Distributed and Parallel Databases (2021) 39:35–77
https://doi.org/10.1007/s10619-020-07284-0

Secrecy and performance models for query processing on outsourced graph data

Gabriela Suntaxi[1] · **Aboubakr Achraf El Ghazi**[1] · **Klemens Böhm**[1]

Published online: 29 January 2020

Abstract
Database outsourcing is a challenge concerning data secrecy. Even if an adversary, including the service provider, accesses the data, she should not be able to learn any information from the accessed data. In this paper, we address this problem for graph-structured data. First, we define a secrecy notion for graph-structured data based on the concepts of indistinguishability and searchable encryption. To address this problem, we propose an approach based on bucketization. Next to bucketization, it makes use of obfuscated indexes and encryption. We show that finding an optimal bucketization tailored to graph-structured data is NP-hard; therefore, we come up with a heuristic. We prove that the proposed bucketization approach fulfills our secrecy notion. In addition, we present a performance model for scale-free networks which consists of (1) a number-of-buckets model that estimates the number of buckets obtained after applying our bucketization approach and (2) a query-cost model. Finally, we demonstrate with a set of experiments the accuracy of our number-of-buckets model and the efficiency of our approach with respect to query processing.

Keywords Secrecy · Data outsourcing · Graph data · Performance model

1 Introduction

Outsourcing databases to a third-party service provider (SP) has become ubiquitous. While economic and organizational advantages are obvious, database outsourcing remains challenging concerning data secrecy. Databases contain sensitive

✉ Gabriela Suntaxi
gabriela.suntaxi@kit.edu

Aboubakr Achraf El Ghazi
elghazi@kit.edu

Klemens Böhm
klemens.boehm@kit.edu

[1] Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany

information that needs to be protected against adversaries, including the SP. If an unauthorized user accesses the data, she should not be able to learn anything.

Another important trend is that a broad range of real-world datasets exhibits a graph structure. Furthermore, many real graphs, such as the email network or the Web follow a scale-free power-law distribution [2]. At the same time, these graphs often contain sensitive information. Graphs with this characteristic are the data we focus on in this current paper.

In addition to the information attached to nodes, information is also attached to edges. In general, a node can be identified by its label as well as by its degree (number of edges). An adversary can learn information about an individual in a graph if the adversary identifies the node that represents the individual and its connections. Therefore, approaches for secure storage of graph-structured data should protect against leaking this kind of information. Only encrypting node labels is not enough. Next, there have to be provable secrecy guarantees. At the same time, the approaches should not do away with the advantages of database outsourcing. In particular, query processing should take place on the server as much as possible. While we are not aware of any previous work on secure storage featuring a cost model for query processing, this actually is important to (1) have a good understanding of the expected performance of query processing, (2) facilitate comparisons between alternatives, assuming that the alternatives also have a cost model, and (3) predict the impact of parameter changes. Any query optimizer, which is an integral component of a modern database-management system, depends on cost models to come up with good query-executions plans [13, 29, 31].

So there are two requirements on a secure storage scheme for graph-structured data. R1: An adversary, including the SP, must not be able to learn any useful information from the outsourced graph database, except for some predefined information. This must be provable (i.e., secrecy). R2: The approach should support a broad range of queries. It should do so efficiently, with controlled effort, and most of the work should be done at the server side. To quantify this, a performance model is needed.

The first requirement calls for a rigid definition of secrecy. This includes specifying the type of adversaries one is dealing with and the information that one accepts to leak. Here, we consider adversaries who have access to the secretized graph and can observe query executions over time, i.e., the encrypted queries issued and their encrypted results. Our secrecy notion explicitly states all possible leakages which could result from our specified adversary. We prove that, given a secretized graph, an adversary cannot learn any useful information about the original graph beyond the leakage stated. Since existing secrecy notions consider different secrecy guarantees and deal with different types of adversaries, we propose a new one, i.e., formalize the notion just sketched. Related secrecy notions also deal with different types of adversaries. Notions such as the ones presented by Fan et al. [10] and Zhang et al. [43] offer guarantees against chosen- plaintext attacks and known plaintext attacks, respectively. In our scenario, these guarantees are not enough. This is because the edges of the graph can also reveal information. Wang and Lakshmanan [39] define a secrecy notion for XML documents. It is based on the definition of perfect secrecy. Their secrecy notion considers adversaries who have access to (1) the secretized XML documents and (2) some

metadata needed to perform queries. We additionally assume that the adversary can observe query executions over time.

Secure database storage has been widely studied. However, existing techniques such as Hore et al. [18], Aggarwal et al. [1] either cannot be applied to graph-structured data [18], or they do not cover both requirements R1 and R2 [1]. Approaches for graph-structured data such as Syalim et al. [37] do not keep the information of the entire graph. Next, they cannot answer certain queries, such as neighbor and adjacency queries. These two types of queries are essential needs when working with graphs [28]. Other approaches like [1, 16, 18] could exhibit unwanted behavior when being adapted to graph-structured data, e.g., leak information, see Sect. 2. Next, none of these approaches features a model of the costs of query processing that considers relevant characteristics of the graph.

We propose a bucketization approach for secure storage of graph-structured data that meets our requirements. Our approach uses: (1) Encryption of the labels of the nodes and edges to protect them against deterministic chosen-plaintext attacks, i.e., an adversary cannot learn any useful information from the encrypted nodes and edges. (2) Bucketization of the edges to protect against frequency attacks, i.e., an adversary could learn secret information based on the frequency of the ciphertexts. In our scenario, the frequency represents the degree of a node.

It has turned out that subtle design decisions have a significant impact. For example, it makes a big difference regarding secrecy, whether we partition nodes into buckets instead of edges. This is because partitioning nodes could leak information on the graph structure, as we will explain. While our approach works for all types of graph queries in principle, we focus on neighbor and adjacency queries. These queries are essential information needs regarding graphs [28]. We then describe the specifics for these queries, such as division of work between client and server.

Summing up, our contributions are: First, we propose a secrecy notion for graph-structured data based on the concepts of indistinguishability [21] and searchable encryption [9]. Second, after showing that existing design alternatives do not cope with all requirements, given that notion, we propose a solution featuring bucketization for graph-structured data. We show that finding an optimal bucketization is NP-hard. Consequently, we propose a heuristic, which we also evaluate through experiments later, with positive results. Third, we prove that our bucketization scheme fulfills our secrecy notion. Fourth, we come up with a performance model for query processing on scale-free graphs. Our performance model consists of (1) a number-of-buckets model which estimates the number of buckets obtained when applying our bucketization approach, and (2) a query-cost model. Finally, we conduct systematic experiments both on synthetic and on real datasets. They validate the accuracy of our estimation model and demonstrate the efficiency of the proposed bucketization technique.

2 Related work

In this section, we first review existing secrecy notions. Then we analyze work on bucketization for relational databases and on secure storage of graph-structured data. We omit related work that we have already discussed in the introduction.

2.1 Secrecy notions

Searchable encryption was first introduced in Goh et al. [14] and then extended in Curtmola et al. [9], Van Liesdonk et al. [38] and Kamara et al. [20]. Searchable encryption is a technique that allows performing keyword searches in encrypted documents. An index is generated based on the plaintext data to increase the performance of the search process. The index consists of two data structures: (1) an array that stores, for each keyword w, the encrypted set of identifiers of all documents containing w and (2) a look-up table which contains, for each w, information that allows to locate and decrypt the elements from the array. A trapdoor, which is a deterministic algorithm run by the client, allows testing for the occurrence of a keyword in a document. Their secrecy notions consider two adversarial models, namely (1) nonadaptive chosen keyword attacks (IND-CKA1) and (2) adaptive chosen keyword attacks (IND-CKA2). These notions define secrecy for indexes to ensure that an adversary will not be able to learn the content of an encrypted document from its index. Formally, the secrecy notions IND-CKA1/2 comprise secrecy for trapdoors and guarantee that the trapdoors do not leak information about the keywords apart from the outcome and access pattern of a query. Similarly to the secrecy notion proposed in this paper, IND-CKA1/2 use the concept of indistinguishability. This is a concept based on an experiment game between a challenger and an adversary [21]. However, our secrecy notion is applied to graphs, i.e., the challenger takes as input two graphs given by the adversary. In Sect. 4, we present details of our secrecy notion.

2.2 Bucketization on relational databases

Data secrecy in relational databases has been investigated extensively [15, 16, 18]. Several approaches are based on bucketization. In this context, bucketization (1) encrypts each tuple in an original relation as one string, (2) assigns an index to each encrypted tuple. Indexes are generated in such a way that more than one encrypted tuple could have the same index value. Encrypted tuples with the same index value are called partitions. Each index value is related to a partition of the domain of an original attribute. The server stores the secretized relation and the index information. In what follows, we sketch two adaptations of these approaches to graphs and show that these alternatives are not appropriate to solve our problem.

With both adaptations, we represent the edges in a two-attribute relation, T_{Edges}, where each attribute stores one node of the edge. Borrowing from bucketization schemes for relational databases, two alternatives come to mind, one-dimensional bucketization and multidimensional bucketization.

2.3 One dimensional bucketization

Here, the domains of the two-attributes in T_{Edges} are considered as one domain and then divided into partitions. This solution cannot be considered secure because it could exhibit some of the original graph structure.

Example 1 Consider a graph with edges $E = \{(A,B), (A,D), (B,C), (B,D), (C,A), (D,E)\}$. If bucketization assigns Nodes A, B and C to different buckets, the connections between the buckets will share the same structure as the original graph. Table 1 shows the secretized relation. This solution represents a partition of nodes into buckets. The partitions are $[b1, \{A\}], [b2, \{B\}], [b3, \{C\}], [b4, \{D, E\}]$. The relationships between the index values ($b1$, $b2$), ($b2$, $b3$) and ($b3$, $b1$) share the same structure as the original edges E.

2.4 Multidimensional bucketization

With this option, the domain of each attribute is partitioned individually. Given an optimal multidimensional bucketization, this bucketization can be secure. However, finding an optimal multidimensional bucketization with respect to query performance is NP-hard LeFevre et al. [24]. Nevertheless, this NP-hard problem can be solved with heuristics such as in LeFevre et al. [24] and Wang and Du [40]. But these solutions do not consider certain graphs characteristics such as grouping edges of a node in the same partition to answer important graph queries such as neighbor queries efficiently. So these approaches do not solve our problem.

2.5 Secure storage for graph-structured data

Syalim et al. [37] have proposed an approach which guarantees secrecy for the labels of nodes of a graph. Their secrecy model is designed to protect provenance metadata. Provenance metadata is information that allows tracing who has contributed to the creation of a document. The authors represent the provenance metadata of documents as a directed acyclic graph. The labels of nodes store the provenance data of a specific document version, and the edges represent the

Table 1 Secretized relation of Example 2.1

e-tuple	Node 1	Node 2
$enc(A, B)$	$b1$	$b2$
$enc(A, D)$	$b1$	$b4$
$enc(B, C)$	$b2$	$b3$
$enc(B, D)$	$b2$	$b4$
$enc(C, A)$	$b3$	$b1$
$enc(D, E)$	$b4$	$b4$

relationships between a version of a document and its successors. They use a multiple layer encryption scheme that guarantees that only authorized users who have the corresponding decryption keys can access the provenance metadata they are authorized to. Since the labels of the nodes are encrypted, an adversary who does not possess the encryption keys cannot learn the original plaintexts. However, in general, an adversary can identify a node by its degree and learn the relationships between nodes without knowing the content of the labels, i.e., edge leakage. Our approach guarantees that neither nodes nor edges leak information. Regarding query processing, the authors consider two types of queries, i.e., access to the label of a node and access to the label of the parent nodes of a node. These two types of queries may be enough for provenance metadata. In general, however, different graph-specific types of queries should be supported. An approach for finding the shortest path between two nodes in a directed graph is presented in He et al. [17]. Random perturbation of the edges is required in order to offer edge privacy. The perturbation modifies the structure of the graph to some extent. Therefore, query results only are approximate. As XML documents are a specific kind of graph, we briefly turn to this research direction as well. Wang and Lakshmanan [39] proposed an encryption scheme for XML documents which considers different levels of granularity for encryption, i.e., the XML document is divided into blocks of different sizes, and then each block is encrypted as a whole. A block can contain subtrees of the XML document at any depth, e.g., parent and child elements, or just the content of chosen elements. Cao et al. [7] proposed a solution for evaluation of tree pattern queries in encrypted XML documents. The authors take as starting point that XML documents have a domain hierarchy, i.e., parent and child hierarchy. Each element of the XML document is given a position based on the domain hierarchy. The authors use the position of the elements to create a vector for each XML document. The vectors are encrypted to ensure secrecy. Similarly, a tree pattern query is transformed into a vector. Then the evaluation of a tree pattern query requires measuring the distance between the encrypted vector that represents the XML document and the encrypted vector that represents the query. These two approaches [7, 39] require the existence of a domain hierarchy such as parent-child, to create blocks or vectors, respectively. In graph-structured data, such a hierarchy typically does not exist.

To summarize, none of the related approaches we are aware of does address Requirements R1 and R2.

3 Preliminaries and notation

We now present some notation that we will use in the paper.

Definition 1 A **graph** G is a tuple (V, E), where V is a set of nodes and $E \subseteq V \times V$ is a relation between nodes. $|V|$ and $|E|$ are the number of nodes and edges, respectively, and $\mathcal{G}$ is the set of all graphs.

Without loss of generality, we assume that the relationships between the nodes are directed. An undirected edge can be represented by two directed edges.

Definition 2 Given a graph G, the **size of** G, dubbed $size(G)$, is a tuple $(|V|, |E|)$ that contains the number of nodes and the number of edges of G.

Definition 3 Given a graph G and a node $u \in V$, the **degree** of u, $deg(u)$, is the number of outgoing edges of u.

Definition 4 Given a graph $G = (V, E)$, the **multiset of degrees** $Deg(G)$ is the multiset that contains the degree of each node $u \in V$.

