend on the label anonymization approach. We find that the three approaches introduce nearly the same number of noise edges (as shown in Figure 11). The number of noise edges roughly grows linearly when $k$ goes from 2 to 6. The experimental results on UK-2002 are also similar with those on the other two datasets; see Appendix B.

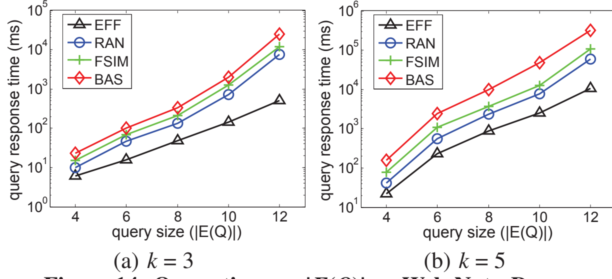
We also report the number of edges in $G^o$ and $G^k$ (i.e., space and communication cost) in Figure 12. Since the three approaches introduce nearly the same number of noise edges, we only report the results generated by EFF method. As shown in Figure 12, The edge number of $G^o$ is much less than that of $G^k$. Especially, when $k$ is small, $|E(G^o)|$ is close to $|E(G)|$. It shows that our method not only ensures the data privacy but also saves the online query cost, since online subgraph matching is performed on $G^o$ rather than $G^k$.

Although the three label anonymization algorithms have similar efficiency in generating $G^k$, they do generate different $G^k$'s. EFF produces $G^k$'s that perform much better than the graphs produced by the other two methods in terms of the query performance. We will report these results shortly.

## 6.3 Performance in the Cloud

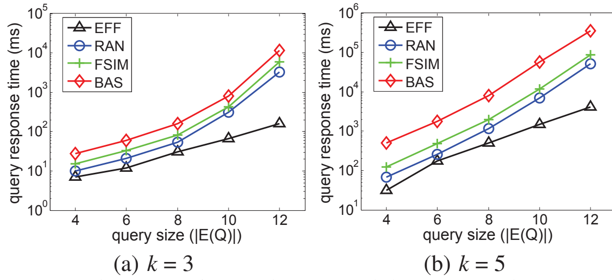
We first report the cost of index construction for our method. Figure 13 shows the space and time cost of constructing the index structure, which is discussed in Section 4.2.1. Both space and time costs of our index construction decrease while $k$ increases from 2 to 6. The reason lies in the fact that the size of our index (see Figure 7) depends on the vertex number of $G^o$ (i.e., $|V(G^o)|$). Obviously, $|V(G^o)|$ decreases with the increase of $k$, since $G^o$ roughly contains only the first block of vertices in $G^k$ and each block in $G^k$ has about $\frac{|V(G^k)|}{k}$ vertices. Thus, larger $k$ leads to less index size and less construction cost.

Then we pay attention to the cost of subgraph matching in the cloud. We generate query graphs by randomly extracting connected subgraphs from the data graph $G$, ensuring that $|E(Q)|$ meets a user-specified parameter value $N$. Specifically, we randomly locate the first edge $e$ from the data graph $G$ and set $E(Q) = \{e\}$. We then expand the current query graph $Q$ through a random walk over $G$ iteratively until it reaches $N$ edges. We used 100 queries and report the average. We compared our method with the baseline BAS and the other two label generalization techniques. First, we report the query performance for different query graph sizes $|E(Q)|$. Figure



