Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2017,5
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

On Secrecy and Performance Models for Query Processing on Outsourced Graph Data

Gabriela Suntaxi, Aboubakr Achraf El Ghazi, Klemens Böhm

2017

On Secrecy and Performance Models for Query Processing on Outsourced Graph Data

Gabriela Suntaxi
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
gabriela.suntaxi@kit.edu

Aboubakr Achraf El Ghazi
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
elghazi@kit.edu

Klemens Böhm
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
klemens.boehm@kit.edu

ABSTRACT

Database outsourcing is a challenging task 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 concept of indistinguishability. The notion ensures that an adversary can learn the edges existing between the nodes only with negligible probability. 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 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 (1) the accuracy of our number of buckets model for scale-free networks and (2) the efficiency of our approach with respect to query processing.

1. INTRODUCTION

Outsourcing databases to a third-party service provider has become ubiquitous. While economic and organizational advantages are obvious, database outsourcing remains challenging concerning data secrecy. Databases contain sensitive information that needs to be protected against adversaries, including the service provider. If an unauthorized user accesses the data, she should not be able to learn anything.

A broad range of real-world datasets exhibits a graph structure. Furthermore, many real graphs such as the email network or the World Wide Web follow a scale-free power-law distribution [3]. At the same time, these graphs often contain sensitive information.

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). 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 secrecy guarantees that are provable. At the same, the approaches should not do away with the advantages of database outsourcing, and query processing in particular 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 and (3) predict the impact of parameter changes.

So there are two requirements on a secure storage scheme for graph-structured data. **R1:** An adversary, including the service provider, must not be able to learn the label of nodes or their degree. 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. To quantify this, a performance model is needed.

The first requirement calls for a rigid definition of secrecy. This includes specifying the assumed knowledge of the adversary and the security property, i.e., a description of what constitutes a breach of the scheme. We consider adversaries with knowledge of (i) the algorithm used to secretize the graph G, (ii) the labels of all the nodes in G and (iii) the multiset that contains the degree of each node in G, without stating the correspondence between nodes and degrees. The secrecy property ensures that, given a secretized graph, an adversary cannot say with non-negligible probability if the secretized graph corresponds to the original graph G.

Since existing secrecy notions are different from this, we propose a new one, i.e., formalize the notion just sketched. Related secrecy notions deal with different types of adversaries. Notions such as [7], [23] offer guarantees against chosen plaintext attacks. In our scenario, these guarantees are not enough. This is because the edges of the graph can also reveal information. Wang et al. [22] define a secrecy notion for XML documents. It is based on the definition of perfect secrecy. As an XML document has a tree structure, they assume that an adversary knows the domain values of the data and the distribution on the leaf nodes. We additionally assume that the adversary knows the degree distribution of the complete graph G.

Secure database storage has been widely studied. However, existing techniques such as [19], [1] either cannot be applied to graph-structured data, or they do not cover both requirements R1 and R2. Approaches for graph-structured data such as [19], [18] do not keep the structure of the graph. Then they cannot answer certain queries, such as neighbor and adjacency queries. Other approaches like [1], [10], [2] could exhibit unwanted behavior when being adapted to graph-structured data, e.g., leak information, see Section 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. 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 [17]. Then in what follows we describe the specifics, such as division of work between client and server, for these queries.

Summing up, our contributions are as follows: First, we propose a secrecy notion for graph-structured data based on indistinguishability [13]. 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. Our approach partitions edges into buckets. In order to answer queries, we store index information. It contains, next to other information, the labels of the nodes. We show that finding an optimal bucketization is NP-hard. Consequently, we propose a heuristic, which we also evaluate empirically 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 graphs that are scale-free. Our performance model consists of (1) a number-of-buckets model, which estimates the number of buckets obtained after 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.