Definition 5 A **neighbor query** $Q_{Neighbor}(G, u)$ of a graph $G = (V, E)$ and a node $u \in V$ returns the set of all nodes adjacent to u in G: $Q_{Neigbhor}(G, u) = \{v \in V \mid (u, v) \in E\}$.

Definition 6 An **adjacency query** $Q_{Adjacency}(G, u, v)$ of a graph $G = (V, E)$ and a pair of nodes u, v, checks whether node u is adjacent to node v: $Q_{Adjacency}(G, u, v) = true$ iff $(u, v) \in E$.

Definition 7 Given a graph G, a **query history** qH_G is a list of n queries $qH_G = [q_1, \ldots, q_n]$ over G , where q_1 is the earliest query in the list, and $q_1, \ldots, q_n$ either are neighbor or adjacency queries.

Definition 8 Given a value v and a multiset of values V, the **frequency of** v is the number of occurrences of v in V.

Definition 9 A **deterministic encryption scheme** $E_d = (k_{gen}, enc_d^K, dec_d^K)$ applied to a plaintext m consists of three parts: (1) a key generation algorithm k_{gen} that returns a cryptographic key K; (2) a deterministic encryption algorithm enc_d^K that takes the cryptographic key K and the plaintext m to compute a ciphertext c, and (3) a deterministic decryption algorithm dec_d^K that takes the cryptographic key K and the ciphertext c to revert the deterministic encryption, such that $dec_d^K(enc_d^K(m)) = m$. Deterministic encryption involves no randomness and always produces the same ciphertext for a given plaintext m and key K.

Definition 10 A **probabilistic encryption scheme** $E_p = (k_{gen}, enc_p^K, dec_d^K)$ applied to a plaintext m consists of three parts: (1) a key generation algorithm k_{gen} that returns a cryptographic key K, (2) a probabilistic encryption algorithm enc_p^K that takes the cryptographic key K and the plaintext m to compute a ciphertext c and (3) a deterministic decryption algorithm dec_d^K that takes the cryptographic key K and the ciphertext c to revert the probabilistic encryption, such that $dec_d^K(enc_p^K(m)) = m$. Probabilistic encryption involves randomness, as follows: When encrypting the same plaintext m with key K several times it produces different ciphertexts.

Definition 11 Given an encryption scheme $\mathcal{E}$, the **security parameter** of $\mathcal{E}$ is a number that grows monotonically with the number of operations performed during the encryption-decryption process and with the size of the cryptographic key used.

The next notion is a standard one from cryptography [21].

Definition 12 A **function** f **of type** $\mathbb{N} \rightarrow \mathbb{R}_0^+$ **is negligible** iff $\forall c \in \mathbb{N} : \exists n_0 \in \mathbb{N}$ such that for $n \geq n_0, f(n) < n^{-c}$.

Definition 13 A **chosen-plaintext attack** is an attack in which the adversary can chooses several plaintexts to be encrypted and obtain their corresponding ciphertexts. Then the adversary sends to the challenger two plaintexts m_0 and m_1 and receives the ciphertext of one of them. The goal of the adversary is to distinguish if she has received the ciphertext of m_0 or m_1.

Definition 14 A **deterministic chosen-plaintext attack** is a relaxed notion of Definition 13, in which the adversary never sees the same plaintexts encrypted with the same key more than once. The adversary can choose several plaintexts to be encrypted and obtain their corresponding ciphertexts. The adversary sends two different plaintexts m_0 and m_1 to the challenger, with the restriction that the plaintexts are distinct from the messages sent previously. The adversary receives the ciphertext of one of them. The goal of the adversary is to distinguish if she has received the ciphertext of m_0 or m_1.

A deterministic encryption scheme is not secure against chosen-plaintext attacks. To offer secrecy guarantees against this type of attacks, the encryption scheme must be probabilistic [21]. However, if a deterministic encryption scheme does not encrypt the same plaintext more than once, i.e., the plaintext messages to be encrypted are unique, then a deterministic encryption scheme is secure against deterministic chosen-plaintext attacks [3, 5]. A probabilistic encryption scheme offers secrecy guarantees against both type of attacks, Definitions 13 and 14.

4 The secrecy notion

In this section, we describe the secrecy notion we target at. The target is to build a secrecy notion with the following characteristic: If an algorithm used to secretize a given graph fulfills this notion, it is guaranteed that it only leaks the information stated in Definition 24. Allowing some leakage is standard with state-of-the-art secrecy notions, especially in the area of searchable encryption [6, 8, 9, 27, 30, 41].

Definition 15 A **data structure** *ds* is any type of structure that can be implemented in a database.

This definition—naturally—is somewhat vague. For instance, something can be two separate data structures, or it could be counted as one. This is not important for our purposes, we just need the definition to bring some rigidity to the definitions that follow. There will be concrete instantiations, later on, doing away with this vagueness.

Definition 16 Given a graph G, a **graph-secretization algorithm** τ is an algorithm that takes as input G and transforms it to a list of d data structures $[ds_1, \ldots, ds_d]$ so that some information from G is kept secret. We call the result of applying τ to G, the **transformed graph** $transformed_G$.

Which information τ keeps secret, and how it does so, depends on the secrecy definition that the algorithm complies with.

Definition 17 An **adversary** $\mathcal{A}$ is a malicious user who has access to the transformed graph $transformed_G$ and can observe the query executions over it.

We now define the information leakage that we are willing to accept. We consider four leakages. The first two, called access and search patterns, are related to the execution of queries, Definitions 23 and 21 respectively, and the second two, called the size of G and multiset of degrees, are related to the graph itself, Definitions 2 and 4 respectively.

For a given graph G, and a query history qH_G with n queries, the access pattern is a list of n elements that contains information about G. In concrete, if the i-th query is a neighbor query, the i-th element in the list is a lower, x, and upper bound, y, on the degree of the queried node. If the i-th query is an adjacency query, then the i-th element in the list is a Boolean, stating whether the queried edge exists in G or not.

Definition 18 Given a graph G and a neighbor query $Q_{Neighbor}(G, u)$, a **neighbor access pattern** $\alpha(Q_{Neighbor}(G, u))$ of $Q_{Neighbor}(G, u)$ is a tuple (x, y) such that $x \leq deg(u) \leq y$.

Definition 19 Given a graph G and an adjacency query $Q_{Adjacency}(G, u, v)$, an **adjacency access pattern** $\alpha(Q_{Adjacency}(G, u, v))$ of $Q_{Adjacency}(G, u, v)$ is a Boolean which is *true* iff $(u, v) \in E$.

Definition 20 Given a graph G and a query history qH_G, an **access pattern** $\alpha(qH_G)$ **induced by** qH_G is a list $[\alpha(q_1), \ldots, \alpha(q_n)]$ such that for all $i \in \{1, \ldots, n\}$:

- if q_i is a neighbor query, then $\alpha(q_i) = \alpha(Q_{Neighbor}(G, u))$
- if q_i is an adjacency query, then $\alpha(q_i) = \alpha(Q_{Adjacency}(G, u, v))$.

Definition 21 Given a graph G and a query history qH_G, the **search pattern** $\sigma(qH_G)$ induced by qH_G is a $n \times n$ binary symmetric matrix with the following entries: for $1 \leq i, j \leq n$, $\sigma[i][j] = 1$ if query $q_i = q_j$, and 0 otherwise.

Since we want to avoid that an attacker $\mathcal{A}$ learns the exact degree of a queried node, we limit the neighbor access pattern that we are willing to leak. To do so, we introduce the notions of degree uncertainty and z-access pattern.

Definition 22 Given a (1) a graph G, (2) its transformed graph $transformed_G$, and (3) an access pattern $\alpha(qH_G)$, the **degree uncertainty** z of $transformed_G$ is an integer so that for all neighbor access patterns in $\alpha(qH_G)$ it holds that $|x - y| \geq z$.

Example 2 Think of (1) a graph G, (2) its corresponding $transformed_G$, and (3) the degree uncertainty $z = 5$. In this case, one can be sure that independent from the queries executed over $transformed_G$, the absolute difference between the lower and upper bounds of the neighbor access pattern of any node in G that one can learn is always greater than or equal to 5.

Definition 23 Given a degree uncertainty $z \in \mathbb{Z}$, a **z-access pattern** $\alpha_z(qH_G)$ is an access pattern in which all neighbor access patterns fulfill z.

Definition 24 Given a transformed graph $transformed_G$ and a degree uncertainty $z \in \mathbb{Z}$ with $z \geq 1$, the **accepted information leakage**, with the *Ind-Graph* secrecy notion to be defined, is: (L1) the z-access pattern $\alpha_z(qH_G)$, (L2) the search pattern $\sigma(qH_G)$, (L3) the size of the original graph $size(G)$, and (L4) the multiset of degrees $Deg(G)$.

Although leakage L1 could lead to attacks such us the ones featured in Kellaris et al. [22], Naveed et al. [32], L1 and L2 are in line with the work described in Curtmola et al. [9], Bösch et al. [6]. L3 is similar to the leakage accepted by Wang et al. [41], Meng et al. [30]. Leakage L4 is used only to evaluate the trade-off between secrecy and performance, and for specific graph-secretization algorithms that only leak some of the multiset of degrees. L4 can be relaxed further, as we will prove in Sect. 5.6. Proposing a graph-secretization algorithm that guarantees secrecy against the information leakage L1–L4 is out of the scope of this paper and is future work.

To evaluate the secrecy guarantees offered by a graph-secretization algorithm, one needs a secrecy notion, i.e., given an adversary with certain knowledge, when does a secrecy breach indeed occur.

We propose a secrecy notion for graph-structured data called Graph Indistinguishability, *Ind-Graph*. Our secrecy notion is based on the concepts of indistinguishability presented by Katz and Lindell [21] and the notion of searchable encryption presented by Curtmola et al. [9]. Katz and Lindell [21] have proven that the concept of indistinguishability is equivalent to conventional semantic secrecy, i.e., an adversary is not able to learn any partial information on the plaintext of a given ciphertext. The reason why we use indistinguishability as our secrecy notion is the one featured in Katz and Lindell [21]: Having an algorithm, it is easier to show that it fulfills indistinguishability than the concept of semantic secrecy. However, the secrecy guarantees are the same. The concept

of indistinguishability is defined based on an indistinguishability experiment between an adversary and a challenger. Before describing such an experiment, we define first the trapdoor term for a given query.

Definition 25 Given a query q over graph G and a key K, a **trapdoor** t is the output of a deterministic algorithm $T(K, q)$ that allows to execute q over $transformed_G$.

Observing the execution of a list of queries is equivalent to having their trapdoors. In Sect. 5.6, we specify the trapdoors for our graph-secretization algorithm.

The idea behind the indistinguishability experiment is that an adversary $\mathcal{A}$ is allowed to feed two inputs in the experiment. The challenger randomly chooses one of the inputs and uses an algorithm to secretize the selected input. $\mathcal{A}$ receives the output of the experiment, but she does not know which one has been the selected input. The output of the experiment should represent all the information that $\mathcal{A}$ can observe, and its inputs should represent all the information needed to produce the output mentioned. The selection of the inputs can, however, be restricted based on the accepted information leakage, Definition 24. At the end of the experiment, $\mathcal{A}$ has to "guess" the input chosen by the challenger. The final output of the experiment is defined to be 1 if $\mathcal{A}$ "guesses" correctly and 0 otherwise. If the final output is 1, we say that $\mathcal{A}$ has succeeded.

We use the left arrow "←" to indicate that the value on the right hand side is assigned to the term on the left hand side.

Definition 26 Let $\mathcal{A}$ be an adversary, τ a graph-secretization algorithm and K a cryptographic key. The **indistinguishability experiment *Ind-Graph*** is defined as follows:

$Ind\text{-}Graph_{\mathcal{A},\tau}(K)$

$\mathcal{A}$ *chooses* $(G_0, qH_{G_0}, G_1, qH_{G_1})$
$b \leftarrow\$ \{0,1\}$
for $1 \leq i \leq n$
  $t_{b,i} \leftarrow t_k(q_i)$
let $T_b = [t_{b,1}, ..., t_{b,n}]$
$\overline{b} \leftarrow \mathcal{A}(\tau(G_b), T_b)$
return 1 *if* $b = \overline{b}$ *else* 0

with the restrictions that $\alpha_z(qH_{G_0}) = \alpha_z(qH_{G_1}), \sigma(qH_{G_0}) = \sigma(qH_{G_1})$, $size(G_0) = size(G_1)$ and $Deg(G_0) = Deg(G_1)$.

Definition 27 A graph-secretization algorithm τ is called ***Ind-Graph*** secure if the function $Adv^{\tau}_{\mathcal{A}}(K) := \left| Pr[Ind\text{-}Graph_{\mathcal{A},\tau}(K) = 1] - \frac{1}{2} \right|$ is negligible for any adversary $\mathcal{A}$ whose computational effort is bounded to polynomial time.

5 Our secrecy approach

We now describe our graph-secretization algorithm, called *bucketization algorithm*. We first give an overview and describe the underlying system architecture. Then we describe the challenges, formalize the problem, and present our approach.

5.1 Overview and system architecture

We consider a database-as-a-service setting where a third-party service provider stores data owned by the clients. Clients apply techniques to secretize the data before passing it to the service provider, in order to have data secrecy.

Definition 28 Given a graph G, a **bucket** b is a finite set of edges of G. Each bucket has a *bucketID* denoted by $bucketID(b)$. All buckets have the same capacity denoted by *maxEdges*, i.e., a bucket can store at most *maxEdges* edges. The *frequency of bucket* b, $freq(b)$, is the number of edges that bucket b stores. The set of buckets of G that stores all edges of G is denoted by S^G_B.

Definition 29 Given a graph G and a corresponding set of buckets S^G_B, the **index information** is a map $m : V \rightarrow S^G_B$ that, for each $u \in V$ contains the set of *bucketIDs* of buckets that store at least one outgoing edge of u.