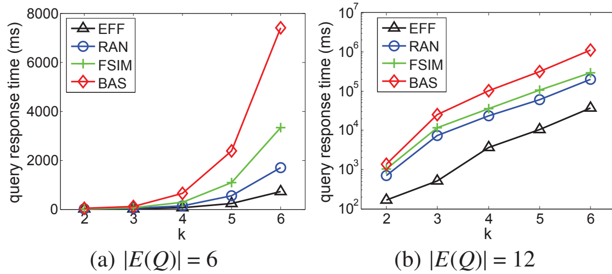

(a) $k = 3$ (b) $k = 5$

**Figure 14: Query time vs. $|E(Q)|$ on Web-NotreDame.**



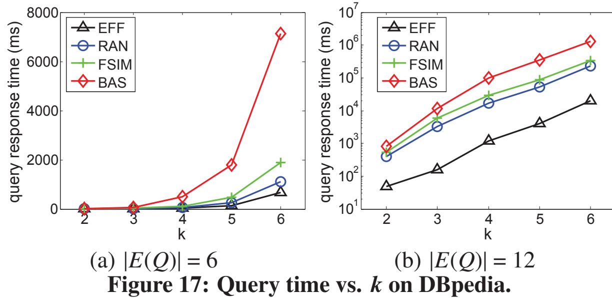

(a) $k = 3$ (b) $k = 5$

**Figure 15: Query time vs. $|E(Q)|$ on DBpedia.**




(a) $|E(Q)| = 6$ (b) $|E(Q)| = 12$

**Figure 16: Query time vs. $k$ on Web-NotreDame.**




(a) $|E(Q)| = 6$ (b) $|E(Q)| = 12$

**Figure 17: Query time vs. $k$ on DBpedia.**

14 shows that our method EFF performs much better than the other three approaches on Web-NotreDame. Note that RAN and FSIM use the same processing framework as EFF (i.e., they rely on $G^o$ and our cloud processing algorithms as well); they only differ from EFF in how to generate the label combinations. Thus, the experimental results confirm the effectiveness of our cost model and the cost model based label combination.

We find similar results on DBpedia, as shown in Figure 15. The experimental results on UK-2002 are also similar with those on other datasets; see Appendix B. When the query size becomes larger, EFF outperforms the other three approaches by at least one order of magnitude. Furthermore, BAS always performs the worst, since the search space on $G^k$ is much larger than that on $G^o$.

The running time of subgraph matching increases with $k$ from 2 to 6, as shown in Figure 16 and Figure 17. This is because that $|E(G^o)|$ increases with $k$ from 2 to 6, since we must insert more noise edges into $G^o$ when $k$ becomes larger. Nevertheless, EFF is always the best method. Furthermore, compared with other approaches, the advantage of EFF becomes even more significant for larger values of $k$.

| Dataset | | k = 3 | | k = 5 | |
|---|---|---|---|---|---|
| | | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 |
| Web-NotreDame | EFF | 9 | 31 | 42 | 137 |
| | RAN | 19 | 69 | 89 | 288 |
| | FSIM | 25 | 87 | 117 | 374 |
| DBpedia | EFF | 7 | 19 | 24 | 54 |
| | RAN | 11 | 36 | 41 | 110 |
| | FSIM | 14 | 50 | 50 | 149 |
| UK-2002 | EFF | 7 | 24 | 34 | 125 |
| | RAN | 11 | 38 | 53 | 196 |
| | FSIM | 15 | 52 | 72 | 267 |

Star Matching Time (ms)

**Figure 18: Star matching time.**

| Dataset | | k = 3 | | k = 5 | |
|---|---|---|---|---|---|
| | | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 |
| Web-NotreDame | EFF | 96 | 188 | 487 | 696 |
| | RAN | 187 | 556 | 792 | 2089 |
| | FSIM | 252 | 729 | 1100 | 2858 |
| DBpedia | EFF | 64 | 126 | 296 | 431 |
| | RAN | 125 | 402 | 581 | 1920 |
| | FSIM | 163 | 478 | 759 | 2544 |
| UK-2002 | EFF | 91 | 160 | 454 | 663 |
| | RAN | 127 | 431 | 638 | 1779 |
| | FSIM | 210 | 528 | 722 | 2668 |

|RS|

**Figure 19: Result set size of the star matching algorithm ($|RS|$).**

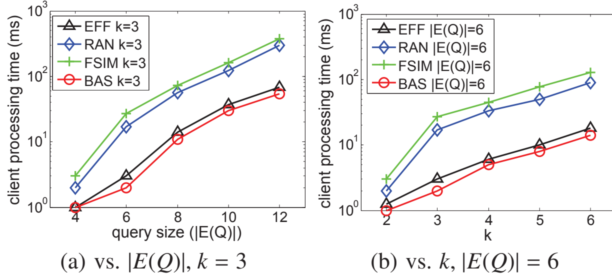
We also find that both the query decomposition algorithm and the star matching algorithm run very fast. Even when $|E(Q)|$ is as large as 12, the time cost of query decomposition algorithm is less than 1 ms. As for the time cost of star matching algorithm, Figure 18 shows the experimental results which are also very efficient. We also report the size of the set $RS$ generated by our star matching algorithm in Figure 19. $|RS|$ has a large influence on the time cost of result join algorithm. These results show the importance of optimizing the result join algorithm.

## 6.4 Processing Cost in the Client Side

The last step of all methods involves processing in the client side. Figure 20 and Figure 21 illustrate this overhead on the first two real graphs. The goal of client side processing is to filter out all false positive matches due to the noise edges and labels in $G^k$.

