# Asymmetric Structure-Preserving Subgraph Queries for Large Graphs

Zhe Fan[1] Byron Choi[1] Jianliang Xu[1] Sourav S Bhowmick[2]
[1]*Department of Computer Science, Hong Kong Baptist University, Hong Kong, China*
{zfan, bchoi, xujl}@comp.hkbu.edu.hk
[2]*School of Computer Engineering, Nanyang Technological University, Singapore*
assourav@ntu.edu.sg

***Abstract*—One fundamental type of query for graph databases is subgraph isomorphism queries (a.k.a *subgraph queries*). Due to the computational hardness of subgraph queries coupled with the cost of managing massive graph data, outsourcing the query computation to a third-party service provider has been an economical and scalable approach. However, confidentiality is known to be an important attribute of Quality of Service (QoS) in Query as a Service (QaaS). In this paper, we propose the first practical private approach for subgraph query services, *asymmetric structure-preserving subgraph query processing*, where the data graph is publicly known and the query structure/topology is kept secret. Unlike other previous methods for subgraph queries, this paper proposes a series of novel optimizations that only exploit graph structures, *not the queries*. Further, we propose a robust query encoding and adopt the novel cyclic group based encryption so that query processing is transformed into a series of private matrix operations. Our experiments confirm that our techniques are efficient and the optimizations are effective.**

## I. Introduction

Subgraph queries (via subgraph isomorphism) are a fundamental and powerful query in various real graph applications [25]. In particular, it is fundamental to various modern graph queries, such as graph pattern queries [2] and ontology-based matching [29]. While it is well known that subgraph queries are NP-hard, there has been significant research progress on enhancing their performance, *e.g.*, [8], [14], [26], [30]. A recent attempt has been to outsource costly computation to a *query service provider* ($\mathcal{SP}$), who is often equipped with powerful machines, to provide *query as a service* (QaaS). Thereby, users not only obtain high performance, scalability, and elasticity [13] but also are free from the burdens of managing IT infrastructure.

Because $\mathcal{SP}$s may not always be trusted, users' privacy may be threatened. In fact, (data or query) confidentiality has been recognized as one of the public's most crucial concerns (*e.g.*, [23]). A stream of research on private query processing has bloomed in the past decade, *e.g.*, in the context of relational databases [16], spatial databases [17] and graph databases [4]. However, to date, subgraph queries that preserve the query structure (a.k.a topology) over large networks has not yet been studied. We motivate this problem with the following scenario.

**Example 1.1:** Law enforcement agencies are increasingly using social media to solve crimes. According to a recent survey[1] of 1,221 federal, state and local law enforcement who use social media, four out of five officials used social media to solve crimes. Suppose a law enforcement agency is investigating a set of suspicious individuals over a public social network (*e.g.*, Cloob, which is a Persian-language social networking website, mainly popular in Iran) held in a third party $\mathcal{SP}$. In order to monitor the online activities of these individuals with one another, the agency wishes to glean information related to interactions between them on the network by issuing a subgraph query representing the relationships between the individuals. Unfortunately, it is possible that the $\mathcal{SP}$ may have been infiltrated by friends or sympathisers of these individuals. Hence, in order to protect the privacy of the intent of the agency from the $\mathcal{SP}$, the agency cannot expose the subgraph query directly, especially the query structure (*i.e.*, specific relationship pattern between the individuals). How can the agency glean relevant information using a subgraph query while preserving its topological privacy?

Unfortunately, previous work on privacy-preserving graph queries [3], [4], [12], [15], [19], [21], [24] (except [9]) cannot support subgraph queries while preserving their structure. Fan et al. [9] keep *both* query and data graphs private. In contrast, as query clients may often have data access privileges, the privacy requirement of this work is on queries only. As a result, querying significantly *larger* graphs becomes possible. Other work has studied privacy-preserving graph publication [5], [6], [22], [32], [33]. Since the published data are sanitized (*i.e.*, modified), it is not clear how subgraph queries can be supported. Recent studies [10], [11] have addressed the authenticity of the query answers, but not their confidentiality.

It is worth highlighting that the *intrinsic* technical challenge of this research direction is that although the data graph is available to the $\mathcal{SP}$, the $\mathcal{SP}$ cannot optimize the queries by directly exploiting the structure of the query graphs. In comparison, recent subgraph isomorphism algorithms (*e.g.*, VF2 [8], $\text{Turbo}_{\text{iso}}$ [14] and QuickSI [26]) intensively *utilize* the query graphs for optimization, which by definition, leaks their structural information. More recently, the work reported in [31] supports "structureless" graph queries . However, the structure is automatically generated by a ranking model and the $\mathcal{SP}$ is aware of the queries.

[1] www.lexisnexis.com/en-us/about-us/media/press-release.page?id=1342623085481181.

One may also attempt to solve the problem with a naive solution in which the $\mathcal{SP}$ exhaustively traverses all of the data graph to enumerate all *candidate mappings* (*i.e.*, possible mappings) between the query and the graph and return them to the client for verification. The intuition is that since the query structure is not exploited, its privacy is preserved. However, this is infeasible because the number of candidate mappings is exponential to the graph size in the worst case.

The first challenge of this research is then "*how to reduce a large data graph and subsequently the number of candidate mappings for verification, without exposing the query structure?*". Our first idea is to determine the minimized *candidate subgraphs* that contain at least a candidate mapping. Then candidate mappings are enumerated from those subgraphs instead of the original graph. In particular, we propose optimizations that use novel *neighborhood containment* of data vertices to minimize the subgraphs. Second, we determine subgraphs (called *candidate matchings*) from a candidate subgraph, where candidate mappings are enumerated. In comparison, in previous work [8], [14], [26] where privacy is not a concern, the matching (*i.e.*, the query graph) is known. We propose a *subgraph cache* and use *neighborhood equivalent classes* to further minimize the number of matchings and mappings.

The second challenge is "*how to verify if a candidate mapping is a subgraph isomorphism mapping without leaking the query structure?*" We propose a query encoding scheme and adopt an encryption scheme for query graphs. With these, we derive a basic *structure-preserving verification method* that consists of a series of private matrix operations. Moreover, to minimize communication overheads, we propose to use the complement of the encoding for an *enhanced verification method* for queries of bounded sizes.

In summary, the contributions of this paper are as follows:

- At query time, we first propose a new candidate subgraph exploration in the *absence of query structure*, to reduce a large data graph for query processing. We propose further reducing the size of candidate subgraphs by using neighborhood containment.
- Since candidate matchings are determined from candidate subgraphs, we propose a subgraph cache to prune the candidate matchings that are enumerated.
- We propose a robust encoding scheme and its verification method. We propose a model for the client to determine a proper encoding for his/her query.
- We conduct extensive experiments with real datasets to investigate the effectiveness and efficiency of our proposed methods.

**Organization.** Sec. II introduces the problem. We provide the preliminaries in Sec. III. Sec. IV presents query preprocessing at clients. Sec. V details the structure-preserving optimizations that minimize the candidate subgraphs and matchings. Sec. VI presents the verification of subgraph isomorphism mapping in an encrypted domain. We analyze privacy in Sec. VII. Sec. VIII shows the experimental results and Sec. IX compares the related work in the literature. We conclude in Sec. X.

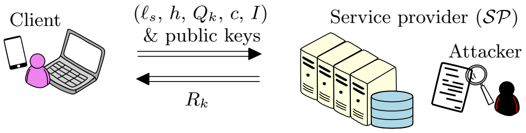

Client
($\ell_s$, $h$, $Q_k$, $c$, $I$)
& public keys
$R_k$
Service provider ($\mathcal{SP}$)
Attacker

Fig. 1. Overview of the system model.

## II. Problem Formulation

This section formulates the technical problem. More specifically, it presents the system model, attack model, privacy target, and problem statement.

**System model.** The system model resembles the classical server-client model, which contains two parties (illustrated in Fig. 1): (1) A *Service Provider* ($\mathcal{SP}$) and (2) the *query client* (or simply *client*). The $\mathcal{SP}$ is equipped with powerful computing utilities such as a cloud and hosts a subgraph query service for publicly known graph data $G$. The client encrypts his/her query $Q$ using a secret key (generated by himself/herself) $Q_k$ and submits $Q_k$ to the $\mathcal{SP}$. The $\mathcal{SP}$ then processes the client's encrypted query over the data $G$, and returns an encrypted result to the client. The client decrypts the result to obtain the query answer.

**Attack model.** We assume the *semi-honest (adversary) model* which is widely used in the database literature [3], [4], [17], [20], where the attackers are *honest-but-curious* (that is, the $\mathcal{SP}$ performs computations according to the system model but the $\mathcal{SP}$ may be interested in inferring secrets). Furthermore, the $\mathcal{SP}$ may be the attacker. For presentation simplicity, *we often call the attacker the $\mathcal{SP}$*. We assume that the attackers can be both *eavesdropping* and adopting the *chosen plaintext attack* (CPA) [20].

**Privacy target**. To facilitate technical discussions, we assume that the privacy target is to protect the *structures* of the query graph $Q$ from the $\mathcal{SP}$ under the attack model defined above. The *structural information* of $Q$ that we consider is the adjacency matrices of $Q$ (*i.e.*, the edge information of $Q$). It is obvious that the complete structure of a query can be derived from the edge information.

To sum up, the *problem statement* of this paper can be stated as follows: *Given the above system and attack model, we seek an efficient approach to complete the subgraph query service while preserving the privacy target.*

## III. Preliminaries and Overview

In this section, we first provide preliminary concepts related to subgraph queries. Then, we present an overview of our proposed solution.

### *A. Subgraph Queries*

The graph $G = (V, E, \Sigma, L)$ considered in this paper is an *undirected labeled connected graph*, where $V(G)$, $E(G)$, $\Sigma(G)$ and $L$ are the set of vertices, edges, vertex labels and the function that maps a vertex to its label, respectively. We use $\mathsf{nb}(v, G)$ to denote the set of neighbors of $v$ in $G$. We use $\mathsf{occ}(\ell, G)$ to represent the number of occurrences of the label $\ell$ in $V(G)$. We use $\mathbf{M_G}$ to represent the adjacency matrix of $G$. $\mathbf{M_G}(v_i, v_j)$ is a *binary* value, where $\mathbf{M_G}(v_i, v_j) = 1$ if

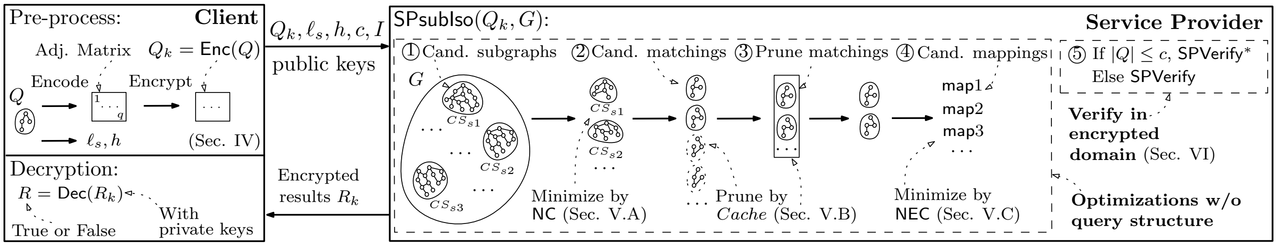


Fig. 2. Overview of our approach.

$(v_i, v_j) \in E(G)$, and otherwise 0. The adjacency matrix $\mathbf{M_G}$ represents the edge information. For the clarity of technical details, we present our technique with graphs having vertex labels only. The techniques we propose can be extended to support graphs with edge labels with minor modifications.

**Subgraph queries.** Def. 3.1 recalls the definition of subgraph isomorphism. We say a graph $G$ is a subgraph of another graph $G'$ *iff* there exists a subgraph isomorphism mapping (or *mapping* for short) from $G$ to $G'$, denoted as $G \subseteq G'$ or $\mathsf{subIso}(G,\ G') = \mathtt{true}$. In this paper, we study *subgraph queries* stated as: given a query graph $Q$ and a data graph $G$, the *subgraph query* is to determine if $\mathsf{subIso}(Q,\ G) = \mathtt{true}$. It is well known that deciding whether $Q$ is the subgraph of $G$ is NP-hard.