Definition 30 A **bucketization structure** B of a given graph G is a representation of G consisting of two parts, (1) a set of buckets S^G_B and (2) the *index information*. We call the set of all possible bucketization structures *Buck*.

From now on, we use the terms bucketization structure B transformed graph $transformed_G$ when we refer to the output of our bucketization scheme and the output of any graph-secretization algorithm, respectively.

Figure 1 illustrates a bucketization structure. We use parentheses to denote tuples and curly brackets to denote sets.

Definition 31 A **bucketization function** $buck : \mathcal{G} \rightarrow Buck$ is a function that generates a bucketization structure B for a graph $G \in \mathcal{G}$.

Definition 32 Given a bucketization structure B, an **encryption function** $enc : Buck \rightarrow Buck$ performs an encryption of B as follows: (1) In the *index information*, each label of a node is encrypted deterministically, and the *bucketIDs* are encrypted probabilistically. (2) In the set of buckets, each edge is encrypted deterministically.

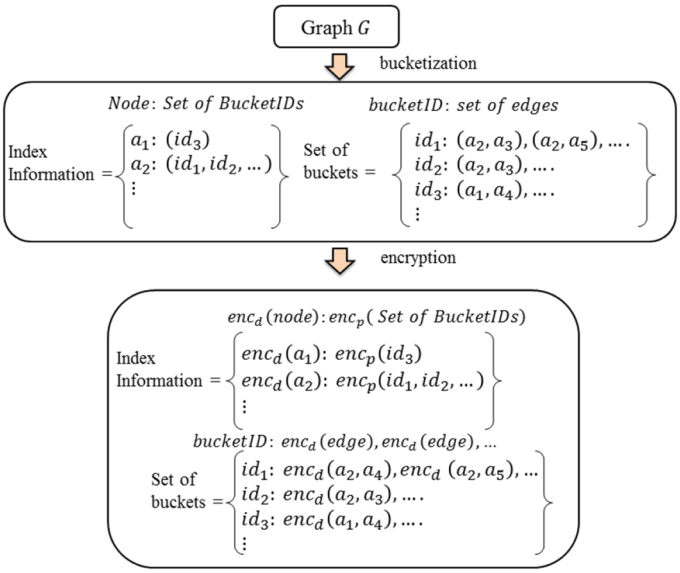

Fig. 1 Bucketization and encryption on Graph G

We use encryption techniques, deterministic and probabilistic, to protect against deterministic chosen-plaintext attacks. In addition to that, we use bucketization techniques to protect against frequency attacks. Regarding bucketization, we aim for an optimal bucketization concerning query performance. The specifics of our approach will be explained in Sect. 5.4.

To achieve data secrecy, before outsourcing a graph G, the client applies a bucketization function on G, and after encryption, the bucketization structure is outsourced to the SP. Figure 2 illustrates the system architecture of the database outsourcing model. It consists of a trusted client and an untrusted server. Since the server that stores the data is untrusted, the client should have some computational

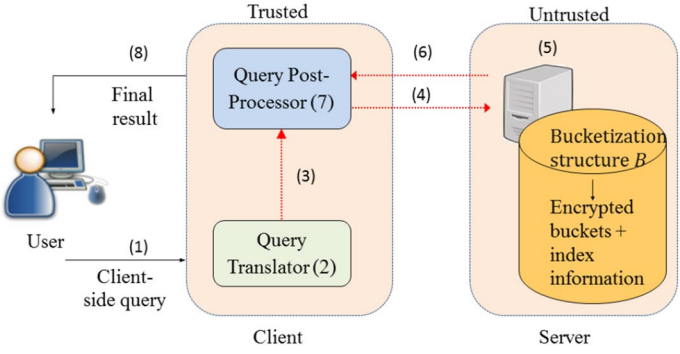

Fig. 2 System architecture and query processing

capabilities to process queries and results between the users and the server. We assume that the client has two components for query processing, namely the query translator and the query post-processor. Query processing is as follows: (1) A user sends a query to the client. (2) The query translator translates the query into a list of queries, called server-query list, which contains one or several server-side queries and one filtering client-side query. The server-side queries in the server-query list, apart from the first one, are in general not concrete queries, i.e., they require additional information to be executed. (3) The query translator sends the server-query list to the query post-processor. (4) The query post-processor sends the next server-side query in the server-query list to the server. (5) The server executes the server-side query, and (6) sends the encrypted results to the client. (7) The query post-processor decrypts the results. If there still are server-side queries in the server-query list, the query post-processor uses the decrypted results to instantiate the next server-side query. Steps 4, 5, 6 and 7 start again until there are no more server-side queries in the server-query list. Finally, the query post-processor gets the final encrypted results, decrypts them, executes the filtering client-side query and (8) sends the result to the user.

5.2 Bucketization: challenges

The encryption function encrypts the label of the nodes with deterministic encryption. Deterministic encryption is secure under deterministic chosen-plaintext attacks [3, 5]. However, frequency attacks are still feasible. In order to secure against frequency attacks, we use a bucketization technique tailored to graph-structured data. Finding a bucketization that guarantees both secrecy as well as good query performance is challenging. This is because it is not obvious how to assign edges to buckets, see Examples 3 and 4. The bucketization structure may expose the frequency of buckets.

Example 3 Think of an email network with nodes $V = \{Alice, Bob, Carol, Dan, Eva\}$ and edges $E = \{(Alice,Bob), (Alice,Dan), (Alice,Carol), (Alice,Eva), (Bob, Dan), (Carol, Eva), (Carol, Alice), (Dan, Carol), (Eva, Bob)\}$. Assume that we apply a bucketization algorithm that assigns edges randomly and stores two edges per bucket. In the worst case, the four edges of Alice are assigned to four different buckets. This means that it is necessary to access four buckets to retrieve the edges of Alice. Then the overall query processing effort, i.e., client and server workload, is rather large, because the server has to access more buckets, and the client has to filter more data.

Example 4 Consider Example 3. If each bucket stores all the edges belonging to only one node and no other edges, the frequency of each bucket reveals the node degree. An adversary who knows the degree of each node in the network, i.e., the number of emails that each user has sent, can conclude that the bucket with four edges corresponds to Alice and the one with two edges to Carol.

Definition 33 Given a graph $G = (V, E)$, a set of buckets S_G^B, a node $u \in V$ and a bucket $b \in S_G^B$, **a link between bucket** b **and the degree of node** u exists if $\forall b' \in S_G^B \setminus \{b\}, \forall v \in V \setminus \{u\} : freq(b) \neq freq(b') \wedge deg(u) \neq deg(v) \wedge freq(b) = deg(u)$.

Assigning edges to buckets randomly is likely to bog down query performance, cf. Example 3. Then the edges of a node should be stored in as few buckets as possible. At the same time, storing all edges of a node in one bucket creates a link between the degree of nodes and their corresponding buckets, which might affect secrecy. Then to avoid information leakage, buckets should be indistinguishable. We aim for an equal frequency of buckets, i.e., all buckets should reach the maximal capacity *maxEdges*. Since an assignment may not always yield full buckets so far, it is promising to merge them a posteriori and/or add dummy edges; our approach will feature both. Of course, the total number of dummy edges should be as small as possible. Preliminary experiments of ours have shown that dummy edges do increase the overall query-processing time significantly both at the client and the server.

5.3 The optimal bucketization problem

The optimal bucketization problem is as follows:

Given a graph $G = (V, E)$ as input, we search for a bucketization B that meets Constraints $c_1 - c_4$:

- (c_1) Each edge $(u, v) \in E$ is assigned to one bucket.
- (c_2) Each bucket stores at most *maxEdges* edges, where *maxEdges* is a given parameter.
- (c_3) Edges adjacent to the same node are placed in as few buckets as possible. Formally, let the function $ind : V \times Buck \rightarrow \mathbb{N}$ be as follows: $ind(u, S) := |\{b \in B \mid \exists x \in V : (u, x) \in b\}|$. Given a node u and a bucketization B, the function *ind* returns the number of buckets that store the edges of node u. Then $\forall B' \in Buck, \forall u \in V : ind(u, B) \leq ind(u, B')$.
- (c_4) The total number of buckets should be as small as possible (while prioritizing Constraint c_3).

We prioritize Constraint c_3 over c_4 so that query performance is not affected, see Example 3.

Definition 34 An **optimal bucketization** is a bucketization that meets Constraints c_1 to c_4.

Our optimal bucketization problem is NP-hard. To prove this, we reduce the Bin-packing problem (BP problem) [19] to our problem. The BP problem has been proven to be NP-complete in Johnson [19] and strongly NP-complete in Garey and Johnson [11]. The hardness result, together with the proofs, are in Appendix.

5.4 The bucketization algorithm

Because finding an optimal bucketization is NP-hard, we propose a heuristic that aims to meet the constraints established in Sect. 5.3. Our algorithm has an initialization phase and a merging phase. Due to the complexity of optimal bucketization problem, we use heuristics in the merging phase.

5.4.1 The initialization phase

Definition 35 Given a graph G, the **initial bucketization** B_0 is the result of the initialization phase of the bucketization algorithm applied to G.

Algorithm 1 is the initialization phase of our bucketization approach for a graph G. Since the length of the ciphertext could reveal the length of the plaintext, we pad the label of the nodes to avoid this leakage (Line 1). Formal secrecy proofs are presented in Sect. 5.6.

Algorithm 1: Initialization ()

Input : Graph: $G(V, E)$, int: $maxEdges$
Output: initial bucketization: B_0

```
1 labelOfNodes.pad();
2 foreach v in V do
3 |   create (ceil(1, v.numberOfEdges()/maxEdges)) buckets;
4 |   assign randomly up to maxEdges edges of v to each bucket;
5 |   generate the corresponding index information;
```

Example 5 illustrates how the assignment of edges works, and Example 6 explains the need for randomness with this assignment.

Example 5 We set $maxEdges = 10$. Given a node v that has 27 edges, three buckets will be created. 10 random edges are chosen from the 27 edges and are assigned to the first bucket, another 10 random edges are chosen from the remaining ones for the second bucket, and the 7 remaining edges go to the third bucket.

Example 6 For the sake of an easy example, consider a setting where emails can be revoked without difficulty. For this example, we consider the buckets in Fig. 3. Assume that the bucketization algorithm does not assign edges randomly; instead, it assigns edges alphabetically. If an adversary knows that user A has sent four emails to persons B, C, D, and E, although the edges are encrypted, the adversary will learn that bucket $b2$ stores the last edge, i.e., the email sent to user E. Gaining this knowledge is a leakage. This leakage can lead to an attack with an extended adversary

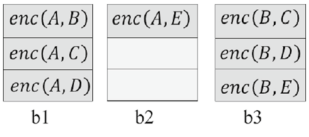

Fig. 3 Illustration of Example 6

model where the adversary has access to the log history. In the case where Bucket $b2$ disappears, the adversary knows that one email was revoked, and she will learn that the revoked email was the last one, i.e., the email sent to E. A random assignment of edges reduces the chance of the adversary learning extra information.

After the initialization phase of the bucketization process, all edges have been placed into their corresponding buckets. At this point, some buckets may not have reached the maximal capacity, *maxEdges*. Even if we encrypt the buckets at this stage, the initial bucketization is not secure. If the degree of a node is less than or equal to *maxEdges*, its edges have been placed in one bucket exclusively. Then for nodes with degree less than or equal to *maxEdges*, there is a link between their degree and the frequency of their corresponding buckets. So the adversary will be able to identify the buckets of these nodes, see Example 4.

One could consider inserting dummy edges at this stage to avoid leaking information. Although adding dummy edges solves secrecy problems, the query performance of this solution is affected. The overall query processing time without the merging phase increases because the number of buckets at the initialization phase is greater than the number of buckets after the merging phase, and this slows down the server.

5.4.2 The merging phase

Definition 36 Bucket merging is an operation that puts the content of two buckets in a new one and then deletes the two emptied ones.

In this phase, the algorithm merges buckets to fulfill Constraint $c4$. Algorithm 2 identifies pairs of buckets that can be merged to obtain buckets with the same frequency. Different heuristics are conceivable at this stage. We choose a First Fit Decreasing approach (FFD) [12]. We will justify this decision after having explained the algorithm. When the algorithm starts, it creates three sets: (1) B', which stores buckets with frequency less than *maxEdges*, (2) B_f, which stores full buckets, and (3) B_m, an auxiliary set that stores the buckets resulting from a merge. For each Bucket $b_i \in B'$, the algorithm searches for the first Bucket b_j in B_m that can be merged with b_i. A merge is possible if $|b_i| + |b_j| \leq maxEdges$. If it finds a possible merge, the function $merge(b_i, b_j)$ does the following: (1) create a new Bucket b to store the edges of b_i and b_j, (2) remove buckets b_i and b_j from B' and B_m, and (3) update the *index information*. If b reaches its maximal capacity, b is placed in B_f. Otherwise, it is placed in B_m so that it can be considered again for a merge. If there is no Bucket b_j available for a merge, b_i is placed in B_m. After the merging process, dummy edges are added to the buckets that have not reached *maxEdges*. We pad the set of *bucketIDs* such that all sets have the same length. The edges inside each bucket are encrypted with enc_d^K individually. In the *index information* the labels of the nodes are encrypted with enc_d^K, and the set of *bucketIDs* is encrypted with enc_p^K.

Definition 37 Given a graph G, a **final bucketization** B_f is a bucketization resulting from the initialization and bucket merging phases applied to G.

Algorithm 2: Merge buckets ()

```
   Input  : initial bucketization B_0, int: maxEdges
   Output: final bucketization: B_f
 1 Initialize: B' := {b ∈ B_0 | b.numberOfEdges() < maxEdges}; B_f := B_0 \ B'; B_m := {}
 2 Order B' by number of edges in decreasing order;
 3 foreach b_i ∈ B' do
 4 |   foreach b_j ∈ B_m do
 5 |   |   if b_i fits in b_j then
 6 |   |   |   b ← merge(b_i, b_j);
 7 |   |   |   delete b_i, b_j;
 8 |   |   |   if b.numberOfEdges() = maxEdges then dd b to B_f;
 9 |   |   |   else add b to B_m;
10 |   |   |   break;
11 |   if (b_i ∈ B') then move b_i to B_m;
12 foreach b ∈ B_m do
13 |   b.addDummyEdges();
14 |   add b to B_f
15 return B_f;
```

Notice that two nodes could have the same set of *bucketIDs*. Encrypting the set of *bucketIDs* with deterministic encryption could lead to frequency attacks. The use of probabilistic encryption prevents these attacks. In Sect. 5.6, we present formal secrecy proofs.