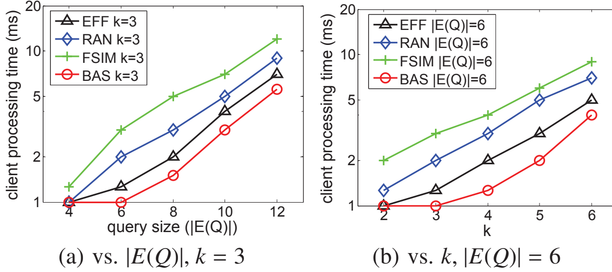
First of all, note that for all methods, client processing cost is much less than cloud processing cost; it is orders of magnitude less. As Figure 20 and Figure 21 show, the overhead in the client side scales very well in terms of query size and $k$. EFF still outperforms both RAN and FSIM, but it is slightly worse than BAS in the client side. The reason is that EFF generates fewer intermediate results than RAN and FSIM, due to its effective label combination algorithm. However, following our processing framework, EFF, RAN, and FSIM produce $R_{in}$ (i.e., a small subset of $R(Q^o, G^k)$, see Section 4.2.1) in the cloud, and the client needs to find $R(Q^o, G^k)$ based on $R_{in}$ and our client side algorithm as shown in Section 4.2.2. In contrast, BAS obtains $R(Q^o, G^k)$ from the cloud directly (since the cloud has $G^k$ instead of $G^o$ as that in our method).

However, note that client processing time is a tiny fraction of the entire processing time. Furthermore, compared with our method like EFF, BAS has much more expensive communication cost, since $|R(Q^o, G^k)|$ is always much larger than $|R_{in}|$ (i.e., the size of a small subset of $R(Q^o, G^k)$, see Section 4.2.1). For example, the average size of matching results for $|E(Q)| = 6, k = 4$ on UK-2002 in BAS is 12,224 bytes, while EFF only transmits 3,056 bytes. Thus




(a) vs. $|E(Q)|$, $k = 3$ (b) vs. $k$, $|E(Q)| = 6$

**Figure 20: Client processing time on Web-NotreDame.**




(a) vs. $|E(Q)|$, $k = 3$ (b) vs. $k$, $|E(Q)| = 6$

**Figure 21: Client processing time on DBpedia.**

| Dataset | | k = 3 | | k = 5 | |
|---|---|---|---|---|---|
| | | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 |
| Web-NotreDame | EFF | 21 | 608 | 249 | 10948 |
| | RAN | 73 | 7941 | 624 | 61306 |
| | FSIM | 110 | 12329 | 1197 | 108880 |
| | BAS | 109 | 25240 | 2423 | 318390 |
| DBpedia | EFF | 14 | 170 | 139 | 4144 |
| | RAN | 24 | 3263 | 265 | 52212 |
| | FSIM | 38 | 6048 | 492 | 86412 |
| | BAS | 64 | 11725 | 1805 | 350867 |
| UK-2002 | EFF | 19 | 465 | 203 | 10076 |
| | RAN | 50 | 4721 | 400 | 50651 |
| | FSIM | 76 | 7956 | 550 | 96353 |
| | BAS | 112 | 17859 | 2381 | 277208 |

Overall Running Time (ms)

**Figure 22: Overall running time.**

the average communication time for transmitting results in BAS is 11 ms, but EFF only spends 3 ms. More importantly, as discussed in Section 4, the step of finding $R(Q^o, G^k)$ based on $R_{in}$ in EFF can be easily moved to the cloud side if the priority is to lower the client processing cost and communication cost is a lesser consideration. As stated above, EFF runs much faster than other methods in terms of the overall running time, which is shown in the next subsection.

## 6.5 Overall Online Performance

Lastly, we report the end-to-end running time for privacy preserving subgraph matching queries in the cloud in Figure 22. The end-to-end time consists of three parts: the running time in the cloud, the network communication time, and the running time in the client side. Figure 22 shows that our method EFF, the cost model based label anonymization with graph size reducation approach, has the best overall running time. In particular, EFF outperforms other competing methods significantly.

# 7. RELATED WORK

**Privacy preserving graph data publication.** Lots of recent works pay attention to privacy protection for graph publication. The majority of these works [13, 24, 21, 26, 6] focus on protecting data privacy from structural attacks, they can be used as a baseline for our problem with significant overheads.

Some existing works [13, 24, 21] assume that the attacker only launches one type of structural attack. In fact, an attacker can launch multiple types of structural attacks to identify the target based on the strong background knowledge. The data graph produced from some privacy preserving techniques may lose a few edges because of edge delete operations [6]. The loss of structural information in original data graph will lead to the infeasibility of subgraph matching on published data graph. Thus, we utilize $k$-automorphism [26] to protect the data graph from compromising sensitive information. First, a $k$-automorphic graph can defend node re-identification based on any background information of any subgraph [26]. Second, a $k$-automorphic graph will not delete any vertices or edges from the original data graph. Third, the key point of utilizing $k$-automorphism is to enable us to leverage the symmetry of the $k$-automorphic graph, which allows our method to answer subgraph matching queries efficiently.