**Definition 3.1:** Given two graphs $G = (V, E, \Sigma, L)$ and $G' = (V', E', \Sigma', L')$, a *subgraph isomorphism mapping* from $G$ to $G'$ is an injective function $f : V(G) \to V(G')$ such that

- $\forall u \in V(G), f(u) \in V(G'), L(u) = L'(f(u))$; and
- $\forall (u, v) \in E(G),\ (f(u), f(v)) \in E(G')$.

### B. Overview of Our Approach

An overview of our solution is sketched in Fig. 2. Our solution essentially consists of the algorithms at the client side and those at the $\mathcal{SP}$ side.

**Client-side algorithms.** For the algorithms at the client side, we propose performing lightweight optimization and encryption on the query graph $Q$. (1) We first analyze the query to determine the *starting label* $\ell_s$ and the *minimum height* $h$ of $Q$, which are useful for minimizing the number and the sizes of candidate subgraphs of $G$. A *candidate subgraph* is a subgraph in $G$ that may contain a candidate matching, whereas a *candidate matching* is a subgraph of the candidate subgraph that may generate a *candidate mapping* between $Q$ and $G$. (2) We then propose a robust encoding scheme for $Q$ (of any size). (3) We adopt the *private-key encryption scheme* CGBE [9] to encrypt the encoded $Q$ to encrypted query $Q_k$, which is issued to the $\mathcal{SP}$ for query processing. (4) The client decrypts the encrypted answer returned by the $\mathcal{SP}$.

**Server-side algorithms.** The main ideas of the algorithms at the $\mathcal{SP}$ side are to localize and minimize the enumeration of candidate mappings between $Q$ and $G$ in candidate subgraphs. (1) The $\mathcal{SP}$ first efficiently determines the candidate subgraphs $CS_s$s (subgraphs) starting from each starting vertex $s$ of the label $\ell_s$ with the traversal depth $h$. We propose *neighborhood containment* (NC) to minimize each $CS_s$ in *the absence of the structure of* $Q$. Subsequently, it minimizes the number of candidate matchings to be enumerated by the $\mathcal{SP}$. (2) In each $CS_s$, the $\mathcal{SP}$ enumerates all candidate matchings ($CM_s$) and candidate mappings. We propose a *canonical labeling-based subgraph cache* and apply *neighborhood equivalent class* (NEC) to further avoid redundant $CM_s$ and candidate mappings, respectively. (3) We derive *structure-preserving verification* SPVerify from [9], where *multiple* encrypted messages $R_k$ (with *negligible false positives*) are returned to the client for decryption of the result.

SPVerify is derived from the seminal subgraph isomorphism algorithm: the Ullmann's algorithm [28]. The major benefit of Ullmann's algorithm is that its computation flow is simple; hence, we can cast the algorithm into a series of matrix operations (additions and multiplications). Since the encryption of SPVerify supports such matrix operations, privacy is preserved.

We also note that SPVerify may send multiple messages to the client for decryption, which may result in high decryption and network communication costs. Thus we propose SPVerify$^*$. The major difference between SPVerify$^*$ and SPVerify is that SPVerify$^*$ uses different query encodings according to different query sizes and significantly fewer encrypted messages are returned for decryption, and the query size is smaller than a system-related constant.

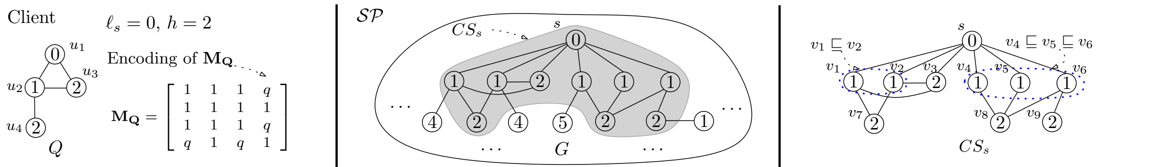
## IV. Query Preprocessing at the Client

In this section, we introduce a preprocessing method of the query graph. It comprises three steps: (1) retrieving optimization parameters; (2) encoding the query; and (3) encrypting the encoded query. The encrypted query is sent to the $\mathcal{SP}$.

### A. Retrieving Parameters for Optimization

In order to minimize (1) the size of each candidate subgraph $CS_s$ and (2) the total number of $CS_s$s, the $\mathcal{SP}$ requires the *minimum height* $h$ of $Q$ and, in the meantime, the starting label $\ell_s$ of $CS_s$s that is *infrequent* in $G$. These parameters ($h$ and $\ell_s$) are efficiently retrieved by the client.

Given a starting label $\ell_s$, the $\mathcal{SP}$ generates $CS_s$s by a breadth first search bounded by the depth $h$ starting at each vertex of $G$ having the label $\ell_s$ (to be detailed in Sec. V-A). On the one hand, to minimize the size of each $CS_s$, we simply find the *spanning tree* of $Q$ with a *minimum height* $h$ rooted from a vertex $u$, where $u \in V(Q)$ and $\ell_s = L(u)$. Intuitively, the smaller the value $h$, the smaller the size of each $CS_s$. Note that we cannot choose the vertex $u$ with $h = 1$ since it trivially leaks the structure of $Q$ (to be analyzed in Sec. VII). When there is a tie (*i.e.*, when vertices $u$ and $v$ of $Q$ have the same $h$) the client selects the vertex of the label that is *less frequent*


Client
$\ell_s = 0,\, h = 2$
Encoding of $\mathbf{M_Q}$
$u_1$
$u_2$
$u_3$
$u_4$
0
1
2
2
$Q$
$\mathbf{M_Q} = \begin{bmatrix} 1 & 1 & 1 & q \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & q \\ q & 1 & q & 1 \end{bmatrix}$
$\mathcal{SP}$
$CS_s$
$s$
$G$
$v_1 \sqsubseteq v_2$
$v_4 \sqsubseteq v_5 \sqsubseteq v_6$
$v_1$
$v_2$
$v_3$
$v_4$
$v_5$
$v_6$
$v_7$
$v_8$
$v_9$
$CS_s$

(a) Encoding of query graph

(b) Construction of candidate subgraphs

(c) Minimization of $CS_s$ by NC

Fig. 3. (a) Illustration of the preprocessing at the client; (b) Construction of candidate subgraphs; and (c) Minimization of $CS_s$ by NC.

in $G$, simply because the number of $CS_s$s is bounded by the occurrence of the label in $G$.

**Example 4.1:** Fig. 3 (a) shows an example of the selection of the starting label of query $Q$. The heights of the spanning trees rooted from $u_1$, $u_3$, and $u_4$ are 2. $u_1$ is finally chosen as the starting label as $\mathsf{occ}(0, G) < \mathsf{occ}(2, G)$, where $L(u_1) = 0$, and $L(u_3) = L(u_4) = 2$. $u_2$ is not considered because the height of its spanning tree is 1.

### B. Query Encoding

For presentation brevity, we present an *encoding* scheme for the query $Q$ (in Definition 4.1) to facilitate the discussion of the subsequent encryption scheme. This encoding is extended for further optimization (to be proposed in SubSec. VI-B).

**Definition 4.1:** The *encoding* of the entries of $\mathbf{M_Q}$ are: $\forall u_i, u_j \in V(Q)$,

$$\begin{cases} \mathbf{M_Q}(u_i, u_j) = q & \text{if } \mathbf{M_Q}(u_i, u_j) = 0; and \\ \mathbf{M_Q}(u_i, u_j) = 1 & \text{otherwise}, \end{cases}$$

where $q$ is a large prime number.

**Example 4.2:** Fig. 3 (a) also shows an example of the encoding of $Q$ by Def. 4.1. The entries in $\mathbf{M_Q}$ with values 0 are replaced by the large prime $q$.

### C. Query Encryption

Based on the encoding of $Q$, we adopt our recent private-key encryption scheme [9] (*cyclic graph based encryption scheme* CGBE) to encrypt the encoding of $Q$ ($\mathbf{M_Q}$). CGBE not only allows for efficient encryption and decryption but also supports both partial additions and multiplications, which is the core of efficient structure-preserving computation.

**Background on cyclic group.** Prior to the presentation of the definition of CGBE, we first recall the preliminaries of cyclic group [20]. Let $\mathbb{G}$ be a group. $p = |\mathbb{G}|$ is denoted as the *order* of $\mathbb{G}$. In particular, $\forall g \in \mathbb{G}$, the order of $\mathbb{G}$ is the smallest positive integer $p$ *s.t.*, $g^p = 1$. Let $\langle g \rangle = \{g^i : i \in \mathbb{Z}_p, g^i \in \mathbb{Z}_n\} = \{g^0, g^1, \cdots, g^{p-1}\}$ denote the set of elements generated by $g$. The group $\mathbb{G}$ is called *cyclic* if there exists an element $g \in \mathbb{G}$ such that $\langle g \rangle = \mathbb{G}$. $g$ is called a *generator* of $\mathbb{G}$.

**CGBE scheme.** The cyclic group based encryption scheme is defined as follows.

**Definition 4.2:** [9] The *cyclic group based encryption* scheme is a *private-key* encryption scheme, denoted as CGBE = (Gen, Enc, Dec), where

- Gen is a *key generation function*, which generates a secret key $x$ uniformly at random, a cyclic group $\langle g \rangle = \{g^i : i \in \mathbb{Z}_p, g^i \in \mathbb{Z}_n\}$. It outputs the private keys as $(x,\ g)$ and the value $p$ which is known to the public.
- Enc is an *encryption function*, which takes as input a message $m$ and the secret key $(x, g)$. It chooses a random value $r$, and outputs the ciphertext
$$e = mrg^x \pmod{p}$$
- Dec is a *decryption function*, which takes as input a ciphertext $e$, and the secret key $(x, g)$. It outputs
$$mr = eg^{-x} \pmod{p}$$

Note that the decryption function Dec in CGBE only decrypts the ciphertext $e$ as a product of the message $m$ and the random value $r$.

**Query encryption.** With CGBE, we define the encryption of $\mathbf{M_Q}$ as follows.

**Definition 4.3:** The *encryption* of $Q$ is denoted as $Q_k$, $Q_k = \{V, \mathbf{M_Q}_k, \Sigma, L\}$, where $\forall u_i, u_j \in V(Q)$,

$$\mathbf{M_Q}_k(u_i, u_j) = \mathsf{Enc}(\mathbf{M_Q}(u_i, u_j), x, g)$$

**Example 4.3:** For example, $\forall u_i, u_j$, if $\mathbf{M_Q}(u_i, u_j) = 1$, then $\mathbf{M_Q}_k(u_i, u_j) = \mathsf{Enc}(1) = rg^x \pmod{p}$; and if $\mathbf{M_Q}(u_i, u_j) = q$, then $\mathbf{M_Q}_k(u_i, u_j) = \mathsf{Enc}(q) = rqg^x \pmod{p}$.

**Discussion.** We remark that the client holds the secret keys $(x, g)$ for decryption and moreover, determines the constant $c$ and an encrypted value $I$ for encrypting verification results (to be discussed in Sec. VI). At last, $\ell_s$, $h$, $Q_k$, $c$, $I$ and $p$ are sent to the $\mathcal{SP}$ for structure-preserving query processing.

## V. Minimized SP Mapping Generation

The query preprocessing at the client side (in Sec. IV) generates ($\ell_s$, $h$, $Q_k$, $c$, $I$, $p$) for the $\mathcal{SP}$. Upon receiving these, the $\mathcal{SP}$ performs *structure-preserving* subIso (termed SPsubIso), presented in Algo. 1.

As outlined in Sec. I, the $\mathcal{SP}$ first minimizes the number of candidate mappings to-be-verified. For brevity, we focus on the most crucial procedures: candidate subgraph generation (Sec. V-A), candidate matching generation (Sec. V-B), and candidate mapping enumeration (Sec. V-C).

### A. Candidate Subgraph Generation

To avoid enumerating mappings on a possibly large graph, the $\mathcal{SP}$ first generates candidate subgraphs (Fig. 3(b)), where possible mappings can only be embedded in those subgraphs. A candidate subgraph is formally described in Def. 5.1.

**Definition 5.1:** A *candidate subgraph* started from $s \in V(G)$, denoted as $CS_s$, is an induced subgraph of $G$, *s.t.*

1) $L(s) = \ell_s$;
2) $\forall v \in V(CS_s)$, $v$ is reachable from $s$ within $h$ hops;
3) $\forall \ell$, $\ell \in \Sigma(CS_s) \Leftrightarrow \ell \in \Sigma(Q)$; and
4) $\forall \ell \in \Sigma(CS_s)$, $\mathsf{occ}(\ell, CS_s) \geq \mathsf{occ}(\ell, Q)$.