Notice that two nodes could have the same set of *bucketIDs*. Encrypting the set of *bucketIDs* with deterministic encryption could lead to frequency attacks. The use of probabilistic encryption prevents these attacks. Formal secrecy proofs are in Sect. 5.6.

Lemma 1 *The worst case solution with our bucketization algorithm with the FFD approach is off by a factor of* $\frac{11}{9}$ *from the optimal one.*

Proof Garey and Johnson [12] have proven that the worst case solution for the bin packing problem with the FFD approach is off by a factor of $\frac{11}{9}$ from the optimal one. Our algorithm uses the FFD approach in the merging phase. □

Other heuristics for the merging phase, such as Best Fit and Next Fit, have a worse approximation ratio, $\frac{17}{10}$ and 2, respectively, Garey and Johnson [12].

5.5 Query transformation

Unlike other approaches such as He et al. [17], our bucketization approach does not lose any information regarding the original graph. Consequently, there is no limitation

regarding the kind of query one can process in principle. However, query processing can have the effect that most or all of the computation is done at the client side. Our focus has been on reducing the client workload for neighbor and adjacency queries. For neighbor queries, the client workload, with our approach, consists of (1) a query transformation process and (2) a decryption and filtering process; the rest of the query execution is done at the server side. For adjacency queries, the client workload, with our approach, consists only of a query transformation process; the rest of the query execution is done at the server side. In the following, we describe the processing of neighbor and adjacency queries.

Client-side queries are transformed into server-side queries. We use the conventions *server.m* and *client.m* to indicate that method *m* runs at the server side and client side, respectively. Algorithm 3 shows the transformation process for neighbor queries. The client encrypts node *u* and generates the server-side query. This query retrieves the set of *bucketIDs* of the encrypted node $enc_d^K(u)$ from the *index information* and returns it to the client. The client decrypts it and generates a new server-side query. This new query retrieves the edges stored in buckets whose *bucketID* corresponds to one of the decrypted *bucketIDs*. Then the client decrypts the edges and filters false positives. So there is no uncertainty in the query answers, i.e., query answers are accurate. Algorithm 4 shows the procedure for adjacency queries, $Q_{Adjacency}(G, u, v)$. The client starts by encrypting the two nodes in the query with encryption $enc_d^{key}(u, v)$ and generating the server-side query. The server-side query searches in the *set of buckets* for these encrypted edges. If there exists such an edge, the nodes are adjacent.

Algorithm 3: Neighbor query processing over B

```
   Input  : Q_Neighbor(G, u), key K
   Output: Edges := {}
 1 Initialize: EncBucketIDs:={}, BucketIDs:={}, EncEdges:={}, EdgesTemp :={};
 2 encNode ← client.enc_d^K(u);
 3 if server.indexInformation.contain(encNode) then
 4 |   EncBucketIDs ← indexInformation(encNode);
 5 |   BucketIDs ← client.dec_p^K (EncBucketIDs);
 6 |   foreach b in BucketIDs do
 7 |   |   e ← server.SetOfBuckets.ValueOf(b);
 8 |   |   add e to EncEdges;
 9 |   foreach e in EncEdges do
10 |   |   EdgesTemp ← client.add.dec_d^K(e);
11 |   foreach e in EdgesTemp do
12 |   |   if !e.isFalsePositive then add e to Edges ;
13 return Edges;
```

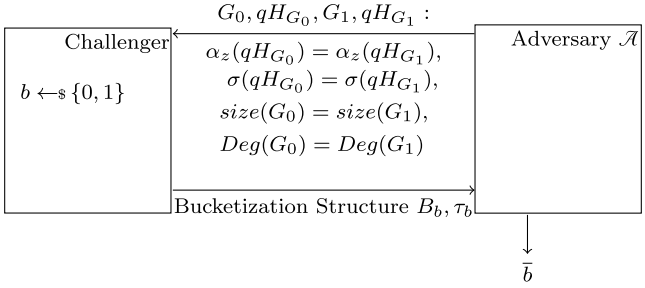

Fig. 4 Indistinguishability experiment *Ind-Graph*

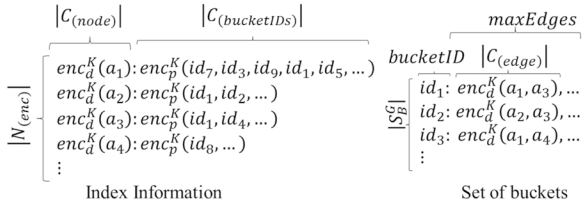

Fig. 5 Abstract output of the bucketization algorithm

Algorithm 4: Adjacency query processing given a bucketization structure B

Input : $Q_{Adjacency}(G, u, v)$, *key* K
Output: Boolean isEdge
1 Initialize: *isEdge* =false;
2 encEdge ← client.$enc_d^K(u, v)$
3 **foreach** *bucket in server.SetOfBuckets()* **do**
4 | **if** *server.setOfBuckets.ValueOf(bucket) contains encEdge* **then** isEdge=true ;
5 return isEdge;

5.6 Our bucketization approach is *Ind-Graph*

In this section, we will prove that our bucketization approach fulfills the secrecy notion *Ind-Graph* defined in Sect. 4.

In our algorithm, the parameter *maxEdges* can be set freely. In Definition 23, we have defined the z-access pattern based on a degree uncertainty z. Then, for a given z, to guarantee that our bucketization algorithm fulfills our secrecy notion, the parameter *maxEdges* must be set accordingly, i.e., *maxEdges* $> z$. We set *maxEdges* $= z + 1$.

Figure 4 shows the setup of the indistinguishability experiment from Definition 26.

Figure 5 illustrates an abstract output of the bucketization algorithm, where $|C_{(node)}|$ is the length of the ciphertext representing an encrypted node, $|N_{(enc)}|$ is the number of encrypted nodes, $|C_{(bucketIDs)}|$ is the length of the ciphertext representing the set of *bucketIDs*, $|C_{(edge)}|$ is the length of the ciphertext representing

an encrypted edge, *bucketID* are random identifiers of the buckets, and $|S_B^G|$ is the number of buckets. We use these notations in our secrecy proofs.

Lemmas 2, 3 and 4 tell us that our bucketization algorithm guarantees that, given two graphs G_0, G_1 and two query histories qH_{G_0}, qH_{G_1} which comply with the restrictions of the experiment, $\mathcal{A}$ cannot say which has been the input selected by the indistinguishability experiment, given the bucketization structure of G_b and the trapdoors. We have organized the proofs of Lemmas 2, 3 and 4 as follows: We demonstrate that for both inputs given by $\mathcal{A}$, the properties of the *index information*, the *set of buckets* and the trapdoors either are the same, or their differences do not allow $\mathcal{A}$ to decide which one has been the input selected.

To facilitate the proof, we first prove that our bucketization algorithm is *Ind-Graph* with respect to the *set of buckets* output, Lemma 2, with respect to the *index information* output, Lemma 3, and with respect to the trapdoors, Lemma 4.

Lemma 2 *Let the following be given:*

1. *An adversary $\mathcal{A}$ that chooses two graphs G_0, G_1 and two query histories qH_{G_0}, qH_{G_1} in line with Definition 26.*
2. *The set of buckets the bucketization algorithm has generated by selecting randomly G_0, qH_{G_0} or G_1, qH_{G_1}.*

Then $\mathcal{A}$ cannot decide whether the tuple (G_0, qH_{G_0}) or the tuple (G_1, qH_{G_1}) has been the input of the bucketization algorithm for maxEdges values equal to or greater than the degree uncertainty z of the z-access pattern.

Proof The properties of the *set of buckets* that do not change for both inputs given by $\mathcal{A}$ are the size of the buckets and the number of buckets $|S_B^G|$. The properties that can be different are the *bucketIDs* and the encrypted edges. First, all buckets have the same size equal to *maxEdges*. Because the bucketization algorithm uses the same value for the parameter *maxEdges*, the size of all buckets for either G_0 or G_1 is the same. Second, $|S_B^G|$ depends on the number of edges of the nodes. After the initialization phase, our bucketization algorithm uses the FFD approach to merge the buckets. Since $Deg(G_0) = Deg(G_1)$, $|S_B^G|$ is the same for both graphs. Third, the *bucketIDs* are generated randomly. Then the bucketization algorithm will generate random *bucketIDs* for both graphs. So $\mathcal{A}$ cannot identify whether the *bucketIDs* correspond to Graph G_0 or G_1. Fourth, the edges in a graph are unique, and they are encrypted deterministically. So the edges are secure against deterministic chosen-plaintext attacks, i.e., an adversary cannot learn any useful information about the original edges, including their frequencies. Therefore, $\mathcal{A}$ cannot recognize whether the encrypted edges in the *set of buckets* correspond to G_0 or G_1. Consequently, an adversary $\mathcal{A}$ cannot distinguish whether the tuple (G_0, qH_{G_0}) or the tuple (G_1, qH_{G_1}) has been the input of the bucketization algorithm. □

Lemma 3 *Let the following be given:*

1. *An adversary $\mathcal{A}$ that chooses two graphs G_0, G_1 and two query histories qH_{G_0}, qH_{G_1} in line with Definition 26.*
2. *The index information the bucketization algorithm has generated by selecting randomly G_0, qH_{G_0} or G_1, qH_{G_1}.*

Then $\mathcal{A}$ cannot distinguish whether the tuple (G_0, qH_{G_0}) or the tuple (G_1, qH_{G_1}) has been the input of the bucketization algorithm for *maxEdges* values equal to or greater than the *degree uncertainty* z of the z-access pattern.

Proof The properties of the *index information* that do not change for both inputs given by $\mathcal{A}$ are the number of encrypted nodes $|N_{(enc)}|$, the length of the ciphertext representing an encrypted edge $|C_{(node)}|$ and the length of the ciphertext representing the set of *bucketIDs* $|C_{(bucketIDs)}|$. The properties that can be different are the encrypted nodes and the encrypted sets of *bucketsIDs*. First, $|N_{(enc)}|$ is the same for both graphs because $|V_0| = |V_1|$. Second, the labels of the nodes are padded before encryption. Then $|C_{(node)}|$ is the same for all the nodes in both graphs. Third, the sets of *bucketIDs* are padded and then encrypted. Then $|C_{(bucketIDs)}|$ is the same for all sets of *bucketIDs* in both graphs. Fourth, the nodes are unique, and they are encrypted deterministically, so they are secure against deterministic chosen-plaintext attacks. Therefore $\mathcal{A}$ cannot distinguish whether the encrypted nodes correspond to G_0 or G_1. Fifth, because the sets of *bucketIDs* are encrypted probabilistically, they are secure against chosen-plaintext attacks, i.e., an adversary cannot learn any useful information on the set of *bucketIDs*, including their frequencies. So $\mathcal{A}$ cannot recognize whether the encrypted sets of *bucketIDs* correspond to G_0 or G_1. Consequently, an adversary $\mathcal{A}$ cannot distinguish whether the tuple (G_0, qH_{G_0}) or the tuple (G_1, qH_{G_1}) has been the input of the bucketization algorithm. □

Lemma 4 *Let the following be given:*

1. *An adversary $\mathcal{A}$ that chooses two graphs G_0, G_1 and two query histories qH_{G_0}, qH_{G_1} in line with Definition 26.*
2. *The list of trapdoors the bucketization algorithm has generated by selecting randomly G_0, qH_{G_0} or G_1, qH_{G_1}.*

Then $\mathcal{A}$ cannot distinguish whether the tuple (G_0, qH_{G_0}) or the tuple (G_1, qH_{G_1}) has been the input of the bucketization algorithm for maxEdges values equal to or greater than the degree uncertainty z of the z-access pattern.

Proof In our setting, a query in a query history qH_G can either be a neighbor or an adjacency query. Given a query history qH_G with n queries, the list of trapdoors T of qH_G consists of n items, one for each query. The content of the trapdoors depends on the type of query. If the i-th query $q_i \in qH_G$ is a neighbor query, the i-th trapdoor t_i consists of (1) an encrypted node that corresponds to the query q_i and (2) a set of *bucketIDs* that correspond to the buckets that store the encrypted edges of that node. If the i-th query $q_i \in qH_G$ is an adjacency query, the i-th trapdoor t_i consists

of an encrypted edge that corresponds to the query q_i. Let T_0 and T_1 be the lists of trapdoors of qH_{G_0} and qH_{G_1}, respectively. First, because $\alpha_z(qH_{G_0}) = \alpha_z(qH_{G_1})$ and $\sigma(qH_{G_0}) = \sigma(qH_{G_1})$ for all $i \in \{1, \ldots, n\}$, the following holds: (1) if the i-th query q_i is a neighbor query, the i-th trapdoor t_i in T_0 has one encrypted node with the same length $|C_{(node)}|$ and the same number of *bucketIDs* as the i-th trapdoor t_i in T_1, (2) if the i-th query q_i is an adjacency query, the i-th trapdoor t_i in T_0 has one encrypted edge, and the i-th trapdoor t_i in T_1 has one encrypted edge as well. Then T_0 and T_1 have the same structure in both cases. Second, T_0 and T_1 could have different content, i.e., the encrypted nodes and the *bucketsIDs* for neighbor queries and the encrypted edges for adjacency queries in T_0 can be different from the ones in T_1. However, as demonstrated in the proofs of Lemmas 2 and 3, the nodes and edges are secure against deterministic chosen-plaintext attacks, and the *bucketsID* are generated randomly. Then, based on the content of the trapdoors, $\mathcal{A}$ cannot distinguish whether the list of trapdoors corresponds to qH_{G_0} or qH_{G_1}. Consequently, an adversary $\mathcal{A}$ cannot distinguish whether the tuple (G_0, qH_{G_0}) or the tuple (G_1, qH_{G_1}) has been the input of the bucketization algorithm. □

We have proven in Lemmas 2, 3 and 4 that the *index information*, the *set of buckets* and the list of trapdoors, generated by our algorithm, do not leak any information that can help an adversary to distinguish the input selected by the indistinguishability experiment—when looked at in isolation. It remains to demonstrate that the entire output does not leak such information. If there are not any links between the data structures of the output, it obviously is enough to demonstrate that each data structure when looked at in isolation does not leak information. However, if there are links between them, it is necessary to show that $\mathcal{A}$ cannot use this information to discern the input selected by the indistinguishability experiment.