Differential privacy [8] is an effective model to protect against unknown attacks with guaranteed probabilistic accuracy. But due to the perturbation, these techniques [16, 4, 5] are useful in finding statistical information (e.g. aggregates) from a graph, but are not feasible in answering privacy preserving subgraph matching queries *exactly*.

**Privacy preserving graph query processing in the cloud.** For answering secure shortest path queries, Das *et al.* propose a linear programming method to anonymize edge weights of the original graph while preserving shortest paths [7]. However, any attacker with some topological knowledge can re-identify sensitive information easily. Gao *et al.* transform an original graph into a linked graph and a set of outsourced graphs to support privacy preserving shortest distance queries in the cloud [9]. However, it has much difficulty in answering subgraph matching queries, because of losing the connectivity in the original graph. Cao *et al.* propose a method to answer subgraph matching queries over encrypted graphs in the cloud [3], *for a database of graphs*. The method pre-builds a feature-based index for each data graph. After filtering in the cloud, each candidate supergraph is verified by checking subgraph isomorphism in the client. However, the method does not apply to our case where the goal is to find subgraph matches on a single large graph (instead of many small graphs).

# 8. CONCLUSION

In this paper, we present an efficient framework for privacy preserving subgraph matching on a large graph in the cloud. Our design protects both structural and label privacy in a graph, without losing data utility. By exploring a number of optimization techniques and leveraging an effective cost model, our method achieves superior query performance compared to the baseline method. Extensive experiments on large real graphs confirm the effectiveness and efficiency of our approach. Extending our framework to handling other important graph queries will lead to a number of interesting and important future works.

**Acknowledgment.** This work was supported by 863 project under Grant No. 2015AA015402, NSFC under Grant No. 61532010 and Beijing Higher Education Young Elite Teacher Project (YETP0016). This paper was also supported by "Special Program for Applied Research on Super Computation of the NSFC-Guangdong Joint Fund (the second phase)". Zhao Chang and Feifei Li were supported in part by NSF grant 1514520. Lei Zou is the corresponding author of this work.

## 9. REFERENCES

[1] http://neo4j.org/.
[2] What is cloud computing? *Amazon Web Services*, (3).
[3] N. Cao, Z. Yang, C. Wang, K. Ren, and W. Lou. Privacy-preserving query over encrypted graph-structured data in cloud computing. In *ICDCS*, pages 393–402, 2011.
[4] R. Chen, B. C. M. Fung, P. S. Yu, and B. C. Desai. Correlated network data publication via differential privacy. *VLDB J.*, pages 1–24, 2013.
[5] S. Chen and S. Zhou. Recursive mechanism: Towards node differential privacy and unrestricted joins. In *SIGMOD*, pages 653–664, 2013.
[6] J. Cheng, A. W. Fu, and J. Liu. K-isomorphism: Privacy preserving network publication against structural attacks. In *SIGMOD*, pages 459–470, 2010.
[7] S. Das, Ömer Eğecioğlu, and A. E. Abbadi. Anonymizing weighted social network graphs. In *ICDE*, pages 904–907, 2010.
[8] C. Dwork, F. Mcsherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, 2006.
[9] J. Gao, J. Y. Xu, R. Jin, J. Zhou, T. Wang, and D. Yang. Neighborhood-privacy protected shortest distance computing in cloud. In *SIGMOD*, 2011.
[10] M. Hay, G. Miklau, D. Jensen, D. F. Towsley, and P. Weis. Resisting structural re-identification in anonymized social networks. *PVLDB*, 1(1):102–114, 2008.
[11] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *SC*, 1995.
[12] J. Lee, W. Han, R. Kasperovics, and J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.
[13] K. Liu and E. Terzi. Towards identity anonymization on graphs. In *SIGMOD*, pages 93–106, 2008.
[14] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, 2010.
[15] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.
[16] A. Sala, X. Zhao, C. Wilson, H. Zheng, and B. Y. Zhao. Sharing graphs using differentially private graph models. In *IMC*, pages 679–688, 2011.
[17] P. Samarati and L. Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. *Technical report, SRI International*, 1998.
[18] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9), 2012.
[19] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.
[20] C. Tai, P. Tseng, P. S. Yu, and M. Chen. Identity protection in sequential releases of dynamic networks. *IEEE Trans. Knowl. Data Eng.*, 26(3):635–651, 2014.
[21] C.-H. Tai, P. S. Yu, D.-N. Yang, and M.-S. Chen. Privacy-preserving social network publication against friendship attacks. In *KDD*, pages 1262–1270, 2011.
[22] W. Wu, Y. Xiao, W. Wang, Z. He, and Z. Wang. K-symmetry model for identity anonymization in social networks. In *EDBT*, pages 111–122, 2010.
[23] X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *TODS*, 30(4):960–993, 2005.
[24] B. Zhou and J. Pei. Preserving privacy in social networks against neighborhood attacks. In *ICDE*, pages 506–515, 2008.
[25] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1), 2009.
[26] L. Zou, L. Chen, and M. T. Özsu. K-automorphism: A general framework for privacy preserving network publication. In *VLDB*, 2009.