**Example 5.1:** Suppose $L(s) = \ell_s = 0$ and $h = 2$. Fig. 3(b) sketches an example of a candidate subgraph $CS_s$ (the grey-colored shadow) rooted from $s$ of $G$. For each vertex $v$ in

$CS_s$, $v$ is reachable from $s$ within 2 hops. The set of labels of $Q$ is the same as that of $CS_s$ (*i.e.*, $\Sigma(CS_s) = \Sigma(Q)$). For each label $\ell$ in $CS_s$, $\mathsf{occ}(\ell, CS_s) \geq \mathsf{occ}(\ell, Q)$.

**Initial generation.** GenCandSubGraph (Procedure 1.1, Lines 8-17) shows the generation of candidate subgraphs. Algo. 1 first initializes the $CS_s$ as $\emptyset$ (Line 1). Then, for each vertex $s \in V(G)$, where $L(s) = \ell_s$, it invokes GenCandSubGraph (Line 1). GenCandSubGraph simply generates $CS_s$ by a simple breadth first search method started from $s$ on $G$ within $h$ hops (Lines 10-15). $V_{CS_s}$ is to record the vertices of $CS_s$ determined so far. For each vertex $v \in V_{CS_s}$, $v$ must be reachable from $s$ within $h$ hops (Lines 13-15), and $L(v) \in \Sigma(Q)$ (Line 13). If $\forall \ell \in \Sigma(CS_s)$, $\mathsf{occ}(\ell, CS_s) \geq \mathsf{occ}(\ell, Q)$ (Line 16), $CS_s$ is set to the induced subgraph of $V_{CS_s}$ in $G$ (Line 17).

**Minimization by neighborhood information.** Since the sizes of candidate subgraphs have a significant impact on performance, we propose MinCandSubGraph (Procedure 1.2) to minimize the size of each $CS_s$. MinCandSubGraph is derived based on our notion of *neighborhood containment class* (NC) of $CS_s$, defined as follows.

**Definition 5.2:** Given $N = \{v_1, v_2, \cdots, v_n\}$ of $V(CS_s)$, $N$ is a *neighborhood containment class* (NC), denoted as $v_1 \sqsubseteq v_2 \sqsubseteq \cdots \sqsubseteq v_n$, *iff* $\forall v_i, v_j \in N$, $i < j$,

1) $L(v_i) = L(v_j)$;
2) a) $\mathsf{nb}(v_i, CS_s) \subseteq \mathsf{nb}(v_j, CS_s)$, if $N$ is an independent set in $CS_s$; or
   b) $\mathsf{nb}(v_i, CS_s) \cup \{v_i\} \subseteq \mathsf{nb}(v_j, CS_s) \cup \{v_j\}$, if $N$ is a clique of $CS_s$.

Based on Def. 5.2, the vertices of a candidate subgraph $CS_s$ exhibit a total ordering with respect to the $\sqsubseteq$ relationships. We have the following lemma for minimizing the size of a candidate subgraph by keeping the "top" vertices in the subgraph. The intuition is that the reduced $CS_s$ preserves all the structures of the original $CS_s$. The proof is established via a simple contradiction.

**Lemma 5.1:** Denote an NC $N$ as $\{v_1, v_2, \cdots, v_n\}$, where $N \subseteq V(CS_s)$ of a graph $G$. Denote the *reduced* $V(CS_s)$ (denoted as $CS_s^r$) is the *induced subgraph* of $V(CS_s) \backslash (N \backslash N_k)$ of $CS_s$, *i.e.* $N_k = \{v_{n-k+1}, v_{n-k+2} \cdots, v_n\}$ contains top-$k$ vertices of $N$ that are kept, where $k = \mathsf{occ}(L(v_1), Q)$. Then, the answer of $Q$ on $CS_s$ is the same as that on $CS_s^r$.

*Proof:* (Sketch) We prove that the removed structures in $CS_s$ are preserved in $CS_s^r$. We only consider $N$ is an independent set for simplicity as the argument when $N$ is a clique is similar.

We denote $N_{\bar{k}} = N \backslash N_k$ which represents the removed vertices in $N$. Any possible structure in $CS_s$ must be formed by some vertices from $N_{\bar{k}}$ and $N_k$. The former can be formulated as $N_{\bar{k}}^x \subseteq N_{\bar{k}}$, for some $x$ ($x \geq 1$), whereas the latter can be $N_k^y \subseteq N_k$, where $y$ ($y = k - x$). Denote $N_k' = N_{\bar{k}}^x \cup N_k^y$. Denote $CS_s^k$ as the induced subgraph of $V(CS_s) \backslash (N \backslash N_k)$ and $CS_s^{k'}$ and that of $V(CS_s) \backslash (N \backslash N_k')$, respectively. For any substructures in the induced subgraph of $CS_s^{k'}$ and for each vertex $v_x \in N_{\bar{k}}^x$, we can always find a distinguished vertex $v_y \in N_k \backslash N_k^y$ such that $\mathsf{nb}(v_x, CS_s) \subseteq \mathsf{nb}(v_y, CS_s)$, by Def. 5.2. This implies that $CS_s^{k'} \subseteq CS_s^k$. Therefore, $CS_s^r$ preserves all the structures in $CS_s$ by only keep $N_k$. ■

**Example 5.2:** Reconsider Example 5.1. $\{v_1, v_2\}$ is an NC as $L(v_1) = L(v_2)$, $\mathsf{nb}(v_1, CS_s) \subseteq \mathsf{nb}(v_2, CS_s)$ and $\{v_1, v_2\}$ forms an independent set of $CS_s$ in Fig. 3(c). Since $\mathsf{occ}(1, Q)$ = 1, by Lemma 5.1, we keep the top-1 vertex. It can be seen that the answer of $Q$ remains the same after removing either $v_1$ or $v_2$ from $CS_s$. For another example, let's consider the NC $\{v_4, v_5, v_6\}$ in Fig. 3(c), as the neighborhood of $v_4$ is contained by that of $v_5$. Hence, $v_4 \sqsubseteq v_5$. Similarly, $v_5 \sqsubseteq v_6$. $\{v_4, v_5, v_6\}$ forms an independent set. Again, by Lemma 5.1, we keep only the top-1 vertex, *i.e.*, $v_6$. The answer of $Q$ remains the same after removing $v_4$ and $v_5$. All in all, Fig. 4(a) shows $CS_s$, the candidate subgraph after the minimization.

**The minimization procedure** MinCandSubGraph**.** Procedure 1.2 shows the minimization of $CS_s$ by NC. For each $\ell \in \Sigma(CS_s)$, a set $\mathcal{N}$ of NC is first initialized as $\{\}$ (Line 18). For each vertex $v$ of $CS_s$ with the label $\ell$, sorted in ascending order of $|\mathsf{nb}(v, CS_s)|$ (Line 19) for efficiency, MinCandSubGraph checks if there is an $N$ in $\mathcal{N}$, such that $N \cup \{v\}$ forms an NC by Def. 5.2 (Line 20). If so, $v$ is then inserted into $N$ (Line 21). Otherwise, the algorithm creates a new $N = \{v\}$ and unions $N$ to $\mathcal{N}$ (Line 22). After the generation of NC of $CS_s$ for the label $\ell$, $CS_s$ can be minimized by Lemma 5.1 via keeping the top-$k$ vertices in each $N$, $N \in \mathcal{N}$, $k = \mathsf{occ}(\ell, Q_k)$ (Lines 23-24).

**Complexity.** The complexity of the generation of NC in Procedure 1.2 is $\mathcal{O}(d_{max}|V(CS_s)|^2)$, where $d_{max}$ is the maximum degree of the vertices in $CS_s$. In practice, $|V(CS_s)|$ is often in the order of hundreds, which is small.

### B. Candidate Matching Generation

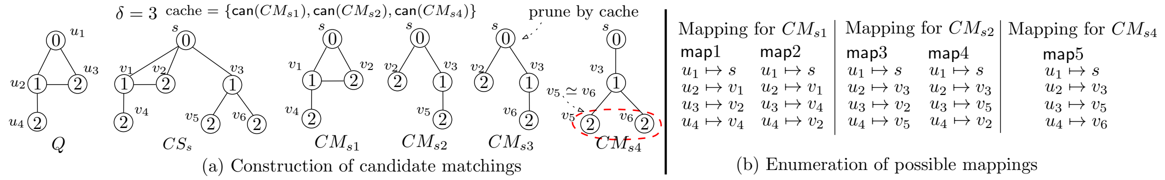
A unique challenge in structure-preserving query processing is that, in the absence of query structure, the $\mathcal{SP}$ matches $Q_k$ to *multiple* possible subgraph structures in $CS_s$. We call such subgraph structures *candidate matchings*. In contrast, if the query structures were not kept secret, the candidate matching was known to be $Q$. Fig. 4(a) shows four candidate matchings, $CM_{s1}$, $CM_{s2}$, $CM_{s3}$, and $CM_{s4}$. For each matching, candidate mappings are enumerated. It is evident that a naive enumeration of all candidate matchings can be inefficient. In this subsection, we propose GenCandMatch to efficiently generate candidate matchings. The main idea is to avoid generating redundant matchings from $CS_s$.

**Definition 5.3:** A candidate matching, denoted as $CM_s$, is a *connected induced subgraph* of $CS_s$, *s.t.*

1) $|V(CM_s)| = |V(Q)|$; and
2) $\forall \ell \in \Sigma(CS_s)$, $\mathsf{occ}(\ell, CM_s) = \mathsf{occ}(\ell, Q)$.

**Example 5.3:** Fig. 4 (a) lists all the $CM_s$s enumerated from $CS_s$. $\forall CM_{si}$, $i \in \{1, ..., 4\}$, $|V(CM_{si}| = |V(Q)|$, and $\forall \ell \in \Sigma(CS_s)$, $\mathsf{occ}(\ell, CM_{si}) = \mathsf{occ}(\ell, Q)$.

**Elimination of redundant $CM_s$.** We make two observations from Example 5.3 and Fig. 4. (1) $CM_{s2}$ is *graph-isomorphic* to $CM_{s3}$. If candidate mappings are generated from $CM_{s2}$, it



(a) Construction of candidate matchings

(b) Enumeration of possible mappings

Fig. 4. (a) Construction of candidate matchings; and (b) Enumeration of possible mappings.

**Algorithm 1** SPsubIso ($Q_k$, $G$, $\ell_s$, $h$)

```
Input: The encrypted query graph $Q_k$, data graph $G$, starting label $\ell_s$ and hop $h$
Output: The encrypted result $R_k$
1: Initialize $CS_s = CM_s = \emptyset$, $Cache = \emptyset$, and $R_k = 1$
2: for each vertex $s \in V(G)$ with the starting label $\ell_s$
3:     GenCandSubGraph($Q_k$, $G$, $s$, $h$, $CS_s$)          /* By Def. 5.1 */
4:     MinCandSubGraph($Q_k$, $CS_s$)                          /* Minimize $CS_s$ */
5:     Initialize set $V_{CM_s} = \{s\}$
6:     GenCandMatch($V_{CM_s}$, $Q_k$, $CS_s$, $R_k$, $Cache$) /* By Def. 5.3 */
7: Return $R_k$
Procedure 1.1 GenCandSubGraph ($Q_k$, $G$, $s$, $h$, $CS_s$)
8: Initialize a queue $Visit$ and a set $V_{CS_s}$ as empty
9: $Visit$.push($s$),  $V_{CS_s}$.insert($s$),  $s$.hop() $= 0$
10: while $Visit$ is not empty                    /* BFS method */
11:     $v = Visit$.pop()
12:     if ($v$.hop() $= h$) continue              /* By 2. in Def. 5.1 */
      /* By 3. in Def. 5.1 */
13:     for each $v' = \mathsf{nb}(v, G)$, $v' \notin V_{CS_s} \wedge L(v') \in \Sigma(Q_k)$
14:         $Visit$.push($v'$),  $V_{CS_s}$.insert($v'$)
15:         $v'$.hop() = $v$.hop() + 1
   /* By 4. in Def. 5.1 */
16: while $\exists \ell \in \Sigma(V_{CS_s})$, s.t. $\mathsf{occ}(\ell, V_{CS_s}) < \mathsf{occ}(\ell, Q_k)$
        remove all $v$ from $\Sigma(V_{CS_s})$, where $v \in \Sigma(V_{CS_s})$ and $\Sigma(v) = \ell$
17: $CS_s$ = GenInducedSub($G$, $V_{CS_s}$)
Procedure 1.2 MinCandSubGraph ($Q_k$, $CS_s$)
18: for each $\ell \in \Sigma(CS_s)$, $\mathcal{N} = \{\}$          /* $\mathcal{N}$ is a set of NC */
      /* Ascending ordered by $|\mathsf{nb}(v, CS_s)|$ */
19:     for each $v \in V(CS_s)$, $L(v) = \ell$,
20:         if $\exists N \in \mathcal{N}$, s.t.,          /* By Def. 5.2 */
          (1) $\{v\} \cup N$ forms an independent set (or a clique); and
          (2) $\mathsf{nb}(v, CS_s)$ (or $\mathsf{nb}(v, CS_s) \cup \{v\}$) contains those of vertices in $N$.
21:             $N$.insert($v$)              /* Ordered by $\sqsubseteq$ */
22:         else create a new $N$, $N = \{v\}$, $\mathcal{N} = \mathcal{N} \cup \{N\}$
23:     for each $N \in \mathcal{N}$, $N_k = \{v_{n-k+1}, \cdots, v_n\}$, $k = \mathsf{occ}(\ell, Q_k)$
24:         Remove $N \backslash N_k$ from $CS_s$          /* By Lemma. 5.1 */
```