Example 7 shows that the output data structures together could leak information even though each data structure separately does not.

Example 7 Think of (1) two graphs $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$, where $V_0 = V_1 = \{A, B, C\}$, $E_0 = \{(A, B), (A, C)\}$, $E_1 = \{(B, A), (B, C)\}$ and (2) two query histories $qH_{G_0} = [Q_{Adjacency}\,(G_0, A, C)]$ and $qH_{G_1} = [Q_{Adjacency}(G_1, A, C)]$. Each data structure in the bucketization structure B_0 of graph G_0 has the same structure as its corresponding data structure in the bucketization structure B_1 of graph G_1. Next, the lists of trapdoors of B_0 and B_1, T_0 and T_1, have the same structure. Additionally, because of the encryption used and the uniqueness of the encrypted values, the content of each data structure cannot be used to distinguish the input given to the algorithm. So they are indistinguishable from each other. However, there are patterns that can only be recognized in the entire structure of the output which one can use to distinguish the input given to the algorithm. In this case, the trapdoor in T_0 occurs in the *set of buckets*, contrary to the trapdoor in T_1. Using this information, $\mathcal{A}$ can recognize the input, given the output of the algorithm, by checking if the ciphertext of the trapdoor occurs in the *set of buckets*. If this occurs, the input is G_0; otherwise, the input is G_1.

Lemma 5 *Let the following be given:*

1. *An adversary $\mathcal{A}$ who chooses two graphs G_0, G_1 and two query histories qH_{G_0}, qH_{G_1} in line with Definition 26.*
2. *The index information, the set of buckets and the list of trapdoors the bucketization algorithm has generated by selecting G_0, qH_{G_0} or G_1, qH_{G_1} randomly.*

Then $\mathcal{A}$ cannot distinguish based on links between the output structures whether the tuple (G_0, qH_{G_0}) or the tuple (G_1, qH_{G_1}) has been the input of the bucketization algorithm for maxEdges values equal to or greater than the degree uncertainty z of the z-access pattern.

Proof The possible links between the output structures are (1) a link between the trapdoor of a neighbor query and the *index information* and (2) a link between the trapdoor of an adjacency query and the *set of buckets*. However, because $\alpha_z(qH_{G_0}) = \alpha_z(qH_{G_1})$ and $\sigma(qH_{G_0}) = \sigma(qH_{G_1})$, these links are the same in both inputs. Consequently, an adversary $\mathcal{A}$ cannot distinguish based on links between the output structures whether the tuple (G_0, qH_{G_0}) or the tuple (G_1, qH_{G_1}) has been the input of the bucketization algorithm. □

Theorem 1 *Our bucketization algorithm fulfills the secrecy notion Ind-Graph, Definition 27, for maxEdges values equal to or greater than the degree uncertainty z of the z-access pattern.*

Proof With Lemmas 2, 3, 4 and 5, we have shown that the *index information*, the *set of buckets*, the list of trapdoors or a combination of them do not violate Definition 27. It remains to prove that the access pattern of our bucketization algorithm does not leak more than the degree uncertainty z. With our bucketization approach, the neighbor access pattern for any neighbor query is the tuple $((\#Buckets - 1) \cdot maxEdges + 1, \#Buckets \cdot maxEdges)$, where $\#Buckets$ is the number of buckets retrieve during the execution of a given neighbor query, i.e., $\#Buckets = \lceil \frac{deg(u)}{maxEdges} \rceil$. Then the degree uncertainty is $|-maxEdges + 1|$. Since we set $maxEdges = z + 1$, the access pattern of our bucketization algorithm does not leak more than the degree uncertainty of the z-access pattern. Thus our algorithm is Ind-Graph secure. □

In the following, we show that our algorithm fulfills our secrecy notion even with a more relaxed Leakage L4. We proceed to define the notion needed, Definition 38, to relax L4.

Definition 38 Given a transformed graph $transformed_G$, the **size of the transformed graph** is a list $sizeT_G = [|ds_1|, \ldots, |ds_d|]$ which contains the size of all data structures in $transformed_G$.

In our scenario, the size of the transformed graph $sizeT_G$ corresponds to the size of the bucketization structure $sizeB$. Our bucketization structure has two data structures, the *index information* and the *set of buckets*. The sizes of the *index*

information and of the *set of buckets* are the number of nodes and the number of buckets, respectively. Then $sizeB = [|V|, |S_B^G|]$.

Example 8 shows two graphs G_0, G_1 that do not fulfill the restriction $Deg(G_0) = Deg(G_1)$. But they still fulfill $sizeB_0 = sizeB_1$, where B_0 and B_1 are the bucketization structures of graphs G_0 and G_1, respectively.

Example 8 Let us consider two graphs G_0 and G_1, each graph has 4 nodes and 8 edges, and their multisets of degrees are $Deg(G_0) = \{2, 2, 2, 2\}$ and $Deg(G_1) = \{3, 2, 2, 1\}$, respectively. Assume that $maxEdge = 4$. Even though $Deg(G_0) \neq Deg(G_1)$, the bucketization algorithm will output 2 buckets for both graphs. Then the size of the bucketization structure for both graphs is the same, i.e., $sizeB_0 = sizeB_0 = [4, 2]$.

Theorem 2 *Our bucketization algorithm fulfills the secrecy notion Ind-Graph, Definition 27, even by replacing leakage L4 with the more relaxed leakage of the size of the transformed graph $sizeT_G$, Definition 38, for maxEdges values equal to or greater than the degree uncertainty z of the z-access pattern.*

Proof With 2, 3, 4 and 5, we have proven that our bucketization algorithm fulfills the secrecy notion *Ind-Graph*. We have used the restriction imposed by Leakage L4, i.e., $Deg(G_0) = Deg(G_1)$, only in the proof of Lemma 2, to demonstrate that the number of buckets $|S_B^G|$ is the same for both graphs given as input by the adversary. This is also guaranteed with the restriction $sizeT_{G_0} = sizeT_{G_1}$, which in our bucketization means that $sizeB_0 = sizeB_1$. Then our bucketization algorithm fulfills the secrecy notion *Ind-Graph* even when replacing Leakage L4 with the more relaxed leakage regarding the size of the transformed graph $sizeT_G$. □

6 Performance model

A performance model is important because it allows predicting the behavior of an algorithm and facilitates meaningful comparisons or evaluations of algorithms. Query optimizers, which are essential features of any modern database system, require such performance models to estimate the costs of various execution plans accurately and to find the most efficient one [13, 29, 31]. If such performance models are not available, query optimizers will resort to coarse estimates, which may be grossly off, with disastrous consequences when it comes to system performance. The difference between the cost of the best execution plan and a random choice could be in orders of magnitude [35]. In our approach, the number of buckets obtained after applying our bucketization algorithm to a graph G is a crucial parameter for query performance, as explained in Sect. 5.2. Estimating the number of buckets is cumbersome, and we estimate a range. But even to estimate this range, it is necessary to have a model that describes relevant properties of the given graph. In the next section, due to the importance of scale-free networks, we review some of their properties and use them to derive the so-called number-of-buckets model and the query-cost model.

6.1 Scale-free networks

Real-world networks have two important features: growth and preferential attachment. Regarding the first feature, real-world networks often are the result of a continuous growth process. Regarding the second one, nodes with a higher degree will have a higher probability to be connected to a new node. This property has the effect that most nodes in the network will have only a few edges, and a few nodes gradually turn into hubs, i.e., their degree greatly exceeds the average. These two features are responsible for the power-law distribution of scale-free networks. Many real-world networks, such as genetic networks or the actor network, follow a power-law distribution [2].

Barabási and Albert [2] introduced a model capturing the properties of scale-free networks, the Barabasi–Albert Model (BA). The properties that we use in our performance model are:

- *Degree Exponent*, γ, which is the exponent of the power-law distribution of scale-free networks. It plays an important role in predicting many properties of these networks, e.g., the highest node degree. The degree exponent of many real networks is between 2 and 3 [2]
- *Growth parameter*, m. At each time step a new node is added to the network with m edges that connect it to m existing nodes.
- *Probability of a node with degree k*, ρ_k. Given the growth parameter m, the probability that a randomly chosen node has degree of k is given by: $\rho_k = \frac{2m(m+1)}{k(k+1)(k+2)}$.
- *Number of Edges*, $|E|$. In the BA, $|E| = m \cdot |V|$.
- *Largest node degree*, k_{max}. The expected value of the largest node degree in the BA is $k_{max} \sim |V|^{\frac{1}{\gamma-1}}$.
- *Lowest node degree*, k_{min}. It is the minimum degree in the network. For k_{min} there is no characterization, each graph can have different values of k_{min}.

6.2 The number-of-buckets model NBM

Recall that after the initialization phase of the algorithm, some buckets are full, and some are not. Lemma 6 captures the number of buckets that have reached their maximal capacity after the initialization phase of the algorithm.

Lemma 6 *The number of full buckets after the initialization phase of the algorithm is* $Bucket^{ini}_{Full} = \sum_{k=k_{min}}^{k_{max}} \left(|V| \cdot \rho_k \cdot \left\lfloor \frac{k}{maxEdges} \right\rfloor \right)$.

Proof Given a node $u \in V$ in a graph G with degree k_u, the number of full buckets generated for u after the initialization phase is $EB^u_{Full} = \left\lfloor \frac{k_u}{maxEdges} \right\rfloor$. EB^u_{Full} is calculated regardless of the other nodes in G. Next, it is required to calculate EB^u_{Full} for all nodes $u \in V$. According to the BA properties, the probability that a randomly chosen node has a degree of k is given by ρ_k. Then the total number of nodes with

degree k is $|V| \cdot \rho_k$. For all the nodes with degree k, the total number of buckets is $|V| \cdot \rho_k \cdot \left\lfloor \frac{k_u}{maxEdges} \right\rfloor$. Finally, to estimate the total number of buckets after the initialization phase, we have to consider all node degrees, which are between k_{min} and k_{max}. □

If we know the number of full buckets, we know the number of edges that have been already stored in these full buckets. Then we can calculate the number of edges stored in non-full buckets, see Lemma 7.

Lemma 7 *The number of edges that have been assigned to buckets that are not full is* $Edges_{NFB} = |E| - Bucket^{ini}_{Full} \cdot maxEdges$.

Proof The number of edges already stored in full buckets after the initialization phase is $Bucket^{ini}_{Full} \cdot maxEdges$. We subtract this number from the total number of edges $|E|$ to obtain $Edges_{NFB}$. □

Using these two lemmas, we introduce the range of the number-of-buckets Model, see Theorem 3.

Theorem 3 *Given a graph* $G = (V, E)$ *that follows the BA Model, the expected number of buckets* E_B *is in the range:*

$$\begin{aligned} Bucket^{ini}_{Full} + \left\lceil \frac{Edges_{NFB}}{maxEdges} \right\rceil \leq E_B \\ \leq Bucket^{ini}_{Full} + \frac{11}{9} \cdot \left\lceil \frac{Edges_{NFB}}{maxEdges} \right\rceil \end{aligned} \tag{1}$$

Proof The lowest value of the range is the number of buckets obtained with the optimal bucketization. With this optimal bucketization, the non-full buckets are merged so that their edges, $Edges_{NFB}$, fill exactly $\left\lceil \frac{Edges_{NFB}}{maxEdges} \right\rceil$ buckets. For the upper bound of the model, the worst performance ratio of the FFD approach used in our algorithm is $\frac{11}{9}$ of the optimal solution. Consequently, the upper bound is the sum of the number of full buckets after the initialization phase, $Bucket^{ini}_{Full}$, and the number of buckets after the merging in the worst case, i.e., $\frac{11}{9} \cdot \left\lceil \frac{Edges_{NFB}}{maxEdges} \right\rceil$. □

Corollary 1 provides a range of the expected number of dummy edges.

Corollary 1 *Given a graph* $G = (V, E)$ *that follows the BA Model, the expected number of dummy edges,* D_E, *is in the range*

$$\left(Bucket_{Full}^{ini} + \left\lceil \frac{Edges_{NFB}}{maxEdges} \right\rceil\right) \cdot maxEdges - |E| \leq D_E \leq \left(Bucket_{Full}^{ini} + \frac{11}{9} \cdot \left\lceil \frac{Edges_{NFB}}{maxEdges} \right\rceil\right) \cdot maxEdges - |E| \tag{2}$$

Proof The lower bound of the expected number of buckets from Theorem 3 multiplied with *maxEdges* yields the total number of edges stored in the buckets. Subtracting from this number the real number of edges yields the lower bound of expected dummy edges. The analogous argument applies to the upper bound. □

The NBM helps us to predict query performance. Depending on the type of queries, the query workload is distributed between the client, and the server, e.g., with neighbor queries, the client has to filter possible false positives. Lemma 6 gives the number of buckets that do not generate false positives because they are full and store edges belonging to the same node. We obtain the percentage of buckets that produce false positives by comparing $Bucket_{Full}^{ini}$ to the expected number of buckets from Theorem 3. Buckets that contain false positives result in more work at the client. A low percentage of full buckets increases the average query processing effort at the client. Note that the number of full buckets does not only depend on the characteristics of the given graph, e.g., distribution of the number of edges per node, but also on parameter *maxEdges*. With respect to adjacency queries, the client does not perform any work. The query performance at the server is affected by dummy edges. Preliminary experiments of ours show that more dummy edges increase the query execution time at the server proportionally. Furthermore, the server has to perform a lookup in the set of buckets to answer queries. Then a large number of buckets also affects the query performance at the server.