Secrecy notions: Adaptive semantic secrecy is proposed in [6] and [5]. This concept is adapted for answering approximate shortest distance queries in graphs in [18]. Their notion uses leakage functions, i.e., information revealed to the server. These approaches assume that the adversaries only have access to information that has leaked, but not to any other sources. We consider adversaries with additional information on the original graph G, i.e., the labels of all nodes of G and the multiset that contains the degree of each node in G.

Bucketization on relational databases: Data secrecy in relational databases has been investigated extensively [10], [2], [9]. Several approaches are based on bucketization. In this context, bucketization (1) encrypts each tuple in an original relation as one string, (2) groups the tuples in partitions, each partition represents a bucket, and (3) stores index values. 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.

- *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, see Example 2.1.

  Example 2.1: *Consider a graph with edges E={(A,B), (B,C), (C,A)}. 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 2.1 shows the secretized relation. The partitions are* $[b1, \{A\}], [b2, \{B\}], [b3, \{C\}]$. *The relationship between the index values (b1,b2), (b2,b3) and (b3,b1) share the same structure as the original edges E.*

Table 2.1. Secretized relation of Example 2.1

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

- *Multidimensional bucketization*. With this option, the domain of each attribute is partitioned individually. Given an optimal multidimensional bucketization, this bucketization can be secure. However, the effort of finding an optimal multidimensional bucketization with respect to query performance is NP-hard [15]. Nevertheless, this NP-hard problem can be solved with heuristics such as in [15] and [21]. But, these solutions do not consider certain graphs characteristics such as the distribution of edges per node or 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.

Secure storage for graph-structured data: An approach for finding the shortest path between two nodes in a directed graph is presented in [12]. 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, queries results can only be approximate. As XML documents are a specific kind of graph, we briefly turn to this research direction as well. The approaches in [22] and [4] require the existence of a domain hierarchy, such as parent-child, in order 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 R1and R2.

3. PRELIMINARIES AND NOTATION

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

***Definition 3.1**: A **graph** G is a tuple (V, E), where V is a finite set of nodes and $E \subseteq V \times V$ is a relation between nodes. $|V|$ is the number of nodes, $|E|$ the one of edges existing in G, and $\mathcal{G}$ is the set of all graphs.*

For a given graph G, $V(G) = V$ and $E(G) = E$. Without loss of generality, we assume that the relationships between the nodes are directed. This means that $(u, v) \in E$ does not imply $(v, u) \in E$. An undirected edge can be represented by two directed edges.

***Definition 3.2**: Given a graph $G = (V, E)$ and a node $u \in V$, the **degree** of u, $deg(u)$, is the number of outgoing edges of node u.*

***Definition 3.3**: A **Neighbor Query** $Q_{Neighbor}(G, u)$ of a graph $G = (V, E)$ and a node $u \in E$ returns the set of all nodes adjacent to u in G: $Q_{\text{Neigbhor}}(G, u) = \{v \in V | (u, v) \in E\}$.*

***Definition 3.4**: 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: $Q_{\text{Adjacency}}(G, u, v) = \text{true iff } (u, v) \in E$.*

***Definition 3.5**: 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_i in the i-th run of the algorithm, such that, if enc_d runs n times, $c_i = c_j$ for all $i, j \in \{1, \ldots, n\}$, and (3) a deterministic decryption algorithm dec_d^K that takes the cryptographic key K and the ciphertext c_i to revert the deterministic encryption, such that $\mathrm{dec}_d^K\left(\mathrm{enc}_d^K(\mathrm{m})\right) = \mathrm{m}$ for all encryption runs.*

***Definition 3.6**: A **probabilistic encryption scheme** $E_p = \left(k_{gen}, enc_p^K, dec_p^K\right)$ 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_i in the i-th run of the algorithm, such that, if enc_p runs n times, $c_i \neq c_j$ for all $i, j \in \{1, \ldots, n\}, i \neq j$, and (3) a probabilistic decryption algorithm dec_p^K that takes the cryptographic key K and the ciphertext c_i to revert the deterministic encryption, such that $dec_p^K\left(enc_p^K(m)\right) = m$ for all encryption runs.*

***Definition 3.7**: Given a graph $G = (V, E)$, **the multiset of degrees** N_E is the multiset that contains the degree of each node $u \in V$.*

4. THE SECRECY MODEL

In this section, we describe the prior knowledge of the adversary and the secrecy notion we target at.