**Algorithm 2** GenCandMatch ($V_{CM_s}$, $Q_k$, $CS_s$, $R_k$, $Cache$)

```
1: if $V_{CM_s}$ are enumerated before                 /* $V_{CM_s}$ is checked */
2:     return
3: if $|V_{CM_s}| = |V(Q_k)|$
4:     $CM_s$ = GenInducedSub($CS_s$, $V_{CM_s}$)
5:     if $Cache$.isHit(can($CM_s$))          /* $CM_s$ is checked before */
6:         return
7:     $Cache$.insert(can($CM_s$)) /* Insert can($CM_s$) into $Cache$ */
8:     if $V(Q) \leq \delta$          /* Insert subgraphs of $CM_s$ into $Cache$ */
9:         $Cache$.insert(can($CM'$)),
           where $CM' \subseteq CM_s$, $|V(CM')| = |V(CM_s)|$
10:     GenAllMap($Q_k$, $CM_s$. $R_k$)     /* Generate candidate mappings */
11: for each $v \in V_{CM_s}$
12:     for each $v' \in \mathsf{nb}(v, CS_s)$ in descending order, $v' \notin V_{CM_s}$
13:         if $\mathsf{occ}(\ell, V_{CM_s}) < \mathsf{occ}(\ell, Q_k)$, $\ell = L(v')$     /* By Def. 5.3 */
14:             $V_{CM_s}$.insert($v'$)
15:             GenCandMatch ($V_{CM_s}$, $Q_k$, $CS_s$, $R_k$)
16:             $V_{CM_s}$.remove($v'$)
```

is obvious that generating mappings from $CM_{s3}$ is *redundant*. (2) $CM_{s1}$ is a supergraph of $CM_{s2}$. One can simply generate mappings from $CM_{s1}$, and skip $CM_{s2}$ and $CM_{s3}$.

To remove the redundancies mentioned above, it is exactly to solve the following problem: "*given a graph $G$ and a graph database $\mathcal{G}$ : $\{G_1, ...\}$, how to efficiently determine if $G$ is a subgraph of $G'$, $G' \in \mathcal{G}$?*" Such a problem has been extensively studied before (*e.g.*, [26], [30], [34]). Existing solutions involve an index computed offline. In our context, candidate matchings are enumerated *online*. Hence, the existing solutions cannot be directly applied.

**Canonical labeling-based subgraph cache.** Let's recall a crucial property of canonical labeling. In the context of graph query processing, the canonical labeling of a graph $G$ is denoted as $\mathsf{can}(G)$, and $\mathsf{can}(G) = \mathsf{can}(G')$ *if and only if* $G$ is isomorphic to $G'$. While the cost for computing the canonical labeling of a graph is not yet known (P or NP), the cost for comparing whether two graphs are isomorphic using the canonical labeling is $O(1)$, once computed. This work adopts the state-of-the-art labeling called *minimum dfs code* [30] from the literature.

For each query, we propose $Cache$ to store $\mathsf{can}(CM_s)$, where each $CM_s$ is the checked candidate matching. Once a new $CM'_s$ is generated, we first check if $\mathsf{can}(CM'_s)$ is already in $Cache$. If so, $CM'_s$ is discarded. Otherwise, we insert $\mathsf{can}(CM'_s)$ into $Cache$. Further, we continue to enumerate subgraphs $CM'$s from $CM'_s$, where for each $CM'$, $|V(CM')| = |V(CM'_s)|$, $CM' \subseteq CM'_s$, and $\mathsf{can}(CM')$ is stored in $Cache$. Putting subgraphs of $CM'_s$ increases the chance of pruning by $Cache$. However, the *trade-off* is that as the query size increases, the computational cost for enumerating all subgraphs of a $CM'_s$ increases exponentially. Thereby, for practical purposes, we enumerate all of the subgraphs $CM'$s of $CM'_s$ only if $|V(Q)| \leq \delta$, where $\delta$ is a user-defined threshold.

**Example 5.4:** The top of Fig. 4 (a) shows the idea of the canonical labeling-based cache. We assume that $\delta$ is 3, and the sequence of the generation of $CM_s$ is from $CM_{s1}$ to $CM_{s4}$. $CM_{s3}$ is eliminated as $\mathsf{can}(CM_{s2})$ is in $Cache$. If we set $\delta$ to 5, then $CM_{s2}$ and $CM_{s3}$ are both eliminated, because $CM_{s2}$ is a subgraph of $CM_{s1}$, and when $CM_{s1}$ is processed, $\mathsf{can}(CM_{s2})$ is inserted into $Cache$.

**The ordering in $CM_s$ generation.** From Example 5.4, it can be observed that the ordering in $CM_s$ generation affects the performance of the cache, when $|V(Q)| \leq \delta$. Suppose $\delta = 5$. Assume $CM_{s2}$ is generated before $CM_{s1}$. Then, $CM_{s2}$ is not eliminated. In general, the earlier the larger $CM_s$s are generated, the better the performance is. Therefore, we find a simple ordering for $CM_s$ generation, by greedily adding vertices to the $CM_s$ by the degree of each vertex.

**$CM_s$ generation.** In Algo. 1, $Cache$ is initialized to $\emptyset$ (Line 1), and the vertex set of each generated $CM_s$, denoted as $V_{CM_s}$, is initialized to $\{s\}$ (Line 5). In Line 6, the algorithm for the generation of $CM_s$s, denoted as GenCandMatch, presented in Algo. 2.

**Algorithm 3** GenAllMap ($Q_k$, $CM_s$, $R_k$)

```
Input: The encrypted query $Q_k$, candidate matching $CM_s$ and encrypted result $R_k$
1: Generate $\mathbf{M}$ from $Q_k$ and $CM_s$
2: Initialize vector used as $\vec{0}$
3: Initialize vector map as $\vec{0}$
4: Construct NEC of $CM_s$
5: EnumMap($u_0$, used, map, $\mathbf{M}$, $Q_k$, $CM_s$, $R_k$)    /* Enumeration */
Procedure 3.1 EnumMap($u_i$, used, map, $\mathbf{M}$, $Q_k$, $CM_s$, $R_k$)
6: if $i = |V(Q_k)|$
7:     if $|V(Q_k)| \leq c$, SPVerify*(map, $Q_k$, $CM_s$, $R_k$)    /* Sec. VI-B */
8:     else SPVerify(map, $Q_k$, $CM_s$, $R_k$)    /* Sec. VI-A */
9: for each $j < |V(CM_s)|$, $\mathbf{M}(u_i, v_j) = 1 \land$ used$[v_j] = 0$
       /* Eliminate redundant mappings by Lemma 5.2 */
10:     if $\exists v_{j'}$, $v_{j'} \simeq v_j$, $j' < j$, used$[v_{j'}] = 0$    /* Lexi. order */
11:         continue
12:     used$[v_j] = 1$, map$[u_i] = v_j$
13:     EnumMap($u_{i+1}$, used, map, $Q_k$, $CM_s$, $R_k$)
14:     used$[v_j]$ = map$[u_i] = 0$
```

In GenCandMatch (Algo. 2), $CM_s$ is generated from $CS_s$ (Lines 11-16) until the size of the matching is the same as the query (Lines 1-10). For each vertex $v \in V_{CM_s}$ (Line 12), it attempts to add $v$'s neighboring vertex $v' \in \mathsf{nb}(v, CS_s)$, where $v' \notin V_{CM_s}$, and it adds the $v'$ with a large degree first ($\mathsf{deg}(v', CS_s)$). If $\mathsf{occ}(\ell, V_{CM_s}) < \mathsf{occ}(\ell, Q_k)$ (Line 13), where $\ell = L(v')$, $v'$ is then added to $V_{CM_s}$ (Line 14). GenCandMatch is called recursively (Line 15) until $|V_{CM_s}| = |V(Q_k)|$ (Lines 1-10). $CM_s$ is an induced subgraph of $V_{CM_s}$ (Line 4).

For each generated $CM_s$, $\mathsf{can}(CM_s)$ is checked if $\mathsf{can}(CM_s)$ is already in $Cache$ (Line 5). If yes, there exists a $CM'$ in $Cache$ such that $CM'$ is isomorphic to $CM_s$. By the property of canonical labeling, $CM_s$ can be eliminated (Lines 5-6). Otherwise, we add $\mathsf{can}(CM_s)$ into $Cache$ (Line 7). If $|V(Q)| \leq \delta$, we enumerate the subgraphs $CM'$ of $CM_s$, where $|V(CM')| = |V(CM_s)|$, and insert $\mathsf{can}(CM')$ into $Cache$ (Lines 8-9). At last, GenAllMap (see Algo. 3) is invoked (Line 10) to generate all possible mappings (see SubSec. V-C) between $Q_k$ and $CM_s$.

### C. Candidate Mapping Generation

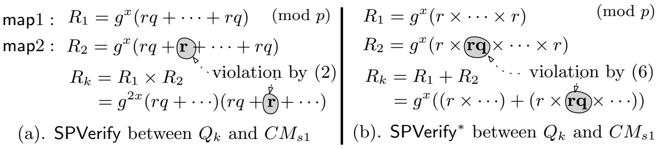
When a new candidate matching $CM_s$ is generated, Algo. 3 invokes GenAllMap to enumerate all possible mappings between $Q_k$ and $CM_s$.

**Elimination of redundant mappings by NEC.** Recall that the number of mappings is exponential to the size of $CM_s$. However, in practice, many mappings are redundant. Hence, before generating the mappings, we utilize *neighborhood equivalent classes* NECs of $CM_s$ (Def. 5.4) to eliminate those redundant mappings. We remark that NEC is a special case of NC. While a similar NEC has been proposed in [14] for query and data graphs, our NEC is applied to data graphs only.

**Definition 5.4:** Given an NC $N = \{v_1, v_2, \cdots, v_n\}$ of $CS_s$, where $N$ is either an independent set or a clique of $CS_s$, $N$ is a *neighborhood equivalent class* (NEC), denoted as $v_1 \simeq v_2 \simeq \cdots \simeq v_n$, *iff* $\forall v_i, v_j \in N$, $v_i \sqsubseteq v_j$ and $v_j \sqsubseteq v_i$.

**Example 5.5:** Let's consider the example of $CM_{s4}$ in Fig. 4 (a), $\{v_5, v_6\}$ is an NEC as $L(v_5) = L(v_6)$ and $\mathsf{nb}(v_6, CM_{s4}) = \mathsf{nb}(v_5, CM_{s4})$ = $\{v_3\}$.

Suppose that $u_3$ and $u_4$ (in Fig. 4 (a)) have been mapped to $v_5$ and $v_6$, respectively. It is not necessary to map $u_3$ and $u_4$ onto $v_6$ and $v_5$, respectively. This can be formalized as the following lemma. Foremost, we often use ($u_i \mapsto v_i$) to denote map$[u_i]$ = $v_i$ for ease of exposition.


map1 : $R_1 = g^x(rq + \cdots + rq)$ (mod $p$)
map2 : $R_2 = g^x(rq + \mathbf{r} + \cdots + rq)$
$R_k = R_1 \times R_2$
$= g^{2x}(rq + \cdots)(rq + \mathbf{r} + \cdots)$
violation by (2)
$R_1 = g^x(r \times \cdots \times r)$ (mod $p$)
$R_2 = g^x(r \times \mathbf{rq} \times \cdots \times r)$
$R_k = R_1 + R_2$
$= g^x((r \times \cdots) + (r \times \mathbf{rq} \times \cdots))$
violation by (6)

(a). SPVerify between $Q_k$ and $CM_{s1}$

(b). SPVerify* between $Q_k$ and $CM_{s1}$

Fig. 5. SPVerify (and SPVerify*) between $Q_k$ and $CM_{s1}$.