All the properties about our algorithm that affect query performance, like the ones discussed in the previous paragraph, are summarized in our query-cost model, Sect. 6.3.

6.3 Query-cost model

Given a query Q, let $SR_{\text{Q-G}}$ and $CR_{\text{Q-G}}$ be the runtime complexity of Q with the original graph G, without index, at the server and the client respectively and $SR_{\text{Q-B}}$ and $CR_{\text{Q-B}}$ the runtime complexity of Q with the bucketization structure B, without index, at the server and the client respectively.

Definition 39 The **query performance ratio** of a given query Q, an original graph G and its corresponding bucketization structure B at the server side is $SP_Q = \frac{SR_{\text{Q-B}}}{SR_{\text{Q-G}}}$, and the query performance ratio at the client side is $CP_Q = \frac{CR_{\text{Q-B}}}{CR_{\text{Q-G}}}$.

We start by analyzing the processing of neighbor queries, followed by adjacency queries. We focus on the case without any index structure either on the original graph G or on the bucketization structure B. Then a single lookup of an edge in

the original graph G has a complexity of $\mathcal{O}(|E|)$. In the bucketization structure, a single lookup of a node in the *index information* has complexity of $\mathcal{O}(|V|)$, and a single lookup of a bucket in the *set of buckets* has a complexity of $\mathcal{O}(E_B)$, where E_B is the number of buckets. The encryption and decryption complexity depends on the security parameter of the underlying encryption scheme [34]. For instance, in the case of AES, the encryption and decryption complexity is $\mathcal{O}(m)$ for variable sized blocks, where m is the number of blocks used which itself depends on the length of the secret key [4, 33]. In our approach, we use two types of encryption schemes, deterministic and probabilistic. Each encryption scheme has a secret key with constant size. Then we consider two different security parameters, one for each type of encryption, ds for deterministic encryption and ps for probabilistic encryption. The complexity of deterministic encryption and decryption is $\mathcal{O}(ds)$, and the complexity of the probabilistic one is $\mathcal{O}(ps)$.

Lemma 8 *Let a Graph $G = (V, E)$, its bucketization structure B and a neighbor query $Q_{Neigbhor}$ (G, u) be given. The server-side and the client-side performance ratio are* $SP_{Q_{Neighbor}(G,u)} = \frac{\mathcal{O}\left(\left\lceil \frac{deg(u)}{maxEdges} \right\rceil \cdot E_B\right)}{\mathcal{O}(|E|)}$ *and* $CP_{Q_{Neigbhor}(G,u)} = \mathcal{O}\left(\left\lceil \frac{deg(u)}{maxEdges} \right\rceil \cdot maxEdges\right)$, *respectively.*

Proof In the original graph, we need to access the edges E, which are stored at the server and retrieve all edges that belong to Node u. Then the effort of executing a neighbor query on the server side is $SR_{Q_{Neighbor}(G,u)} = \mathcal{O}(|E|)$. At the client, no work is necessary. With our bucketization in turn, the following steps are required:

1. Encrypt Node u for querying. The effort is $\mathcal{O}(ds)$.
2. Retrieve the set of *bucketIDs* of Node u from the *index information*. This step has a complexity of $\mathcal{O}(|V|)$.
3. Decrypt the set of *bucketIDs*. The effort is $\mathcal{O}(ps)$.
4. For each *bucketID*, one lookup in the *set of buckets* S_B is required. The number of buckets of u is $|EB_u| = \left\lceil \frac{deg(u)}{maxEdges} \right\rceil$. The complexity of this step is $\mathcal{O}\left(\left\lceil \frac{deg(u)}{maxEdges} \right\rceil \cdot \left(E_B\right)\right)$.
5. Decrypt and filter the $\left\lceil \frac{deg(u)}{maxEdges} \right\rceil \cdot maxEdges$ edges. The decryption and filtering is in $\mathcal{O}\left(\left\lceil \frac{deg(u)}{maxEdges} \right\rceil \cdot maxEdges\right)$.

The server performs Steps 2 and 4, the client Steps 1, 3 and 5. The step with the highest complexity at the client is Step 5, and at the server it is Step 4. Consequently, the effort for executing a neighbor query at the server and at the client is:

$$SR_{Q_{Neighbor}(G,u)-B} = \mathcal{O}\left(\left\lceil \frac{deg(u)}{maxEdges} \right\rceil \cdot \left(E_B\right)\right)$$

$$CR_{Q_{Neighbor}(G,u)-B} = \mathcal{O}\left(\left\lceil \frac{deg(u)}{maxEdges} \right\rceil \cdot maxEdges\right)$$

Finally,

$$SP_{Q_{Neighbor}(G,u)} = \frac{\mathcal{O}\left(\left\lceil \frac{deg(u)}{maxEdges} \right\rceil \cdot \left(E_B\right)\right)}{\mathcal{O}(|E|)}$$

$$CP_{Q_{Neighbor}(G,u)} = \mathcal{O}\left(\left\lceil \frac{deg(u)}{maxEdges} \right\rceil \cdot maxEdges\right)$$

□

Lemma 9 *Let a Graph $G = (V, E)$, its bucketization structure B and an adjacency query $Q_{Adjacency}(G, u, v)$ be given. The server-side and client-side performance ratio are $SP_{Q_{Adjacency}(G,u,v)} = \frac{\mathcal{O}(|E|+|D_E|)}{\mathcal{O}(|E|)}$ and $CP_{Q_{Adjacency}(G,u,v)} = \mathcal{O}(ds)$, where $|D_E|$ is the number of dummy edges.*

Proof In the original graph, in order to check whether Edge (u, v) exists in E, it is necessary to execute one lookup on the edges E. Then the effort of executing an adjacency query at the server is $SR_{Q_{Adjacency}(G,u,v)} = \mathcal{O}(|E|)$. At the client, no work is necessary. On the transformed graph, the following steps are required:

1. Encrypt Edge (u, v) for querying. The effort is $\mathcal{O}(ds)$.
2. Execute one lookup in the encrypted edges, which are stored in the *set of buckets*. The complexity of this step is $\mathcal{O}(|E| + |D_E|)$.

Step 1 takes place at the client, it is an encryption operation in $\mathcal{O}(ds)$. So the ratio at the client is $CP_{Q_{Adjacency}(G,u,v)} = \mathcal{O}(ds)$. At the server, the effort is $SR_{Q_{Adjacency}(G,u,v)-B} = \mathcal{O}\left(|E| + |D_E|\right)$. Then $SP_{Q_{Adjacency}(G,u,v)} = \frac{\mathcal{O}(|E|+|D_E|)}{\mathcal{O}(|E|)}$. □

From the Query-Cost Model, we can learn that for adjacency and neighbor queries the parameter *maxEdges* plays an important role regarding the query-execution effort at client and server. If *maxEdges* increases, the number of dummy edges increases, and the server must take more effort to answer queries. At the client, to answer neighbor queries if *maxEdges* increases, the workload at the client increases as well. This is because the client has to filter more false positives. Therefore, we suggest for scale-free networks that the parameter *maxEdges* should take smaller values, exactly m. In the next section, we demonstrate by means of experiments how this parameter affects the performance of our bucketization algorithm.

7 Experiments

In this section, we present experiments to evaluate (1) the accuracy of our NBM and (2) the performance of our bucketization approach. Notice that the query-cost model is derived from the NBM. Consequently, the accuracy of this model mainly depends on the accuracy of the NBM.

7.1 Experiment setup

7.1.1 Input datasets

In our experiments, we use synthetic and real datasets.

Synthetic datasets We have used Networkx [36] to generate 8 different undirected graphs that follow the BA Model. Table 2 shows the characteristics of the generated graphs, where $|V|$ is the number of nodes, m the growth parameter and E the number of edges. Related experimental studies on secrecy preserving on graph-structured data have considered synthetic graphs with nodes between 3000 and 25,000 [42, 44]. We vary the number of nodes from 5000 to 150,000. For graphs with 5000 and 10,000 nodes, we set the growth parameter m equal to 6 and 8. For graphs with 40,000 and 1,50,000 nodes, we set the growth parameter m equal to 8 and 10. The number of edges of each graph depends on the number of nodes and the growth parameter m.

Real datasets Our bucketization approach can work with any graph. As real datasets, we have chosen two scale-free networks, the Actor network [23] and the Web network [26], and one non-scale-free network, the Citation network [25]. Barabási and Albert [2] have proven that the Actor and the Web network are scale-free. The Actor network contains 1,048,575 edges and 1,137,725 nodes, 89,150 nodes represent actors, and 1,048,575 nodes represent movies. An edge connects a movie with an actor who has played in it. The Actor network exhibits the preferential attachment feature. Namely, if an actor has played in more movies, a casting director is more familiar with his or her skills. Then an actor with a higher degree has higher chances to be considered for a new role. The growth parameter m of the Actor network is 4. The Web network contains 2,381,903 nodes and 2,312,497 edges, and its growth parameter m is 5. The nodes in the Web network are web pages, and the edges represent hyperlinks between them. The Citation network contains 27,770 nodes and 3,52,807 edges. The nodes in the Citation network are articles, an edge is created between articles a and b if article a cites b.

Table 2 Characteristics of the synthetic data

Synthetic data	$\|V\|$	m	E
$G1$	5000	6	29,964
$G2$	5000	8	39,936
$G3$	10,000	6	59,964
$G4$	10,000	8	79,936
$G5$	40,000	8	3,19,936
$G6$	40,000	10	3,99,900
$G7$	1,50,000	8	1,199,936
$G8$	1,50,000	10	1,499,900

7.1.2 Queries

Based on initial experiments and the Query-Cost Model from Sect. 6.3, we observe that node degree plays an important role in the query performance evaluation. Therefore, the objects that will be part of an experiment sample, i.e., the queries, should be carefully selected to have a representative sample of queries. In all experiments that follow, the experiment sample will consist of actual nodes from the graph that is being queried. In the following, we present the experiment results for these type of queries. In the context of neighbor queries, there are two kinds of nodes, hubs, and non-hubs, with very different query performance. So, to have equally represented hubs and non-hubs in our query sample, we divide neighbor queries into two groups: $RandomQ_{Neighbor}(G, u)$ and $HubQ_{Neighbor}(G, u)$. For the group of queries $RandomQ_{Neighbor}(G, u)$, we select the input node u randomly from the set of nodes V without considering the hubs in the graph. For $HubQ_{Neighbor}(G, u)$, we identify the hubs in the graph and use them as input. For adjacency queries, $Q_{Adjacency}(G, u, v)$, the nodes u, v are selected randomly from the set of nodes V. The execution time of adjacency queries depends on the total number of edges, including dummy edges (Sect. 6.3). So a distinct consideration of hubs is not necessary in this case.

7.1.3 Evaluation measures

We use seven metrics which let us evaluate the accuracy of the number-of-buckets model (NBM) and the performance of the bucketization approach.

The NBM metrics are:

- $E_{TotalEB}$: This metric quantifies the number of buckets obtained when applying our bucketization algorithm to graph G.
- E_{dummy}: This is the percentage of dummy edges when applying our bucketization algorithm to Graph G.
- $E_{Buckets^{ini}_{Full}}$: This is the percentage of buckets that are full after the initialization of the bucketization algorithm on graph G.

The bucketization performance metrics are:

- $P_{SQprocessing}$: This metric quantifies the server-query-processing time when using our bucketization, i.e., the time required by the server in order to answer a query sent by the client.
- $P_{CQprocessing}$: This metric quantifies the client query processing time when using our bucketization, i.e., the time required by the client to decrypt the results returned from the server and filter false positives.
- $P_{TQprocessing}$: This metric quantifies the total query processing time using our bucketization structure, i.e., it adds up the processing time at the client and at the server.

– $P_{RQprocessing}$: This is the ratio of the total query processing time using our bucketization structure to that using the original graph G.

7.2 Results

We now present the results of the experiments. First, we discuss the evaluation of the NBM is discussed, then the performance. Our experiments evaluate the tradeoff between secrecy and performance. We study the effect of each parameter from Sect. 7.1.3 one by one. The NBM Model has been developed based on the characteristics of scale-free networks. Therefore for the NBM evaluation, we use only the scale-free datasets, namely all synthetic datasets, $G1, \ldots, G8$, and the two scale-free real datasets, the Actor and Web networks. For the performance evaluation, we use all the real datasets. In the experiments, we have varied the parameter *maxEdges*.