4.1 The adversary knowledge

Let $transformed(G)$ denote the transformed graph after the graph transformation function has been applied to the graph G. We assume an adversary with the following knowledge:

K1: The adversary $\mathcal{A}$ has access to the transformed graph, $transformed(G)$. The adversary also knows the algorithm used to transform the graph.

K2: $\mathcal{A}$ knows the labels of all nodes in G.

K3: $\mathcal{A}$ knows N_E, but she does not know which degree of the multiset corresponds to which node.

Knowledge K1 is based on Kerckhoffs' principle, and it is a standard assumption from cryptology. Knowledge K2 and K3 describe realistic assumptions on external knowledge that the adversary could have. In what follows, when we refer to an *adversary* $\mathcal{A}$, we assume that $\mathcal{A}$ to not have any knowledge beyond K1, K2 and K3.

4.2 Our secrecy notion

We propose a secrecy notion for graph-structured data called Indistinguishability under Independent Node Permutation, Ind-INP. Our secrecy notion is based on the concept of indisinguishability presented in [13]. [13] has proven that this concept is equivalent to the standard 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 the concept of indistinguishability as our secrecy notion is the one featured in [13]: 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.

Definition 4.1: *A **graph transformation function** $f: \mathcal{G} \rightarrow \mathcal{G}$ is a function that transforms a graph G to another graph G'. The set of all graph transformation functions is $\mathcal{F}$.*

Conventional examples of graph transformation functions are addition of edges or deletion of nodes.

Let $G = (V, E)$ be the original graph-structured data and $G' = (V, E')$ be another graph with the same nodes as G, but with different edges E'. Permuting the nodes of the original graph G perturbs the edges and yields E'. Node permutation is also a graph transformation function. It is defined as follows:

***Definition 4.2**: An **Independent Node Permutation function** $\mathfrak{p}$ is a function $\mathfrak{p} \in \mathcal{F}$ where $V(\mathfrak{p}(G)) = V(G)$ for all graphs $G \in \mathcal{G}$ and $|E(\mathfrak{p}(G))| = |E(G)|$.*

Node permutation can be implemented as follows. Given a graph G, replace each node $v \in V$ with a random node $x \in V$. The identity function is a valid node permutation.

Let $\mathcal{A}$ be an adversary, τ a graph transformation function and $\mathfrak{p}$ an independent node permutation. Figure 4.1 features the experiment needed to define the secrecy notion Ind-INP. G_0 is the transformed graph of G, and G_1 is the transformed graph of the permuted graph $\mathfrak{p}(G)$. A random bit $b \in \{0,1\}$ is chosen. The transformed graph G_b is given to the adversary $\mathcal{A}$. $\mathcal{A}$ does not know whether τ has had G or $\mathfrak{p}(G)$ as input. The challenge of the adversary is to "guess" which one of the two graphs G_0 or G_1 has been the input of the transformation. $\mathcal{A}$ outputs a bit $\bar{b}$. The output of the experiment is defined to be 1 if $b = \bar{b}$, and 0 otherwise. If $Ind - INP_{\mathcal{A},\tau}(G) = 1$, we say that $\mathcal{A}$ has succeeded.

```
$Ind - INP_{\mathcal{A},\tau}(G)$:
    $G_0 \leftarrow \tau(G)$
    $G_1 \leftarrow \tau(\mathfrak{p}(G))$
    $b \leftarrow random(\{0,1\})$
    $\bar{b} \leftarrow \mathcal{A}((G_b))$
    $return\ 1\ if\ b = \bar{b}\ else\ 0$
```