**Lemma 5.2:** Suppose the following are true:

1) $u_i, u_j \in V(Q)$, $v_{i'}, v_{j'} \in V(CM_s)$, $L(u_i) = L(u_j) = L(v_{i'}) = L(v_{j'})$;
2) $v_{i'} \simeq v_{j'}$;
3) ($u_i \mapsto v_{i'}$) and ($u_j \mapsto v_{j'}$).

Let map′ be the mapping map except that ($u_i \mapsto v_{j'}$) and ($u_j \mapsto v_{i'}$). Then, map is a candidate mapping between $Q$ and $CM_s$ *if and only if* map′ is also a candidate mapping.

The proof is omitted since it can be established by a simple proof by contradiction. Next, we present the data structures and the mapping generation, that exploit the lemma.

**Data structures.** (i) A *vertex label mapping* $\mathbf{M}$ is a $m \times n$ binary matrix, $m = |V(Q_k)|$ and $n = |V(CM_s)|$. Specifically, $\forall u, v$, $\mathbf{M}(u, v) = 1$ if $L(u) = L(v)$, where $u \in V(Q_k)$ and $v \in V(CM_s)$; and otherwise 0. (ii) A vector map of the size $|V(Q_k)|$ is to record a mapping from $Q_k$ to $CM_s$, map$[u] = v$ (*i.e.*, $u \mapsto v$) represents that vertex $u$ in $Q_k$ is mapped to vertex $v$ in $CM_s$. map$[u]$ = 0 if $u$ is not yet mapped. (iii) A vector used of the size $|V(CM_s)|$ is to denote whether the vertex $v$ in $CM_s$ has been mapped to a vertex of $Q_k$ and recorded in map. used$[v] = 0$ if $v$ is not yet mapped. In other words, used$[v] = 1$ if and only if map$[u] = v$ for some $u \in Q_k$.

**Algorithm for mapping generation.** The detailed algorithm GenAllMap is shown in Algo. 3. It first initializes the data structures, including $\mathbf{M}$, used and map in Lines 1-3. Line 4 constructs NEC of $CM_s$, which is similar to that of NC in Procedure 1.2. EnumMap (Lines 6-14) is then invoked to enumerate all possible mappings. A mapping map is constructed vertex by vertex iteratively. Line 9 checks if $v_j$ is a possible map of $u_i$ by $\mathbf{M}$ and used. We then exploit the equivalence class to further check if $v_j$ can be possibly mapped to $u_i$ (Lines 10-12). The vertices in a NEC are checked in a predefined order (*e.g*, lexicographical order). If $\exists v_{j'}$ *s.t.* $v_{j'} \simeq v_j$, $j' < j$ and $v_{j'}$ is not used before, then $v_j$ is skipped (Line 10). If $v_j$ passes the check, EnumMap is called recursively (Line 13) until a full mapping is constructed (Line 6).

**Example 5.6:** Fig. 4(b) illustrates the possible candidate mapping generation for those $CM_s$s of Example 5.3. Since $v_5 \simeq v_6$ in $CM_{s4}$, by Lemma 5.2, we only enumerate map$_5$, where $u_3 \mapsto v_5$ and $u_4 \mapsto v_6$, but the one with $u_3 \mapsto v_6$ and $u_4 \mapsto v_5$ is eliminated.

## VI. SP Mapping Verification

Section V presented a series of optimizations that reduce the number of mappings to be generated. Then, for each mapping map, the $\mathcal{SP}$ verifies (in the encrypted domain) if there is no violation in map. The encrypted verification results are aggregated before they are transmitted to the client. In this section, we derive a basic verification (SPVerify) from [9] for our problem setting. Next, we propose an enhanced one (SPVerify*) that aggregates many more messages but requires the query size to be smaller than a user-determined constant.

### A. SPVerify

Given a mapping map between $Q_k$ and $CM_s$, we determine if $CM_s$ is a valid mapping or not. Specifically, we define the violation *w.r.t.* the encoding of $Q$ as follows: $\exists u_i, u_j \in V(Q)$,

$$\mathbf{M_Q}(u_i, u_j) = 1 \wedge (v_{i'}, v_{j'}) \notin E(CM_s) \qquad (1)$$

where $v_{i'}, v_{j'} \in V(CM_s)$, $u_i \mapsto v_{i'}$ and $u_j \mapsto v_{j'}$. It states that there exists an edge between vertices $u_i$ and $u_j$ in $Q$, but there is no corresponding edge between the mapped vertices $v_{i'}$ and $v_{j'}$ in $CM_s$. We term the case in Formula 1 as a *violation of subgraph isomorphism* (or simply *violation*). A mapping without violation(s) is called a *valid* mapping.

**Example 6.1:** Let's take the two mappings map1 and map2 of $CM_{s1}$ in Fig. 4 (b) as an example. First, no violation is found in map1. Second, for map2, we find that $\mathbf{M_Q}(u_1, u_3) = 1$ and $(s, v_4) \notin E(CM_{s1})$, where map2$[u_1] = s$ and map2$[i_3] = v_4$. Therefore, map2 is invalid.

**Algorithm for SPVerify.** The intuitive idea of SPVerify is to transform the verification steps into mathematical operations on $\mathbf{M_Q}_k$ and $CM_s$, where (1) the violation (Formula 1) can be detected; (2) only matrix additions and multiplications are involved; and (3) the result can be aggregated with one message or multiple messages.

Algo. 4 shows the detailed algorithm. The inputs are a candidate mapping map, an encrypted query graph $Q_k$, a candidate matching $CM_s$ and an encrypted result $R_k$. We remark that $R_k$ is to record the aggregated result for $CM_s$s, where $R_k$ is initialized to 1 in Line 1 Algo. 1.

We initialize an intermediate result $R_i$ with a value 0 (Line 1). For each pair of vertices $(u_i, u_j)$ in $V(Q)$ and the mapped vertex pair $(v_{i'}, v_{j'})$ in $CM_s$ (Lines 2-3), the following two steps are performed:

1. Additions (Lines 4-7): if $(v_{i'}, v_{j'}) \notin E(CM_s)$, $R_i$ is set to $(\mathbf{M_Q}_k(u_i, u_j) + R_i)$ (mod $p$). This indicates that if $(u_i, u_j)$ is an edge in $Q$, $R_i$ must not contain a factor of $q$, and the decryption value of $R_i$ is non-zero (*i.e.*, the current mapping map contains a violation (by Formula 1), which is not a valid mapping). Otherwise, no violation is caused by $(u_i, u_j)$. This sets $R_i$ to the value $I + R_i$ (mod $p$), where $I$ is an encrypted value *with* a factor $q$ issued by the client, $I = \mathsf{Enc}(q)$; and
2. Multiplications (Line 8): it aggregates $R_i$ into $R_k$, by $R_k = R_k \times R_i$ (mod $p$). If there is at least one valid mapping from $Q$ to $G$, *i.e.*, at least one $R_i$ whose decryption value is zero. The decryption value of $R_k$ must also be zero. Otherwise, it is non-zero. We remark that CGBE leads to *errors* if the number of $R_i$s aggregated in $R_k$ is larger than a predetermined value $M$.

**Example 6.2:** Fig. 5(a) depicts an example of SPVerify between $Q_k$ and $CM_{s1}$. There are two mappings from $Q_k$ to $CM_{s1}$ in Fig. 4(b). In map1, *all* the factors in $R_1$ contain $q$ since map1 is a valid mapping. However, in map2, since there *exists* a violation between $(u_1, u_3)$ and $(s, v_4)$, there is a factor in $R_2$ that has no prime $q$. $R_k = R_1 \times R_2$ (mod $p$).

**Algorithm 4** SPVerify(map, $Q_k$, $CM_s$, $R_k$)

```
1: Initialize $R_i = 0$
2: for each $u_i, u_j \in V(Q)$, $i < j$
3:     $v_{i'} = \mathsf{map}[u_i], v_{j'} = \mathsf{map}[u_j]$
       /* Additions */
4:     if $(v_{i'}, v_{j'}) \notin E(CM_s)$
5:         $R_i \mathrel{+}= \mathbf{M_Q}_k(u_i, u_j)$ (mod $p$)        /* Aggregate violation */
6:     else
7:         $R_i \mathrel{+}= I$ (mod $p$)        /* No violation, $I = \mathsf{Enc}(q)$ */
   /* Multiplications */
8: $R_k \times= R_i$ (mod $p$)    /* Decompose $R_k$ after aggregating $M$ $R_i$ */
```

**Decryption at the client.** After receiving all the encrypted messages $R_k$, the client performs two main steps:

- For each $R_k$, the client computes the *plaintext* of $R_k$ by $R'_k = \mathsf{Dec}(R_k, x, g)^M$; and
- The client computes the final result by $R = R'_k$ (mod $q$).

$R$ equals zero if and only if there is at least one valid mapping from $Q$ to $G$ and thus, $\mathsf{subIso}(Q, G) = \mathtt{true}$.

**Example 6.3:** We show the decryption at the client by using the example in Fig. 5 (a). Assume $M = 2$. The encrypted message $R_k$ only aggregates two $R_i$s. The client generates the $g^{-2x}$, computes $R'_k = R_k \times g^{-2x}$ (mod $p$), and finally computes $R = R'_k$ (mod $q$). The result is zero, which indicates $Q$ is a subgraph of $G$.

**Decomposition scheme**. We recall that the decryption (Dec in Def. 4.2) uses the arithmetic modulo $p$. The message $m * r$ must not exceed $p$. When there are too many $R_i$s multiplied into $R_k$, the product (in the plaintext domain) may exceed $p$. Subsequently, the client will not obtain the correct plaintext under the arithmetic system. Therefore, we decompose the product into smaller numbers and the client decrypts those numbers instead. Through Formula 2 below, we can determine the maximum number of $R_i$s to be aggregated in $R_k$ ($M$):

$$\begin{array}{rrcl} & \mathsf{Len}(p) & \geq & M(\mathsf{Len}(q) + \mathsf{Len}(r)) \\ \Leftrightarrow & M & \leq & \frac{\mathsf{Len}(p)}{(\mathsf{Len}(q)+\mathsf{Len}(r))}, \end{array} \qquad (2)$$

where $\mathsf{Len}(p)$ is the size of $p$.

Let's say we set $M = 10$. From experiments, the number of mappings (after our minimizations) for our queries is around 500 on average. Each message is 2048 bits in size. Thus, the communication cost is around 12.8KB, which is very small.

**False positives**. Due to CGBE [9], the two matrix operations in SPVerify introduce negligible false positives: (1) additions with computing $R_i$ (Lines 4-7); and (2) multiplications with computing $R_k$ in each decomposed number (Line 8). However, the probabilities of the above two false positives are negligible.

The probability of false positives from the aggregation (additions) while computing $R_i$ and the multiplication of $R_k$s in each decomposed number are respectively stated in Props 6.1 and 6.2, which can be established by simple arithmetics.

**Proposition 6.1:** The probability of false positives in $R_i$ is $\frac{1}{q}$, which is negligible.

**Algorithm 5** SPVerify*(map, $Q_k$, $CM_s$, $R_k$)

```
1: Initialize $R_i = 1$
2: for each $u_i, u_j \in V(Q)$, $i < j$
3:     $v_{i'} = \mathsf{map}[u_i], v_{j'} = \mathsf{map}[u_j]$
       /* Multiplications */
4:     if $(v_{i'}, v_{j'}) \notin E(CM_s)$
5:         $R_i \times= \mathbf{M_Q}_k(u_i, u_j)$ (mod $p$)      /* Aggregate violation */
6:     else
7:         $R_i \times= I$ (mod $p$)                  /* No violation, $I = \mathsf{Enc}(1)$*/
   /* Additions */
8: $R_k += R_i$ (mod $p$)
```

*Proof:* The case of false positives is that each individual values in the addition are not divisible by $q$ but the sum equals a multiple of $q$. Such a probability is

$$\begin{aligned} \Pr(\text{false positives in } R_i) &= \Pr(r_1 + \cdots + r_{m(m-1)/2} = 0(\text{mod } q)) \\ &= \tfrac{1}{q} \end{aligned} \quad (3)$$

where $m = V(Q)$, and $q$ is a large prime number, *e.g.*, 32bits. Thus, the probability is negligible, in practice. ■

**Proposition 6.2:** The probability of false positives in $R_k$ is $1 - e^{-\frac{M}{q}}$, which is negligible, in each decomposed number.