7.2.1 NBM evaluation

$E_{TotalEB}$: Fig. 6 shows the numbers of buckets obtained with the synthetic and real datasets. For both types of datasets, we have used different values for the parameter *maxEdge*. The markers on each bar of both figures are the lower and upper bounds calculated with our NBM. For all experiments, the number of buckets obtained is always inside the range calculated with Theorem 3. Figure 6 also shows that, if $maxEdges \leq m$, the number of buckets obtained is between the lower bound and the middle of the range given by the NBM. If $maxEdges > m$, the number of buckets gets closer to the upper bound of the NBM. We explain this effect as follows: In scale-free networks, most of the nodes in the graph have degree equal to m. If the parameter *maxEdge* is set to m, most buckets will have reached their maximal capacity after the initialization phase, and fewer buckets will be considered for merging. Then the total number of buckets gets closer to the optimal solution of our algorithm, which is the lower bound of our estimation. If the parameter *maxEdges* is set to values greater than m, after the initialization phase most of the buckets will not

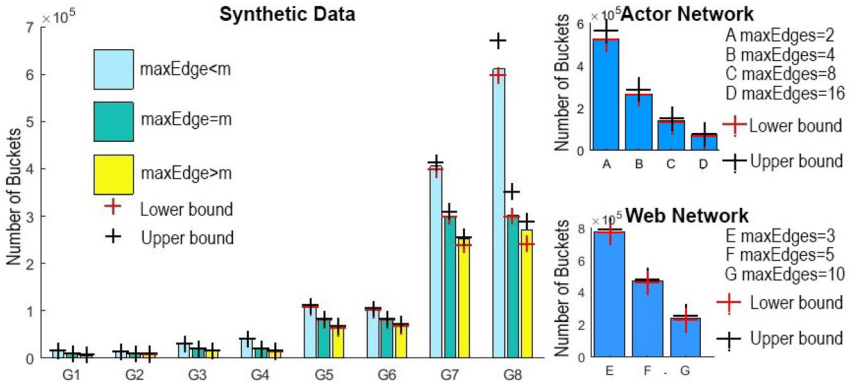

Fig. 6 $E_{TotalEB}$ obtained for the synthetic and real datasets

Table 3 Percentage of E_{dummy} for the synthetic datasets

maxEdges	% of E_{dummy}
$1 < maxEdges < m$	1.217
$maxEdges = m$	0.889
$maxEdges > m$	26.513

Table 4 Percentage of E_{dummy} for the real datasets

Dataset	*maxEdges*	% of E_{dummy}
Actor network	2	0.428
	4	0.748
	16	7.629
Web network	3	0.223
	5	1.112
	10	6.349

have reached their maximal capacity, and they will be considered for the merging. In the merging phase, because of the heuristic used, it is not always possible to reach an optimal solution. Therefore, the number of buckets obtained gets closer to the upper bound of the NBM.

E_{dummy}: We calculate the percentage of dummy edges in comparison with the size of the original graph for the synthetic data and the real datasets. Table 3 shows the average percentage of dummy edges for the synthetic data, i.e., eight datasets. Table 4 shows the exact percentage of dummy edges for the real datasets. Both tables show that the number of dummy edges needed increases, as parameter *maxEdges* takes values greater than m. More dummy edges mean a larger database. This is likely to affect the efficiency of the querying process on the server as well, and this will be examined in Sect. 7.2.2.

$E_{Buckets^{ini}_{Full}}$: Table 5 shows the average percentage of full buckets after the initialization phase for the synthetic data. For the real datasets the exact percentage is given, see Table 6. The number of full buckets decreases, as the parameter *maxEdges* takes values greater than m. If a bucket stores edges that belong to different nodes and/or dummy edges, the client will have to do more work. Full buckets after the initialization phase contain edges that belong to a single node. More full buckets right after initialization let our algorithm to come closer to the optimal solution. An optimal solution implies fewer buckets for the merging process, fewer dummy edges, and fewer false positives when querying.

Table 5 Percentage of $E_{Buckets_{Full-ini}}$ for the synthetic datasets

maxEdges	% of $E_{Buckets_{Full-ini}}$
$1 < maxEdges < m$	88.79
$maxEdges = m$	86.81
$maxEdges > m$	46.65

Table 6 Percentage of $E_{Buckets^{ini}_{Full}}$ for the real datasets

Dataset	*maxEdges*	% of $E_{Buckets^{ini}_{Full}}$
Actor Network	2	59.15
	4	55.78
	16	14.49
Web Network	3	81.38
	5	80.18
	10	47.96

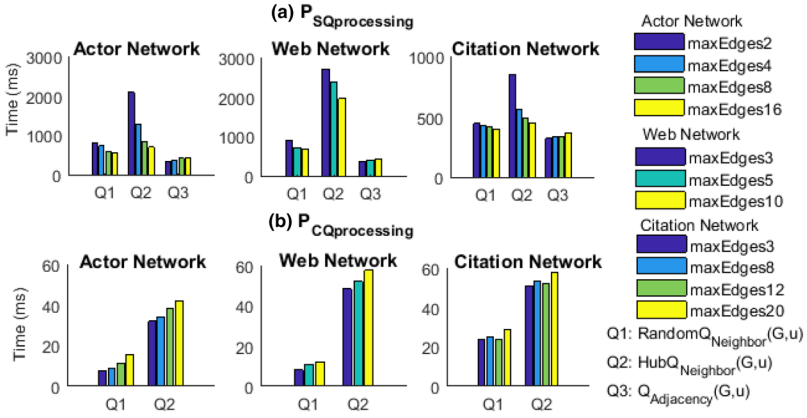

Fig. 7 $P_{SQprocessing}$ and $P_{CQprocessing}$ in the Actor, Web and Citation Networks

7.2.2 Performance evaluation

As in the previous section, we have conducted our experiments with synthetic and real datasets. The results of the experiments with both datasets, i.e., synthetic and real, are very much the same. In what follows, due to its applicability to real-world scenarios, we present the results on the real data.

$P_{SQprocessing}$: Fig. 7a shows the average query-processing time of the server for the three groups of queries for real datasets. For $RandomQ_{Neighbor}(G,u)$ and $HubQ_{Neighbor}(G,u)$, the query-processing time of the server increases as *maxEdges* decreases. This is because, if *maxEdge* decreases, the number of buckets increases, and when executing a neighbor query, the server has to retrieve more buckets. In contrast, the query-processing time of the server for $Q_{Adjacency}(G,u,v)$ increases as *maxEdges* takes higher values. The increase, in this case, is due to the higher number of dummy edges inserted. Our experiments in the previous section show that the number of dummy edges needed grows, as *maxEdges* increases.

$P_{CQprocessing}$: For this part of the evaluation we only consider two type of queries, i.e., $RandomQ_{Neighbor}(G,u)$ and $HubQ_{Neighbor}(G,u)$. We omit adjacency queries because they do not require any post-processing on the client. See Fig. 7b. In the scale-free networks, i.e., the Actor and Web networks, the query processing time

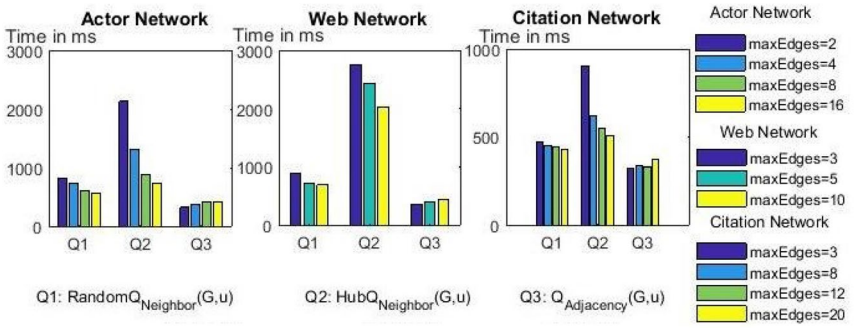

Fig. 8 $P_{TQprocessing}$ in the actor, web and citation networks

at the client increases as *maxEdges* takes larger values. In the Citation network, with $maxEdges = 12$, i.e., the average degree of the network, the client query processing time decreases in comparison with the time obtained with other values that *maxEdges* takes.

$P_{TQprocessing}$: Fig. 8 shows the total average query processing time for the three groups of queries defined in Sect. 7.1.2 in the real datasets. For the queries $RandomQ_{Neighbor}(G,u)$ and $HubQ_{Neighbor}(G,u)$, the total execution time increases as *maxEdges* decreases. This increment is related to the increase of the query-processing time of the server, which we have explained with the metric $P_{SQprocessing}$. For adjacency queries, the query processing time of the server and the total query-processing time are the same. This is because adjacency queries do not require any post-processing on the client, i.e., all the work is done at the server.

We can see from the analysis of $P_{SQprocessing}$, $P_{CQprocessing}$ and $P_{TQprocessing}$ that the best value to set *maxEdges* in scale-free networks is the growth parameter m. In scale-free networks, most nodes have a degree equal to m, so most buckets will be full after initialization. For non-scale-free networks, we observe that the best value to set *maxEdges* is the average degree of the network. For the last experimental results, $P_{RQprocessing}$, we set *maxEdges* to the best option, i.e., $maxEdges = m$ for the scale-free networks and $maxEdges = 12$, average degree, for the non-scale-free network.

$P_{RQprocessing}$: Fig. 9 shows a comparison of the total query processing time for $RandomQ_{Neighbor}(G,u)$, $HubQ_{Neighbor}(G,u)$, and $Q_{Adjacency}(G,u,v)$, using our bucketization structures and the original graphs of the three real datasets. For each diagram in Fig. 9, each pair of points on the x-axis, i.e., 1 and 2, 3 and 4, 5 and 6, corresponds

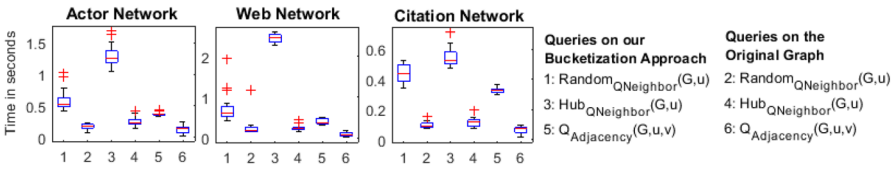

Fig. 9 Total query processing time-real datasets

to one type of query. The first pair corresponds to $RandomQ_{Neighbor}(G, u)$, the second one to $HubQ_{Neighbor}(G, u)$ and the third one to $Q_{Adjacency}(G, u, v)$. In each pair, the first point represents the total query processing time using our bucketization and the second one is the total time using the original graph. We deem the total execution time on the original graph the optimum. So we evaluate our approach depending on how much the query-processing time increases in comparison with the original graph. Regarding the Actor network, $RandomQ_{Neighbor}(G, u)$ with our bucketization approach is on average 3.44 times slower than with the original graph, $HubQ_{Neighbor}(G, u)$ is 5.12 times slower and $Q_{Adjacency}(G, u, v)$ is 2.88 times slower. In the Web graph, $RandomQ_{Neighbor}(G, u)$ with our bucketization approach is 2.90 times slower than with the original graph on average, $HubQ_{Neighbor}(G, u)$ is 10.15 times slower, and $Q_{Adjacency}(G, u, v)$ is 4.76 times slower. In the Citation network, with our bucketization approach $RandomQ_{Neighbor}(G, u)$ is on average 4.51 times slower than with the original graph, $HubQ_{Neighbor}(G, u)$ is 4.78 times slower, and $Q_{Adjacency}(G, u, v)$ is 4.93 times slower. To summarize, in the scale-free networks, i.e., the Actor and Web network, except for $HubQ_{Neighbor}(G, u)$, with our approach the query execution time is 3.5 times slower than with the original graph. In the non-scale-free network, i.e., the Citation network, with our approach the query execution time is approximately 5 times slower than with the original graph. In our opinion, these are reasonable prices for secrecy guarantees. So our bucketization approach is effective and feasible for secrecy for graph-structured data.

8 Conclusions

A core challenge when outsourcing a database is to ensure the secrecy of the data. In this paper, we have studied this problem for graph-structured data. We have proposed a secrecy model for this kind of data based on the concept of indistinguishability. Existing proposals, such as Fan et al. [10], Zhang et al. [43], Wang and Lakshmanan [39], deal with different types of adversaries and different secrecy guarantees. Our secrecy notion guarantees that, given a secretized graph, an adversary cannot learn any information about the original graph beyond the information leakage specified. While a bucketization of the edges gives way to the secrecy envisioned here, as we have shown, finding an optimal bucketization is NP-hard. We have proposed a heuristic that guarantees that the worst bucketization solution will be off by a factor of $\frac{11}{9}$ of the optimal one. Next, to facilitate query planning, we predict the behavior of our algorithm with respect to its parameters and properties of the input graph. In other words, we propose a performance model that allows estimating (1) the number of buckets and (2) the query-processing complexity. Our experiments with both real and synthetic datasets confirm the accuracy of our model and the effectiveness of our approach.

In the future, it will be interesting to study more complex graph queries, which cannot be easily represented using relational databases. Another future research direction is to study and propose secrecy approaches for graph-structured data under stronger secrecy guarantees, including access pattern leakage.

Acknowledgements Open Access funding provided by Projekt DEAL. The first author thanks "Escuela Politécnica Nacional, Ecuador—Departamento de Informática y Ciencias de la Computación" for its support. This work was partially funded by the German Research Foundation (DFG) as part of the research Datenschutzkonforme Verwaltung relationaler Datenbestände (DFG; ref. nb BO 2129/13-1).

Appendix: Hardness result

We start by introducing the BP problem.

Definition 40 Let a set of n bins $C = \{c_1, c_2, \ldots, c_n\}$ and the same number of n items $I = \{a_1, a_2, \ldots, a_n\}$ be given. All bins have equal capacity w_c, and the weight of each item $a_i \in I, w_{a_i}$, is smaller than or equal to the capacity w_c. The **Bin-packing problem** is finding a mapping $BP : I \rightarrow C$ of each item in I to one bin in C such that the following Constraints $bp1$, $bp2$ and $bp3$ are met.

(bp_1) An item is assigned to only one bin.
(bp_2) The sum of the weights of all items assigned to a bin does not exceed the bin capacity w_c. Formally, $\forall c_j \in C : W_{c_j} \leq w_c$ where $W_{c_j} = \sum_{a_i \in \{a \in I | BP(a) = c_j\}} w_{a_i}$.
(bp_3) The number m of bins used is as small as possible, i.e., minimize

$$m = \left| \bigcup_{a_i \in I} \{BP(a_i)\} \right|.$$

For the hardness proof, we use a restricted version of the BP-Problem, Definition 40, where we restrict the weight of the items to be polynomial in n. Since the BP problem is strongly NP-complete, bounding any of its numerical parameters by a polynomial in the length of the input, the resulting problem remains NP-complete [11].

We introduce Lemmas 10 and 11, which are used in the hardness proof. These two lemmas help us (1) to show that an instance of the BP problem, called *initial BP*, can be reduced in polynomial time to an instance of the bucketization problem, called *transformed BP*, and (2) to prove that a given solution of the *transformed BP* can be transformed to a solution of the *initial BP* in polynomial time. In what follows, we identify the steps required to construct the *transformed BP*.