## Appendix

### A. Proof of Theorems.

**Theorem** 1. *Given a data graph $G$ and a query graph $Q$, $R(Q,G) \subseteq R(Q^o, G^k)$.*

PROOF. For any match $M \in R(Q,G)$, we assume that $g : V(Q) \rightarrow V(G)$ is the injective function. We can let $g' : V(Q^o) \rightarrow V(G^k)$ have the same mapping relation with $g$, since $G$ is a subgraph of $G^k$. Thus we have $\forall q_i \in V(Q^o) \Rightarrow g'(q_i) \in V(G^k)$ and $\forall q_i, q_j \in V(Q^o), \overline{q_i q_j} \in E(Q^o) \Rightarrow \overline{g'(q_i)g'(q_j)} \in E(G^k)$. Since we use the same label grouping strategy on $G$ and $Q$, $\forall q_i \in V(Q^o) \Rightarrow L(q_i) \subseteq L(g'(q_i))$. According to Definition 2, $M \in R(Q^o, G^k)$, represented as $\langle g'(q_1), \cdots, g'(q_n)\rangle$. □

**Theorem** 2. *Given an outsourced query graph $Q^o$, finding the query decomposition with the minimum cost with respect to Definition 6 is an NP-hard problem.*

PROOF. Finding a minimum vertex cover is a classical NP-hard problem. When each $|R(S_i)| = 1$ ($i = 1, \cdots, n$), finding the optimal query decomposition is equivalent to finding a minimum vertex cover. Thus finding a minimum vertex cover is a subproblem of finding the optimal query decomposition. □

**Theorem** 3. *Given $Q^o$ with $m$ vertices $q_a$ ($a = 1, \cdots, m$), $M'$ with $m$ vertices $v'_a$ ($a = 1, \cdots, m$) is a subgraph match of $Q^o$ over graph $G^k$, where $v'_a$ matches $q_a$. Consider a vertex $q_a$ in $Q^o$. If vertex $v'_a$ matching $q_a$ is not in $B_1$ (the first block of $G^k$), we can find another match $M$ that contains a vertex $v_a$ matching $q_a$, where $v_a \in B_1$ and $v'_a = F_j(v_a)$ under some automorphic function $F_j$. Furthermore, $M' = F_j(M)$*[4].

PROOF. Since $M'$ is a subgraph of $G^k$, we have $v'_a \in G^k$. Without loss of generality, assume that $v'_a \in P_i$ ($1 < i \leq k$). We set $v_a = F_{k-(i-1)}(v'_a)$, i.e., $j = i - 1$ and $v'_a = F_j(v_a)$. Thus $v_a \in P_1$.

Here, we prove that $M = F_{k-(i-1)}(M')$ is another match of $Q^o$ over $G^k$. Thus $M' = F_j(M)$.

For the match $M'$, we assume that $g' : V(Q^o) \rightarrow V(G^k)$ is the injective function. Here, we prove that $g = F_{k-(i-1)}(g')$ is the injective function of $M$. Obviously, $\forall q_s \in V(Q^o) \Rightarrow g'(q_s) \in V(G^k) \Rightarrow F_{k-(i-1)}(g'(q_s)) \in V(G^k) \Rightarrow g(q_s) \in V(G^k)$. In a similar way, $\forall q_s, q_t \in V(Q^o), \overline{q_s q_t} \in E(Q^o) \Rightarrow \overline{g'(q_s)g'(q_t)} \in E(G^k) \Rightarrow \overline{F_{k-(i-1)}(g'(q_s))F_{k-(i-1)}(g'(q_t))} \in E(G^k) \Rightarrow \overline{g(q_s)g(q_t)} \in E(G^k)$. According to Definition 3, $\forall q_s \in V(Q^o) \Rightarrow L(q_s) \subseteq L(g'(q_s)) = L(F_{k-(i-1)}(g'(q_s))) = L(g(q_s))$. According to Definition 2, $M \in R(Q^o, G^k)$, represented as $\langle g(q_1), \cdots, g(q_n)\rangle$. □