*Proof:* The probability of false positives in each $R_k$ is

$$\begin{aligned} \Pr(\text{false positives in } R_k) &= 1 - \Pr(\text{true positive in all } R_i) \\ &= 1 - (1 - \tfrac{1}{q})^M \\ &\approx 1 - e^{-\frac{M}{q}} \end{aligned} \quad (4)$$

where $M$ is the size of the decomposed number. Since $M \ll q$, the probability is negligible in practice. ■

### B. Optimized SPVerify for Queries of Bounded Sizes

Each encrypted message $R_k$ sent by SPVerify aggregates at most $M$ messages $R_i$s. In this subsection, we propose SPVerify*, which significantly reduces the number of messages returned, which in turn reduces both the communication and computational costs at the client. The main idea behind SPVerify* is *to use multiplications to detect violations* since queries are often small and *use additions to aggregate* $R_i$*s*. Hence, the value of $R_k$ may not exceed $p$ even after many aggregations. However, a tradeoff of SPVerify* is that the query size must be bounded by a pre-determined constant $c$.

Similar to SPVerify, SPVerify* also detects the violation by multiplications and additions. In order to achieve that, we first define a *complement* encoding of the query (see Def. 6.1).

**Definition 6.1:** The *encoding* of the entries of $\mathbf{M_Q}$ are: $\forall u_i, u_j \in V(Q)$,

$$\begin{cases} \mathbf{M_Q}(u_i, u_j) = 1 & \text{if } \mathbf{M_Q}(u_i, u_j) = 0 \\ \mathbf{M_Q}(u_i, u_j) = q & \text{otherwise} \end{cases}$$

where $q$ is a large prime number.

In relation to Def. 6.1, we adopt Formula 1 to state the violation: $\forall u_i, u_j \in V(Q)$,

$$\mathbf{M_Q}(u_i, u_j) = q \wedge (v_{i'}, v_{j'}) \notin E(G) \quad (5)$$

where $v_{i'}, v_{j'} \in V(G)$, $u_i \mapsto v_{i'}$ and $u_j \mapsto v_{j'}$.

**Algorithm for** SPVerify***.** For ease of comparison, we present the pseudo-code of SPVerify* (shown in Algo. 5) in the style of SPVerify. The inputs and the initialized data structures are the same as SPVerify, except that $R_k$ must be initialized to $0$. We first initialize an intermediate result $R_i$ with a value 1 (Line 1). For each pair of vertex $(u_i, u_j)$ in $V(Q)$ and the mapped vertex pair $(v_{i'}, v_{j'})$ in $CM_s$ (Lines 2-3), the following two steps are performed:

1. Multiplications (Lines 4-7): according to the violation (by Formula. 5), if $(v_{i'}, v_{j'}) \notin E(CM_s)$, set $R_i$ as the value $\mathbf{M_Q}_k(u_i, u_j) \times R_i$ (mod $p$). This indicates that as soon as $(u_i, u_j)$ is an edge in $Q$, $R_i$ must contain the factor $q$, and the decryption value is zero (*i.e.*, the current mapping map contains a violation). Otherwise, $R_i$ is set to a value $I \times R_i$ (mod $p$), where $I$ is an encrypted value *without* factor $q$ issued by the client, $I = \mathsf{Enc}(1)$; and
2. Additions (Line 8): it aggregates $R_i$ to $R_k$, where $R_k = R_k + R_i$ (mod $p$). If there is at least one valid mapping from $Q$ to $G$ (*i.e.*, at least one $R_i$ whose plain text is non-zero). The decrypted value of $R_k$ must also be non zero. Otherwise, it is zero.

**Example 6.4:** Fig. 5 (b) illustrates an example of SPVerify*. Similarly, since there is no violation in map1, all the factors in $R_1$ do not contain $q$. Regarding map2, since there is a violation, $R_2$ contains a factor $q$. $R_k = R_1 + R_2$ (mod $p$).

**Decryption at the client.** The decryption is modified as:

- The client computes the message encoded in $R_k$ as $R'_k = \mathsf{Dec}(R_k, x, g)^{m(m-1)/2}$, where $m = |V(Q)|$; and
- The client computes the final result by $R = R'_k$ (mod $q$).

$R$ equals non-zero if and only if there is at least one valid mapping from $Q$ to $G$. Thus $\mathsf{subIso}(Q, G) = \mathtt{true}$.

**Example 6.5:** We show the decryption in Fig. 5 (b). For simplicity, we assume that $R_k$ only aggregates $R_1$ and $R_2$. The client generates $g^{-6x}$, computes $R'_k = R_k \times g^{-6x}$ (mod $p$), and finally computes $R = R'_k$ (mod $q$). The result is non-zero which indicates that $Q$ is a subgraph of $G$.

**Determining the constant $c$ to decide when to use** SPVerify **or** SPVerify***.** In SPVerify*, multiplications are used to aggregate violations by edges in $CM_s$ (Line 4 in Algo. 5), instead of aggregating numerous mapping results ($R_i$ in Line 8 of Algo. 4). Similarly, when $R_i$ (Lines 4-7) in Algo. 5 exceeds $p$, the client cannot recover the plaintext. The number of multiplications for each $R_i$ is directly related to the size of the query ($|V(Q)|$). We can determine the maximum size of the query, denoted as $c$, using the following inequality.

$$\begin{aligned} \mathsf{Len}(p) &\geq \tfrac{c(c-1)}{2}(\mathsf{Len}(q) + \mathsf{Len}(r)) \\ \Leftrightarrow \quad 0 &\geq c^2 - c - \tfrac{2\mathsf{Len}(p)}{\mathsf{Len}(q) + \mathsf{Len}(r)} \end{aligned} \quad (6)$$

Putting these together, in Lines 7-8 of Algo. 3, once $|V(Q)| \leq c$, the $\mathcal{SP}$ uses SPVerify*. Otherwise, it uses SPVerify.

**False positives.** Since both SPVerify and SPVerify* use CGBE, we can obtain that the probabilities of false positives of SPVerify* are also *negligible*. Their proofs are almost identical to those of Props. 6.1 and 6.2 and [9], and hence, omitted.

## VII. Privacy Analysis

In this section, we prove the privacy of the encryption method and SPsubIso. The attack model is the one defined in Sec. II. The attackers or $\mathcal{SP}$s are eavesdroppers and can adopt chosen plaintext attack (CPA) [20].

**Privacy of the encryption method.** CGBE is adopted to encrypt the query graph in this paper. The privacy of CGBE and $\mathbf{M}_{\mathbf{Q}_k}$ can be recalled from [9].

**Lemma 7.1:** [9] CGBE is secure against CPA. $\mathbf{M}_{\mathbf{Q}_k}$ is preserved from the $\mathcal{SP}$ against the attack model under CGBE.

Then, based on Lemma 7.1, we have the following.

**Proposition 7.1:** The structure of the query is preserved from the $\mathcal{SP}$ against the attack model under CGBE.

*Proof:* (Sketch) The proof can be derived from Lemma 7.1. After receiving $Q_k$, the $\mathcal{SP}$ cannot break the $\mathbf{M}_{\mathbf{Q}_k}$ since they are secure against CPA. $V$, $\Sigma$ and $L$ do not contain structural information. Thus, the structure of query is preserved from the $\mathcal{SP}$ against the attack model. ■

**Privacy of** SPsubIso**.** SPsubIso mainly consists of five steps: (1) GenCandSubGraph; (2) MinCandSubGraph; (3) GenCandMatch; (4) GenAllMap; and (5) SPVerify (or SPVerify$^*$). We now analyze the privacy of each step as follows. However, first, the analysis requries some notations. We denote a function $P(m, h, \Sigma)$ that returns all possible graphs of $m$ vertices with a minimum height $h$ and the labels $\Sigma$. $|P(m, h, \Sigma)|$ is exponential to the value $m$ and the size of $\Sigma$. [2] Let $\mathcal{A}(Q)$ is a function that returns 1 if $\mathcal{SP}$ is able to determine the exact structure of $Q$, and 0 otherwise. The probability that the $\mathcal{SP}$ can determine the structure of the query $Q$ is denoted as $Pr[\mathcal{A}(Q) = 1]$. Given a query $Q$ and $(m, h, \Sigma)$, the probability of determining its structure is $Pr[\mathcal{A}(Q) = 1] = 1/|P(m, h, \Sigma)|$.

**Proposition 7.2:** Under GenCandSubGraph, MinCandSubGraph, GenCandMatch and GenAllMap, $Pr[\mathcal{A}(Q) = 1] = 1/|P(m, h, \Sigma)|$.

*Proof:* (Sketch) The proof is established by one main fact: $\mathcal{SP}$ does not utilize any structural information of the query, except the value $h$ in the algorithm.

- GenCandSubGraph utilizes $\ell_s$, $h$, $Q_k$ and $G$ to generate *all* the $CS_s$s;
- MinCandSubGraph minimizes the size of each $CS_s$ by using only the structure of $CS_s$ itself;
- GenCandMatch utilizes $Q_k$ and $CS_s$ to generate $CM_s$s;
- GenAllMap enumerates *all* the possible mappings maps between $Q_k$ and $CM_s$.

The $\mathcal{SP}$ cannot learn the structure of $Q$ by invoking them, and thus the probability of determining a structure remains $Pr[\mathcal{A}(Q) = 1] = 1/|P(m, h, \Sigma)|$. ■

In SPVerify and SPVerify$^*$, $\mathcal{SP}$ sends messages to the clients. The clients may terminate the algorithm when a mapping is found, which may leak information to the $\mathcal{SP}$. Such a leak can be quantified in the following proposition.

**Proposition 7.3:** Under SPVerify or SPVerify$^*$, the following hold for :

- If $Q$ is a subgraph of $G$, $Pr[\mathcal{A}(Q) = 1] = 1/|S|$, where $S = \{G | G \in P(m, h, \Sigma)$, $G \subseteq CM_s$, where $CM_s \in Cache\}$; and
- If $Q$ is not a subgraph of $G$, $Pr[\mathcal{A}(Q) = 1] = 1/|P(m, h, \Sigma)|$.

*Proof:* (Sketch) Since the algorithm SPVerify$^*$ is similar to that of SPVerify, due to the space constraint, we prove it with SPVerify only. The proof involves two aspects:

*(1) $\mathcal{SP}$ can never determine any structural information from the mathematical computations in each steps of* SPVerify*:*

Recall that SPVerify comprises a fixed number of mathematical operations in the encrypted domain in Algo. 4.

- Lines 4-7 invoke a constant number of additions of $\mathbf{M}_{\mathbf{Q}_k}$ and $R_i$, and only structure of $CM_s$ is considered. More specifically, $\forall i, j$, $m^2$ additions are invoked for $\mathbf{M}_{\mathbf{Q}_k}(i, j)$ and $R_i$; and
- Line 8 requires one multiplication on each $R_i$ and $R_k$.

Based on Lemma 7.1, all the intermediate computations results are securely protected against the attack model. Moreover, each step of SPVerify has a constant number of operations in the encrypted domain. $\mathcal{SP}$ cannot learn any information from them.

*(2) $\mathcal{SP}$ may* only *infer some structural information from the message communications:*

Recall that once $M$ $R_i$s are aggregated into $R_k$, $R_k$ is returned to the client, the client may decide to terminate SPVerify after receiving $R_k$s. There are two cases:

- Suppose there is at least one valid $R_k$ such that $Q$ is a subgraph of $G$. In this case, $Q$ must be graph (or subgraph) isomorphic to one of $CM_s$ in $Cache$. Therefore, $Pr[\mathcal{A}(Q) = 1] = 1/|S|$, where $S = \{G | G \in P(m, h, \Sigma)$, $G \subseteq CM_s$, $CM_s \in Cache\}$; and
- If the client does not terminate the algorithm, $\mathcal{SP}$ does not know if there is a valid $R_k$ or not. Thus, the probability of determining the structure of $Q$ is still $Pr[\mathcal{A}(Q) = 1] = 1/|P(m, h, \Sigma)|$. ■

Based on Prop. 7.3, we note that the client can make a tradeoff between privacy and response times by terminating the algorithm as late as acceptable.