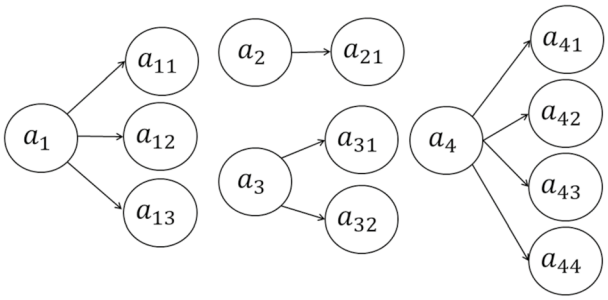

Fig. 10 *Transformed BP* of Example 9

Transformed BP construction Given a BP problem with a set of items I, the *transformed BP* is constructed as follows:

- For each item $a_i \in I$, create the set of nodes $V_i = \{a_i, a_{i1}, a_{i2}, \ldots, a_{iw_{a_i}}\}$ and the set of edges $E_i = \{(a_i, a_{i1}), (a_i, a_{i2}), \ldots, (a_i, a_{iw_{a_i}})\}$.
- The graph that represents the *transformed BP* is $G = (\cup_{i=1}^{n} V_i, \cup_{i=1}^{n} E_i)$.

Lemma 10 *Given an initial BP, the transformed BP can be constructed in polynomial time.*

Proof For each item $a_i \in I$, in order to build *transformed BP*, we need $(w_{a_i} + 1)$ nodes and w_{a_i} edges. Altogether this requires $\sum_{i=1}^{n}(w_{a_i} + 1)$ steps. Since in the restricted version of the BP problem, the weight of the items are polynomial in n, the construction is still polynomial in n. Then an *initial BP* can be transformed to a *transformed BP* in polynomial time. □

Example 9 illustrates the construction of the *transformed BP*.

Example 9 Consider the *initial BP* with set of items $I = \{a_1, a_2, a_3, a_4\}$ with weights $w_{a_1} = 3, w_{a_2} = 1, w_{a_3} = 2, w_{a_4} = 4$ and the set C of bins with capacity $w_c = 5$. Figure 10 shows the corresponding *transformed BP*.

Once we have built *transformed BP*, we can run an algorithm that solves the bucketization problem on it, by setting *maxEdges* to w_c. The solution of the *transformed BP* is a bucketization B that fulfills Constraints c1-c4. The following set of buckets S_B^G, which is part of the bucketization structure B, is a possible solution of the *transformed BP* of Example 9:

$$S_B^G = \left\{ \begin{array}{l} b_1 : (a_1, a_{11}), (a_1, a_{12}), (a_1, a_{13}), (a_3, a_{31}), (a_3, a_{32}) \\ b_2 : (a_2, a_{21}), (a_4, a_{41}), (a_4, a_{42}), (a_4, a_{43}), (a_4, a_{44}) \end{array} \right\}$$

where b_1 and b_2 are the *bucketIDs*. Since we set $maxEdges = w_c$, it holds for all buckets $b \in S_B^G$ that $freq(b) \leq w_c$.

The next lemma, Lemma 11, states that a solution of the *initialBP* can be constructed in polynomial time from a solution of the *transformed BP*. Before moving to Lemma 11, we first explain the solution construction process.

Initial BP solution construction A solution of the *initial BP* can be constructed from a solution of the *transformed BP* as follows:

- Select the bins $c_j, \ldots, c_m$ needed to store the items in I, where $m = |S_B^G|$.
- Map each item $a_i \in I$ to one bin c_j, where $j \in \{1, \ldots, m\}$, as follows: $\forall b_j \in S_B^G : I_{c_j} = \{a_i \mid \exists y \in \{1, \ldots, w_{a_i}\}, (a_i, a_{iy}) \in b_j\}$. Then $\forall a_i \in I_{c_j} : a_i \rightarrow c_j$.

The following shows the solution constructed for the *initial BP* from the *transformed BP* of Example 9:

$$m = \left|S_B^G\right| = 2$$
$$I_{c_1} = \{a_1, a_3\}, I_{c_2} = \{a_2, a_4\}$$
$$a_1 \rightarrow c_1$$
$$a_3 \rightarrow c_1$$
$$a_2 \rightarrow c_2$$
$$a_4 \rightarrow c_2$$

Lemma 11 *A solution of the transformed BP can be transformed to a solution of the corresponding initial BP in polynomial time.*

Proof Consider a bucketization of *transformed BP* that fulfills Constraints $c_1 - c_4$. We transform it to a BP solution with the *solution construction process*. Now we proceed to demonstrate that the transformed solution fulfills the constraints of the BP problem, bp1 to bp3, with respect to the *initial BP* problem. We start by analyzing the constraints of the BP problem and of the bucketization problem. First, Constraint bp1 is fulfilled because of Constraints c_1 and c_3 of the bucketization problem. Constraint c_1 ensures that each edge is assigned to only one bucket. Then $\forall i \neq j : c_i \cap c_j = \emptyset$. Together with the fact that for all items $a_i \in I, w_{a_i} \leq maxEdges$, Constraint $c3$ ensures that the edges belonging to the same node are placed in the same bucket.

Second, Constraint bp2 is fulfilled because of Constraint c_2 of the bucketization problem. For all bins $c_j \in C, W_{c_j} = freq(b_j)$ and $maxEdges = w_c$, then $freq(b_j) \leq w_c$, which fulfills Constraint bp1. Third, bp3 is fulfilled because of Constraint c_4. The number of buckets is the number of bins used in the *initial BP* solution. Then minimizing the buckets is the same as minimizing the number of bins used.

Finally, a bucketization solution of a *transformed BP* can be transformed to a solution of the *initial BP* in polynomial time. The reconstruction requires one lookup in all the elements of each bucket $b_i \in S_B^G$. Then the complexity of the reconstruction is $\mathcal{O}(m)$, where m is the total number of elements, i.e., edges, stored in S_B^G. Since S_B^G stores the set of items I, $m = \sum_{i=1}^{n} w_{a_i}$.

□

Theorem 4 *Finding an optimal bucketization that meets Constraints c_1−c_4 is NP-hard.*

Proof With Lemmas 10 and 11 we have shown that an instance of the BP problem can be reduced to an instance of the bucketization problem in polynomial time. Since the BP problem is NP-hard [19], the optimal bucketization problem is NP-hard as well.

□

References

1. Aggarwal, G., Bawa, M., Ganesan, P., Garcia-Molina, H., Kenthapadi, K., Motwani, R., Srivastava, U., Thomas, D., Xu, Y.: Two can keep a secret: a distributed architecture for secure database services. CIDR (2005)
2. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
3. Bellare, M., Boldyreva, A., O'Neill, A.: Deterministic and efficiently searchable encryption. In: Annual International Cryptology Conference, pp. 535–552. Springer, New York (2007)
4. Bhandari, A., Gupta, A., Das, D.: A framework for data security and storage in cloud computing. In: 2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT). IEEE, pp. 1–7 (2016)
5. Boneh, D., Shoup, V.: A graduate course in applied cryptography. (2008) https://crypto.stanford.edu/~dabo/cryptobook/
6. Bösch, C., Brinkman, R., Hartel, P.H., Jonker, W.: Conjunctive wildcard search over encrypted data. Secur. Data Manag. **6933**, 114–127 (2011)
7. Cao, J., Rao, F.Y., Kuzu, M., Bertino, E., Kantarcioglu, M.: Efficient tree pattern queries on encrypted xml documents. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops. ACM, pp. 111–120 (2013)
8. Cao, N., Yang, Z., Wang, C., Ren, K., Lou, W.: Privacy-preserving query over encrypted graph-structured data in cloud computing. In: 2011 31st International Conference on Distributed Computing Systems (ICDCS). IEEE, pp. 393–402 (2011)
9. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. J. Comput. Secur. **19**(5), 895–934 (2011)
10. Fan, Z., Choi, B., Chen, Q., Xu, J., Hu, H., Bhowmick, S.S.: Structure-preserving subgraph query services. IEEE Trans. Knowl. Data Eng. **27**(8), 2275–2290 (2015)
11. Garey, M.R., Johnson, D.S.: "Strong" np-completeness results: motivation, examples, and implications. J. ACM **25**(3), 499–508 (1978)
12. Garey, M.R., Johnson, D.S.: A Guide to the Theory of np-Completeness, p. 70. WH Freemann, New York (1979)
13. Getoor, L., Taskar, B., Koller, D.: Selectivity estimation using probabilistic models. In: ACM SIGMOD Record. ACM, vol. 30, pp. 461–472 (2001)
14. Goh, E.J., et al.: Secure indexes. IACR Cryptol. ePrint Arch. **2003**, 216 (2003)

15. Hacigümüş, H., Iyer, B., Li, C., Mehrotra, S.: Executing sql over encrypted data in the database-service-provider model. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. ACM, pp. 216–227 (2002)
16. Hacıgümüş, H., Iyer, B., Mehrotra, S.: Query optimization in encrypted database systems. In: International Conference on Database Systems for Advanced Applications pp. 43–55. Springer, New York (2005)
17. He, X., Vaidya, J., Shafiq, B., Adam, N., Lin, X.: Reachability analysis in privacy-preserving perturbed graphs. In: 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT). IEEE, vol. 1, pp. 691–694 (2010)
18. Hore, B., Mehrotra, S., Tsudik, G.: A privacy-preserving index for range queries. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB Endowment, vol. 30, pp. 720–731 (2004)
19. Johnson, D.S.: Near-optimal bin packing algorithms. PhD thesis, Massachusetts Institute of Technology (1973)
20. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. ACM, pp. 965–976 (2012)
21. Katz, J., Lindell, Y.: Introduction to modern cryptography (2007)
22. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic attacks on secure outsourced databases. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 1329–1340 (2016)
23. Kunegis, J.: Konect: the koblenz network collection. In: Proceedings of the 22nd International Conference on World Wide Web. ACM, pp. 1343–1350 (2013)
24. LeFevre, K., DeWitt, D.J., Ramakrishnan, R.: Mondrian multidimensional k-anonymity. In: 22nd International Conference on Data Engineering (ICDE'06). IEEE, pp. 25–25 (2006)
25. Leskovec, J., Krevl, A.: Snap datasets: stanford large network dataset collection. http://snaps tanfordedu/data/cit-HepThhtml (2014)
26. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters. Internet Math. **6**(1), 29–123 (2009)
27. Li, J., Wang, Q., Wang, C., Cao, N., Ren, K., Lou, W.: Fuzzy keyword search over encrypted data in cloud computing. In: 2010 Proceedings IEEE INFOCOM. IEEE, pp. 1–5 (2010)
28. Maserrat, H., Pei, J.: Neighbor query friendly compression of social networks. In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 533–542 (2010)
29. Matias, Y., Vitter, J.S., Wang, M.: Wavelet-based histograms for selectivity estimation. In: ACM SIGMoD Record. ACM, vol. 27, pp. 448–459 (1998)
30. Meng, X., Kamara, S., Nissim, K., Kollios, G.: Grecs: graph encryption for approximate shortest distance queries. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 504–517 (2015)
31. Muthukrishnan, S., Poosala, V., Suel, T.: On rectangular partitionings in two dimensions: algorithms, complexity and applications. In: International Conference on Database Theory, pp. 236–256. Springer, New York (1999)
32. Naveed, M., Kamara, S., Wright, C.V.: Inference attacks on property-preserving encrypted databases. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 644–655 (2015)
33. Orhanou, G., El Hajji, S., Bentaleb, Y.: Eps aes-based confidentiality and integrity algorithms: complexity study. In: 2011 International Conference on Multimedia Computing and Systems (ICMCS). IEEE, pp. 1–4 (2011)
34. Prakash, A.J., Uthariaraj, V.R.: Multicrypt: A provably secure encryption scheme for multicast communication. In: First International Conference on Networks and Communications, (NETCOM'09). IEEE, pp. 246–253 (2009)
35. Reddy, N., Haritsa, J.R.: Analyzing plan diagrams of database query optimizers. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB Endowment, pp. 1228–1239 (2005)
36. Schult, D.A., Swart, P.: Exploring network structure, dynamics, and function using networkx. In: Proceedings of the 7th Python in Science Conferences (SciPy 2008), vol. 2008, pp. 11–16 (2008)

37. Syalim, A., Nishide, T., Sakurai, K.: Preserving integrity and confidentiality of a directed acyclic graph model of provenance. In: Data and Applications Security and Privacy, vol. XXIV, pp. 311–318 (2010)
38. Van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P., Jonker, W.: Computationally efficient searchable symmetric encryption. In: Workshop on Secure Data Management, pp. 87–100. Springer, New York (2010)
39. Wang, H., Lakshmanan, L.V.: Efficient secure query evaluation over encrypted xml databases. In: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB Endowment, pp. 127–138 (2006)
40. Wang, J., Du, X.: A secure multi-dimensional partition based index in das. In: Asia-Pacific Web Conference, pp. 319–330. Springer, New York (2008)
41. Wang, Q., Ren, K., Du, M., Li, Q., Mohaisen, A.: Secgdb: Graph encryption for exact shortest distance queries with efficient updates. In: International Conference on Financial Cryptography and Data Security, pp. 79–97. Springer, New York (2017)
42. Yi, P., Fan, Z., Yin, S.: Privacy-preserving reachability query services for sparse graphs. In: 2014 IEEE 30th International Conference on Data Engineering Workshops (ICDEW). IEEE, pp. 32–35 (2014)
43. Zhang, Y., Su, S., Wang, Y., Chen, W., Yang, F.: Privacy-assured substructure similarity query over encrypted graph-structured data in cloud. Secur. Commun. Netw. **7**(11), 1933–1944 (2014)
44. Zhou, B., Pei, J.: Preserving privacy in social networks against neighborhood attacks. In: IEEE 24th International Conference on Data Engineering, (ICDE 2008). IEEE, pp. 506–515 (2008)