**B. Additional Experimental Results** Figure 23 shows the time cost in generating $G^k$ on UK-2002. Figure 24 shows the number of noise edges in $G^k$ on UK-2002. Figure 25 and Figure 26 show the query time in the cloud on UK-2002. Figure 27 shows the filtering time in the client side on UK-2002. As these figures show, the experiment results on UK-2002 are similar with those on other datasets. Figure 28, Figure 29 and Figure 30 show the query time in the cloud on the three datasets, respectively. Figure 31 shows the time cost of our star matching algorithm on the three datasets. Figure 32 shows the size of the set $RS$ generated by our star matching algorithm on the three datasets. Figure 33 shows the network overhead on the three datasets when transferring the candidate matching results from the cloud to the client side. Figure 34 shows the end-to-end running time on the three datasets. These results show similar trends as that in the experiment section and further confirm the superiority of our method.

[4] $F_j(M)$ is a mapping graph of $M$ under an automorphic function $F_j$, which is defined in Definition 4.

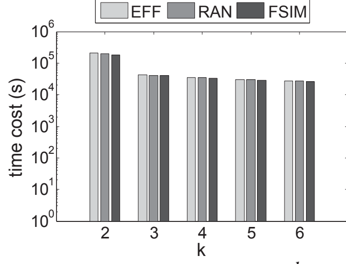



**Figure 23: Time cost in generating $G^k$ on UK-2002.**

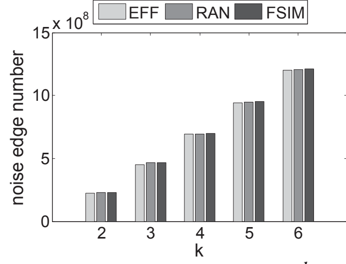



**Figure 24: Number of noise edges in $G^k$ on UK-2002.**

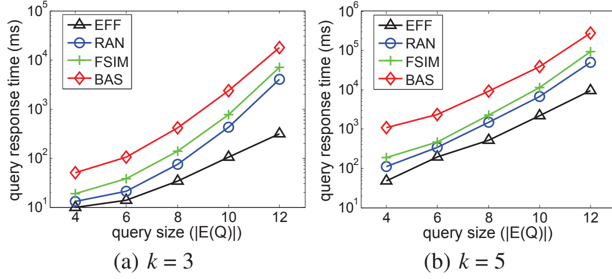



**Figure 25: Query time vs. $|E(Q)|$ on UK-2002.**

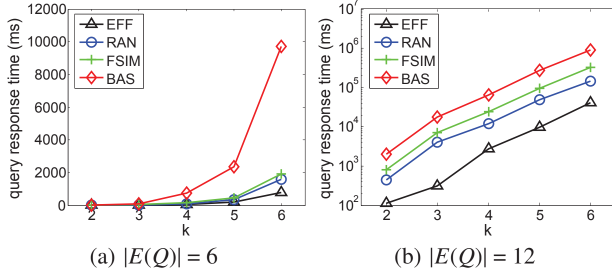



**Figure 26: Query time vs. $k$ on UK-2002.**

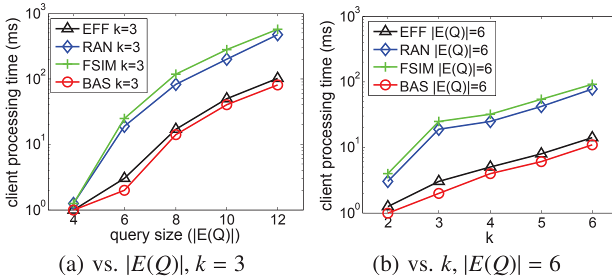



**Figure 27: Client processing time on UK-2002.**

| Dataset | | k = 2 | | k = 3 | | k = 4 | | k = 5 | | k = 6 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 |
| Web-Notre Dame | EFF | 3 | 10 | 9 | 31 | 22 | 70 | 42 | 137 | 77 | 254 |
| | RAN | 6 | 21 | 19 | 69 | 49 | 171 | 89 | 288 | 172 | 539 |
| | FSIM | 9 | 28 | 25 | 87 | 63 | 206 | 117 | 374 | 233 | 741 |
| DBpedia | EFF | 2 | 5 | 7 | 19 | 15 | 37 | 24 | 54 | 38 | 83 |
| | RAN | 3 | 10 | 11 | 36 | 23 | 66 | 41 | 110 | 60 | 168 |
| | FSIM | 4 | 14 | 14 | 50 | 32 | 101 | 50 | 149 | 89 | 264 |
| UK-2002 | EFF | 2 | 6 | 7 | 24 | 19 | 68 | 34 | 125 | 57 | 192 |
| | RAN | 4 | 14 | 11 | 38 | 28 | 97 | 53 | 196 | 89 | 307 |
| | FSIM | 6 | 20 | 15 | 52 | 41 | 152 | 72 | 267 | 124 | 449 |