## VIII. Experimental Evaluation

In this section, we present a detailed experimental evaluation of our proposed techniques with popular real world datasets.[3] The results show that our techniques are efficient and our optimizations are effective.

### A. Experimental Setup

**The platform.** We conducted all our experiments on a machine with an Intel Core i7 3.4GHz CPU and 16GB memory running Windows 7 OS. All techniques were implemented on C++, and CGBE was implemented on the GMP library. We simulate the bandwidth as 10Mbits/s.

**Data and query sets**. We benchmarked real-world datasets: DBLP, Amazon, Youtube, and LiveJournal [1]. Since the vertices do not have labels, we adopt the approach that uses

[2] We remark that if $h = 1$, the $\mathcal{SP}$ is able to infer that the vertex with $\ell_s$ *must* connect to other vertices in $Q$. To avoid this special case, as mentioned in Sec. IV, we choose the starting vertex where $h$ equals or larger than 2.

[3] As discussed in Sec. I, previous studies are not applicable to our problem, since they heavily exploit query structures, which are secret in this work.

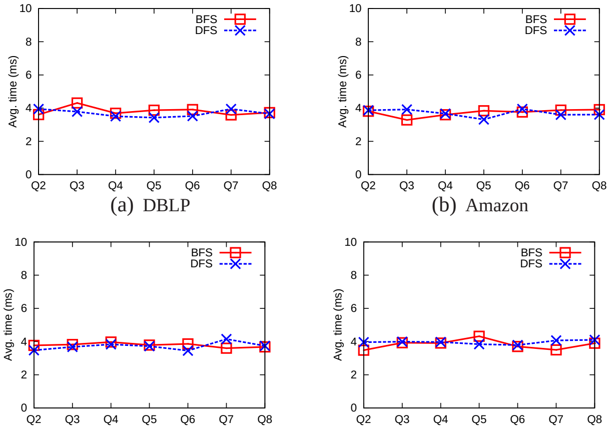
TABLE I
STATISTICS OF THE REAL-WORLD DATASETS

| Graph $G$ | $\lvert V(G)\rvert$ | $\lvert E(G)\rvert$ | Avg. Degree | $\lvert\Sigma(G)\rvert$ |
|---|---|---|---|---|
| DBLP | 317,080 | 1,049,866 | 6.62 | 199 |
| Amazon | 334,863 | 925,872 | 5.52 | 153 |
| Youtube | 1,134,890 | 2,987,624 | 5.26 | 978 |
| LiveJournal | 3,997,962 | 34,681,189 | 17.34 | 1355 |

the degree of the vertex as its label [18]. Some statistics of the datasets are shown in Table I.

For each dataset, we generated two types of queries [27]: (1) BFS queries (BFS) and (2) DFS queries (DFS) by random BFS and DFS methods, respectively. Both BFS and DFS contain query sets $Q2$-$Q8$, wherein each $Qn$ contains 1,000 query graphs, and $n$ is the number of vertices of each query of the query set. $h$ of the query sets are around 3-4 on average.

**Default values of the parameters.** In CGBE, the prime $p$ and $q$ are $2048$ bits and $32$ bits, respectively. The random number $r$ is 32 bits. The largest value $c$ is 12 by Formula 6. However, to study the performance of both SPVerify$^*$ and SPVerify, we first set $c$ to 6, by default. That is, if $|V(Q)| \leq 6$, we used SPVerify$^*$. Otherwise, we used SPVerify. We finally investigated the effectiveness of SPVerify$^*$ with $c = 11$. For SPVerify$^*$, we set $M = 100$ by default (*i.e.*, we aggregated $100$ $R_i$s into each $R_k$). For SPVerify, we set $M = 10$ only. Unless specified otherwise, $\delta = 5$. Under these settings, no false positives was detected from the entire experiments.



(a) DBLP (b) Amazon

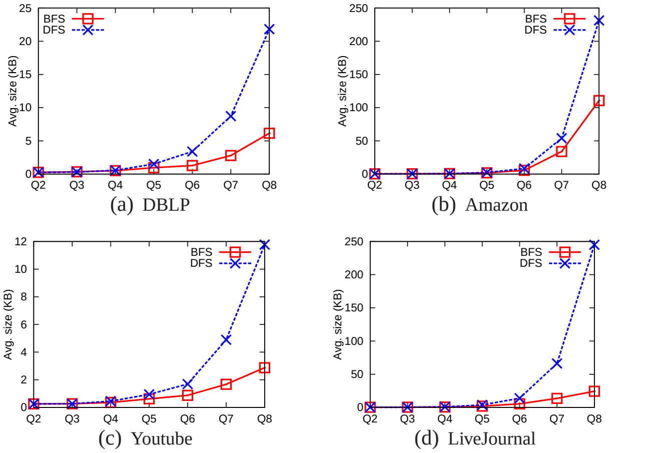
(c) Youtube (d) LiveJournal

Fig. 6. Average preprocessing time at the client.

## B. Performance at the Client Side

**Preprocessing time at the client.** We report the average preprocessing time of the query $Q$ at the client side on all datasets in Fig. 6. Specifically, the preprocessing of $Q$ includes (1) the computation for $\ell_s$ and $h$; and (2) the encryption of $Q$ by CGBE. We observe that the average times for each query on all datasets are around 4ms, which shows that the preprocessing is in cognitively negligible.

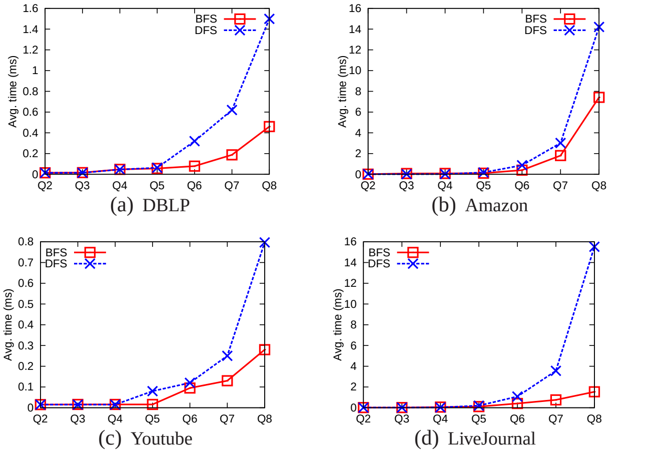
**The sizes of messages received by the client.** We report the sizes of the encrypted messages $R_k$s that the client received in Fig. 7. Due to the optimizations by SPsubIso, the largest sizes of $R_k$s (at $Q6$) are around 13KB on LiveJournal, which can be efficiently transmitted via today's networks. For $Q7$-$Q8$, as we set $c$ to 6 (by default), SPsubIso uses SPVerify. The number of $R_i$s aggregated in each $R_k$ is 10. Thus, the message sizes for $Q7$-$Q8$ are larger. Since the maximum value of $c$ is 11 in the current configuration, SPVerify$^*$ can be used to produce much smaller messages (to be discussed with Fig. 14).



(a) DBLP (b) Amazon

(c) Youtube (d) LiveJournal

Fig. 7. Average received encrypted message size at the client.



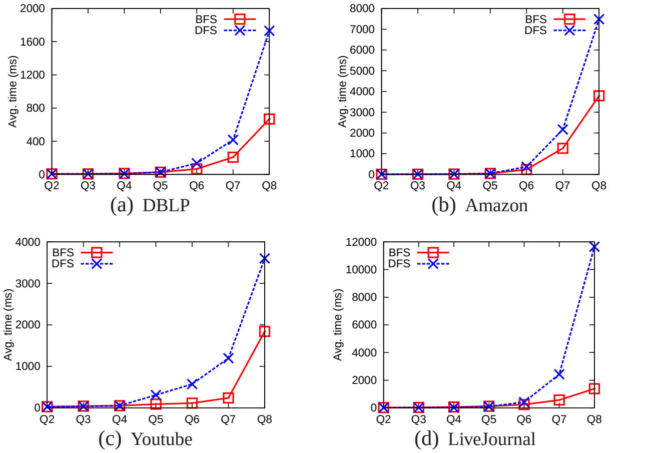
(a) DBLP (b) Amazon

(c) Youtube (d) LiveJournal

Fig. 8. Average decryption time at the client.

**The decryption time at the client.** After receiving the encrypted messages $R_k$s, the client decrypts $R_k$s. The decryption time is shown in Fig. 8. Since the sizes of $R_k$s are small and the decryption method is simple, the average decryption times at the client are correspondingly fast at most 16ms.

## C. Performance at the SP Side



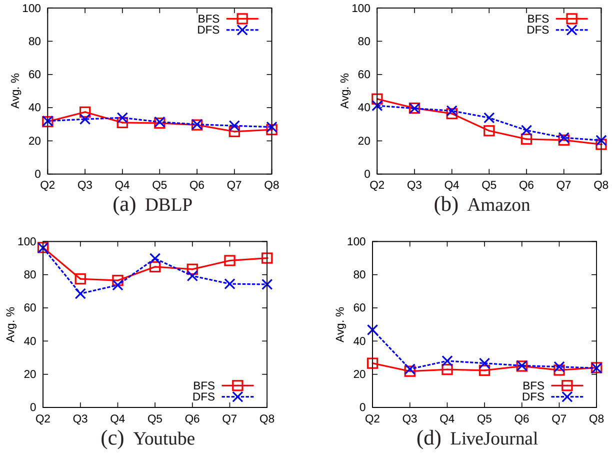
(a) DBLP (b) Amazon

(c) Youtube (d) LiveJournal

Fig. 9. Average total running time at the $\mathcal{SP}$.

**The total runtime at the $\mathcal{SP}$.** Fig. 9 shows the average total runtime at the $\mathcal{SP}$ on all datasets, which is exactly the runtime of SPsubIso. For the simplicity of performance analysis, we terminated SPsubIso once the client found at least one valid mapping. (The client may postpone the termination to achieve higher privacy [9], although that introduces small but non-trivial overhead to SPsubIso.) It is not surprising that the runtimes increase exponentially with the query sizes. For $Q8$, the largest runtime is around 12s on LiveJournal. However, the running times for small queries ($Q2$-$Q6$) are well below 600ms for all datasets.

We further report the breakdowns of the total runtimes of SPsubIso: (1) GenCandSubGraph and MinCandSubGraph; and (2) GenCandMatch and SPVerify. For the DBLP and Amazon datasets, the breakdown percentages of both query sets are similar: $30\%$ and $70\%$. For Youtube, they are $81\%$ and $19\%$, and for LiveJournal, they are $53\%$ and $47\%$.



(a) DBLP (b) Amazon

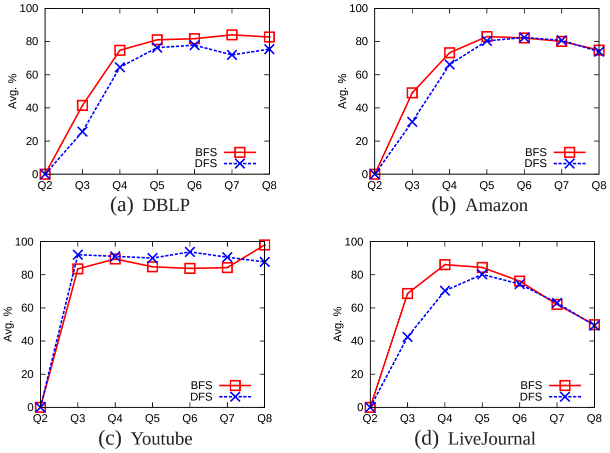
(c) Youtube (d) LiveJournal

Fig. 10. Average % of reduced vertices in $CS_s$ by NC.

**The effectiveness of minimization of $CS_s$.** In Fig. 10, we show the average percentage of the reduced vertices of $CS_s$ by NC in MinCandSubGraph. We observe that MinCandSubGraph reduces around 40% of the vertices of $CS_s$s on DBLP and Amazon, and at least 60% on Youtube. However, for LiveJournal, the percentage (on average) is around $20\%$.