Star Matching Time (ms)

**Figure 31: Star matching time.**

| Dataset | | k = 2 | | k = 3 | | k = 4 | | k = 5 | | k = 6 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 |
| Web-Notre Dame | EFF | 53 | 131 | 96 | 188 | 210 | 416 | 487 | 696 | 955 | 1592 |
| | RAN | 98 | 225 | 187 | 556 | 344 | 1383 | 792 | 2089 | 1411 | 4286 |
| | FSIM | 119 | 262 | 252 | 729 | 727 | 1580 | 1100 | 2858 | 2007 | 4790 |
| DBpedia | EFF | 38 | 66 | 64 | 126 | 158 | 283 | 296 | 431 | 728 | 1215 |
| | RAN | 54 | 177 | 125 | 402 | 255 | 1032 | 581 | 1920 | 1246 | 4505 |
| | FSIM | 87 | 231 | 163 | 478 | 342 | 1427 | 759 | 2544 | 1633 | 4992 |
| UK-2002 | EFF | 41 | 102 | 91 | 160 | 193 | 382 | 454 | 663 | 1007 | 1684 |
| | RAN | 63 | 186 | 127 | 431 | 331 | 787 | 638 | 1779 | 1390 | 3904 |
| | FSIM | 89 | 233 | 210 | 528 | 399 | 1391 | 722 | 2668 | 1634 | 4823 |

|RS|

**Figure 32: The size of result set generated by star matching algorithm ($|RS|$).**

| Dataset | | k = 2 | | k = 3 | | k = 4 | | k = 5 | | k = 6 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 |
| Web-Notre Dame | EFF | 1 | 14 | 2 | 34 | 3 | 61 | 5 | 108 | 9 | 184 |
| | RAN | 1 | 57 | 9 | 151 | 17 | 259 | 25 | 420 | 45 | 653 |
| | FSIM | 2 | 75 | 14 | 190 | 23 | 353 | 39 | 604 | 65 | 977 |
| | BAS | 1 | 27 | 6 | 100 | 12 | 242 | 23 | 522 | 50 | 1098 |
| DBpedia | EFF | 1 | 2 | 1 | 4 | 1 | 8 | 2 | 17 | 3 | 35 |
| | RAN | 1 | 2 | 1 | 5 | 2 | 11 | 3 | 26 | 4 | 69 |
| | FSIM | 1 | 3 | 2 | 6 | 2 | 14 | 3 | 33 | 5 | 81 |
| | BAS | 1 | 3 | 3 | 11 | 4 | 31 | 9 | 84 | 17 | 202 |
| UK-2002 | EFF | 1 | 22 | 2 | 53 | 3 | 85 | 4 | 121 | 7 | 170 |
| | RAN | 2 | 100 | 10 | 235 | 13 | 342 | 21 | 501 | 39 | 759 |
| | FSIM | 2 | 123 | 13 | 286 | 16 | 447 | 27 | 688 | 46 | 1021 |
| | BAS | 1 | 44 | 7 | 155 | 11 | 336 | 19 | 598 | 39 | 1010 |

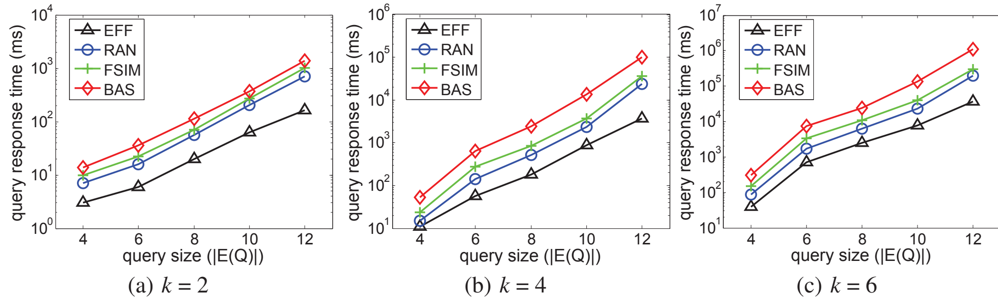
Network Transmission Time (ms)