In our experiment, we note that a small fraction of queries have $CS_s$s that contain numerous candidate mappings. The reason is that SPsubIso cannot exploit query structures for optimizations. In this case, for each $CS_s$, we compute an upper bound of the number of candidate mappings of a query by simple calculations on $CS_s$. For those candidate subgraphs that may exceed $100,000$ mappings, we transmit the candidate subgraphs to the client to do subIso (*e.g.*, using [14] or [8]). The percentage of such queries is very small, at most $1\%$ for $Q2$-$Q7$ on all datasets. For $Q8$, the percentage is only $10\%$. In other words, most subgraph queries are successfully outsourced to the $\mathcal{SP}$.

**The effectiveness of the elimination of redundant $CM_s$.** Fig. 11 shows the average percentage of redundant $CM_s$s pruned by $Cache$ in GenCandMatch. We note that as the query size increases, the effectiveness of $Cache$ increases. For $Q2$-$Q4$ of all datasets, the percentage of the elimination of redundant $CM_s$ increases from $0\%$ in $Q2$ to $80\%$ in $Q4$. For $Q5$-$Q8$ on DBLP, Amazon and Youtube, the percentages

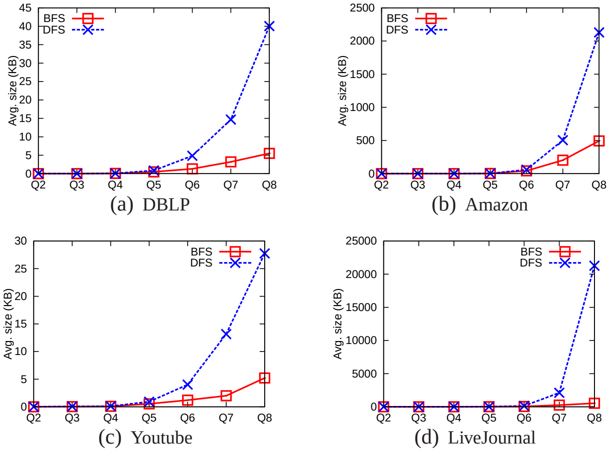


(a) DBLP (b) Amazon

(c) Youtube (d) LiveJournal

Fig. 11. Average % of the pruned redundant $CM_s$ by $Cache$.

are stable at around $80\%$. We note the graph structures of LiveJournal are diverse and there are many distinct $CM_s$s. The effectiveness of $Cache$ then decreases from $80\%$ to $50\%$ for $Q5$-$Q8$. This is also reflected by the fact that the sizes of the encrypted messages $R_k$s are the largest for LiveJournal (see Fig. 7).

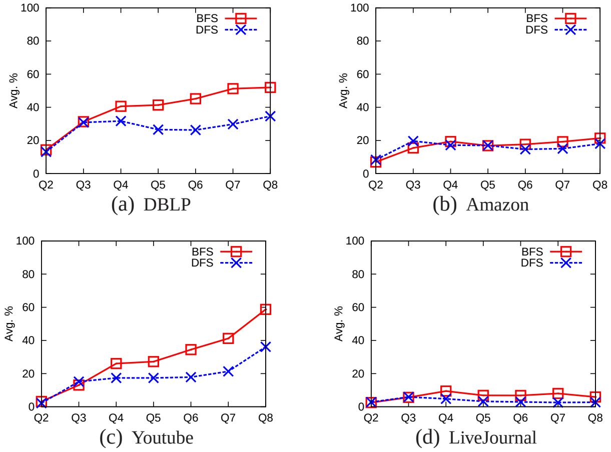


(a) DBLP (b) Amazon

(c) Youtube (d) LiveJournal

Fig. 12. Average $Cache$ size at $\mathcal{SP}$.

**The memory consumption of $Cache$.** We report the memory consumption of $Cache$ in Fig. 12. As we only store the hash code of the canonical labeling of each distinct $CM_s$, the memory consumption is very small (at most $25MB$).



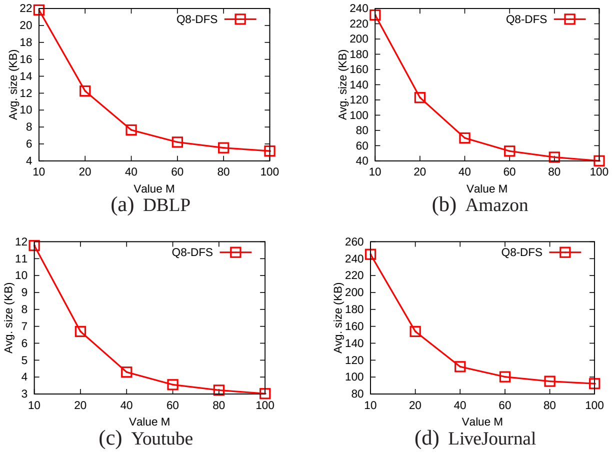
(a) DBLP (b) Amazon

(c) Youtube (d) LiveJournal

Fig. 13. Average % of the pruned redundant mappings by NEC.

**The effectiveness of pruning redundant mappings by NEC.**

We report the pruning of redundant mappings by using NEC in Fig. 13. We observe that, for most of the queries, we pruned approximately $20\%$ of redundant mappings on average. This further saves on computations in SPVerify and SPVerify$^*$.



(a) DBLP

(b) Amazon

(c) Youtube

(d) LiveJournal

Fig. 14. Average size of messages $R_k$s when $c = 11$.

**The number of aggregated messages by SPVerify$^*$.** In Fig. 7, since $c$ was set to 6 by default, we used SPVerify for $Q7$-$Q8$, where each $R_k$ is an aggregate of $M$ messages and $M = 10$. As discussed, the messages are small. To study SPVerify$^*$, we then set $c = 11$. We used $Q8$ with DFS and varied the values of $M$ from 10 to 100. Fig. 14 shows the detailed performance of all datasets. We report that for $M = 10$, the message size is the same as those values of $Q8$ DFS in Fig. 7. Importantly, as $M$ increases, the message size decreases accordingly.

## IX. Related Work

While there has been some work on privacy-preserving query processing, due to space limitations, we can only include the work relevant to graph queries.

**Privacy-preserving graph queries.** Cao et al. [4] proposed supporting subgraph queries over an encrypted database with a number of small graphs. Their work protects the privacy of the query, index and data features. However, this work does not address the subgraph isomorphism verification of candidate graphs. Cao et al. [3] studied tree pattern queries over encrypted XML documents. The traversal order for each query (required by their method) is predetermined. In the context of graphs, the order cannot be predetermined. He et al. [15] analyzed vertex reachability, while preserving edge privacy. Gao et al. [12] proposed neighborhood-privacy protection for the shortest distance. It aims to preserve the neighborhood connections and the shortest distances between vertices. Mouratidis et al. [24] proposed a shortest path computation with no information leakage using the PIR protocol [7], whose high computational cost is a known concern. Karwa et al. [19] addressed *subgraph counts* by satisfying the differential privacy of edges. Fan et al. [10], [11] proposed *authenticated* subgraph query services under the classical data outsourcing setting. A subgraph isomorphism verification method that keeps *both* query and data graphs secret was proposed [9]. To our knowledge, this is the first work that subgraph queries are protected, whereas the data graph is publicly known.

**Subgraph isomorphism.** Ullmann [28] proposed a seminal algorithm for subgraph isomorphism. The basic idea is a search with backtracking with respect to the matrix that represents possible isomorphic relationships. In the last decade, several algorithms (*e.g.*, VF2 [8], QuickSI [26] and Turbo$_{\text{iso}}$ [14]) have been proposed to enhance performance significantly. They all require to *traversing* the query on graph data. For instance, VF2 [8] relies on a set of state transitions and traversals on the graph and query. QuickSI [26] optimizes the ordering in traversals of graphs. Turbo$_{\text{iso}}$ [14] exploits neighborhood information and local regions of vertices. Turbo$_{\text{iso}}$ also involves determining an optimal traversal in query processing. However, the traversals themselves carry topological information, which makes privacy preservation complicated if it is possible at all. Wu et al. [31] supports structureless graph queries, as the query structure is automatically formulated. However, the queries are known to the $\mathcal{SP}$.

## X. Conclusion

This paper studies the first practical private approach for subgraph query service: asymmetric structure-preserving subgraph query processing. Our techniques include deriving minimized candidate subgraphs to significant reduce the number of candidate mappings, generating candidate matchings and then candidate mappings without redundancies and verifying candidate mappings without leaking query structures. Our experiments confirm that our techniques are efficient and effective. A future work is to support data values associated with the graphs. We also plan to apply distributed computing once candidate subgraphs are generated.

## References

[1] SNAP. http://snap.stanford.edu/, 2014.

[2] P. Barceló, L. Libkin, and J. L. Reutter. Querying graph patterns. In *PODS*, 2011.

[3] J. Cao, F.-Y. Rao, M. Kuzu, E. Bertino, and M. Kantarcioglu. Efficient tree pattern queries on encrypted xml documents. EDBT, 2013.

[4] N. Cao, Z. Yang, C. Wang, K. Ren, and W. Lou. Privacy-preserving query over encrypted graph-structured data in cloud computing. In *ICDCS*, 2011.

[5] R. Chen, B. Fung, P. Yu, and B. Desai. Correlated network data publication via differential privacy. *The VLDB Journal*, in press.

[6] J. Cheng, A. W.-c. Fu, and J. Liu. K-isomorphism: privacy preserving network publication against structural attacks. SIGMOD, 2010.

[7] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45:965–981, 1998.

[8] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *PAMI, IEEE*, 26(10):1367–1372, 2004.

[9] Z. Fan, B. Choi, Q. Chen, J. Xu, H. Hu, and S. S. Bhowmick. Structure-preserving subgraph isomorphism query services. *submitted to TKDE (under 2nd round review)*, 2014. http://www.comp.hkbu.edu.hk/~zfan/techreport14a.pdf.

[10] Z. Fan, Y. Peng, B. Choi, J. Xu, and S. S. Bhowmick. Towards efficient authenticated subgraph query service in outsourced graph databases. *IEEE Transactions on Services Computing*, 99, 2013.

[11] Z. Fan, Y. Peng, B. Choi, J. Xu, and S. S. Bhowmick. Authenticated subgraph similarity search in outsourced graph databases. *IEEE Transactions on Knowledge and Data Engineering*, 99, 2014.

[12] J. Gao, J. X. Yu, R. Jin, J. Zhou, T. Wang, and D. Yang. Neighborhood-privacy protected shortest distance computing in cloud. SIGMOD, 2011.

[13] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, 2002.

[14] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. SIGMOD, 2013.

[15] X. He, J. Vaidya, B. Shafiq, N. Adam, and X. Lin. Reachability analysis in privacy-preserving perturbed graphs. WI-IAT, pages 691–694, 2010.

[16] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. VLDB, 2004.

[17] H. Hu, J. Xu, Q. Chen, and Z. Yang. Authenticating location-based services without compromising location privacy. SIGMOD, 2012.

[18] H. Hung, S. Bhowmick, B. Truong, B. Choi, and S. Zhou. Quble: towards blending interactive visual subgraph search queries on large networks. *The VLDB Journal*, 23:401–426, 2014.

[19] V. Karwa, S. Raskhodnikova, A. Smith, and G. Yaroslavtsev. Private analysis of graph structure. In *VLDB*, 2011.

[20] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007.

[21] A. Kundu, M. J. Atallah, and E. Bertino. Efficient leakage-free authentication of trees, graphs and forests. *IACR Cryptology ePrint Archive*, 2012:36, 2012.

[22] K. Liu and E. Terzi. Towards identity anonymization on graphs. SIGMOD, 2008.

[23] D. A. Menascé. Qos issues in web services. *IEEE Internet Computing*, (6):72–75, 2002.

[24] K. Mouratidis and M. L. Yiu. Shortest path computation with no information leakage. *PVLDB*, 2012.

[25] NCBI. PubChem. http://pubchem.ncbi.nlm.nih.gov/.

[26] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 2008.

[27] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 2012.

[28] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23:31–42, 1976.

[29] Y. Wu, X. Yan, and S. Yang. Ontology-based subgraph querying. In *ICDE*, 2013.

[30] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.

[31] S. Yang, Y. Wu, H. Sun, and X. Yan. Schemaless and structureless graph querying. *PVLDB*, 2014.

[32] B. Zhou and J. Pei. Preserving privacy in social networks against neighborhood attacks. In *ICDE*, pages 506–515, 2008.

[33] L. Zou, L. Chen, and M. T. Özsu. k-automorphism: a general framework for privacy preserving network publication. *VLDB*, 2009.

[34] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, 2008.