**Figure 33: Network overhead.**

| Dataset | | k = 2 | | k = 3 | | k = 4 | | k = 5 | | k = 6 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 | \|E(Q)\| = 6 | \|E(Q)\| = 12 |
| Web-Notre Dame | EFF | 8 | 207 | 21 | 608 | 65 | 3871 | 249 | 10948 | 748 | 37561 |
| | RAN | 19 | 873 | 73 | 7941 | 192 | 24162 | 624 | 61306 | 1841 | 201831 |
| | FSIM | 27 | 1245 | 110 | 12329 | 347 | 36818 | 1197 | 108880 | 3519 | 300001 |
| | BAS | 38 | 1428 | 109 | 25240 | 664 | 101584 | 2423 | 318390 | 7472 | 1109777 |
| DBpedia | EFF | 5 | 54 | 14 | 170 | 45 | 1196 | 139 | 4144 | 671 | 19863 |
| | RAN | 7 | 402 | 24 | 3263 | 73 | 16823 | 265 | 52212 | 1115 | 232458 |
| | FSIM | 12 | 538 | 38 | 6048 | 118 | 29022 | 492 | 86412 | 1908 | 340659 |
| | BAS | 16 | 805 | 64 | 11725 | 516 | 97434 | 1805 | 350867 | 7158 | 1279688 |
| UK-2002 | EFF | 5 | 177 | 19 | 465 | 68 | 3052 | 203 | 10076 | 800 | 41116 |
| | RAN | 12 | 749 | 50 | 4721 | 140 | 13055 | 400 | 50651 | 1704 | 149985 |
| | FSIM | 17 | 1176 | 76 | 7956 | 199 | 25316 | 550 | 96353 | 2065 | 324241 |
| | BAS | 22 | 2095 | 112 | 17859 | 764 | 64515 | 2381 | 277208 | 9763 | 909963 |

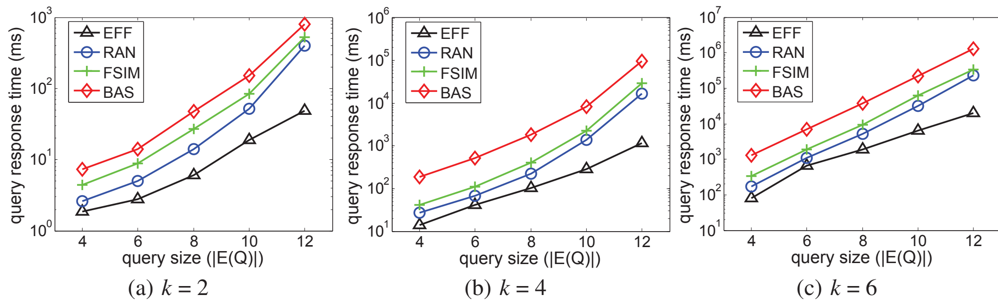
Overall Running Time (ms)

**Figure 34: Overall running time.**



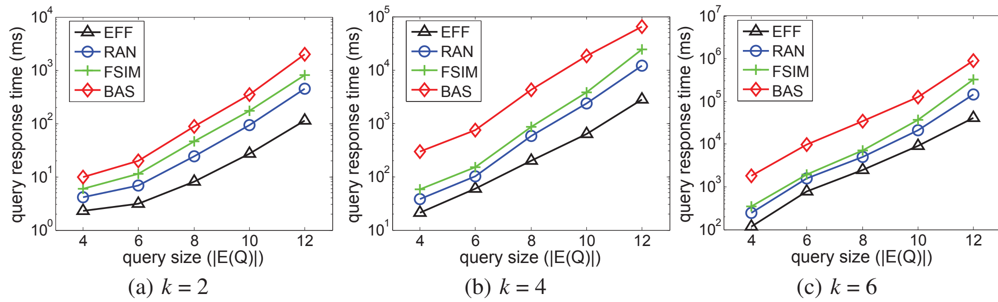

(a) $k = 2$ (b) $k = 4$ (c) $k = 6$

**Figure 28: Query time vs. $|E(Q)|$ on Web-NotreDame.**




(a) $k = 2$ (b) $k = 4$ (c) $k = 6$

**Figure 29: Query time vs. $|E(Q)|$ on DBpedia.**




(a) $k = 2$ (b) $k = 4$ (c) $k = 6$

**Figure 30: Query time vs. $|E(Q)|$ on UK-2002.**