Figure 4.1. The experiment $\mathbf{Ind - INP}_{\mathcal{A},\mathcal{T}}(\mathbf{G})$

***Definition 4.3**: A **graph transformation τ is called Ind-INP secure** if the function*

$$Adv_{\mathcal{A}}^{\tau}(G) := \left|Pr\left[\text{Ind-INP}_{\mathcal{A},\tau}(G) = 1\right] - \frac{1}{2}\right|$$

is negligible for any adversary $\mathcal{A}$ with knowledge K1, K2, and K3 whose computational effort is bounded to run in polynomial time.

***Definition 4.4**: A **function f is negligible** iff $\forall c \in \mathbb{N}\ \exists n_o \in \mathbb{N}$ such that for $n \geq n_0$, $f(n) < n^{-c}$.*

Although indistinguishability offers guarantees equivalent to semantic secrecy, it is not intuitive what this secrecy notion guarantees. Therefore, we describe a property of our secrecy definition Ind-INP, which will help users understanding the secrecy guarantees offered by our secrecy notion. It will be Theorem 4.1 that actually introduces this property, and before introducing it, some notation and definitions are needed. The following explains the probability of guessing the degree of a node in G, $P_{\mathcal{A}}$, by an adversary $\mathcal{A}$. Next to other things, $\mathcal{A}$ knows the set of nodes N and the multiset of degrees N_E. Calculating $P_{\mathcal{A}}$ requires the identification of all possible permutations of the elements of N_E.

***Definition 4.5**: Given a multiset of degrees N_E of a graph G, the **frequency of an element** $i \in N_E$ is the number of times the element i occurs in N_E. The set of the frequencies of the different elements in N_E is $\overline{N_E} = \bigcup_{i=1}^{k} d_i$, where d_i is the frequency of element i and k is the number of different elements in N_E*[1]*.*

***Lemma 4.1.** Given a multiset of degrees N_E of a graph G, the number of different permutations of the elements of the multiset N_E, $Per_{(N_E)}$ is given by the function*

$$Per_{(N_E)} = \frac{|N_E|!}{\prod_{d_i \in \overline{N_E}} d_i!} \qquad (4.1)$$

PROOF: Let k be the number of different elements in the multiset N_E and d_i, the frequency of element i, where $i \in \{1,\cdots,k\}$. There are $\binom{|N_E|}{d_1}$ ways to place the first different element, $\binom{|N_E|-d_1}{d_2}$ ways to place the second different element, and so on. Then the total number of different permutations $Per_{(N_E)}$ is

$$Per_{(N_E)} = \frac{|N_E|!}{d_1! \cdot (|N_E|-d_1)!} \cdot \frac{(|N_E|-d_1)!}{d_2! \cdot (|N_E|-d_1-d_2)!} \cdot \ldots \cdot \frac{(|N_E|-d_1-\cdots-d_{k-1})!}{d_k!}$$

$$= \frac{|N_E|!}{\prod_{d_i \in \overline{N_E}} d_i!} \qquad \blacksquare$$

***Lemma 4.2.** An adversary $\mathcal{A}$ can guess the degree of a node in a graph G with probability $P_{\mathcal{A}} = \left(\frac{1}{Per_{(N_E)}}\right)$.*

PROOF: From Lemma 4.1, we know the number of all possible permutations of elements of the multiset N_E. The probability of identifying the degree of the nodes is 1 divided by the number of permutations $Per_{(N_E)}$. $\blacksquare$

We now introduce Lemmas 4.3 and 4.4, which will help us to prove Theorem 4.1 subsequently.

***Lemma 4.3.** Given a multiset of degrees N_E of a graph G, $\frac{|N_E|!}{(|N_E|-|\overline{N_E}|+1)!}$ is a lower bound of the function $Per_{(N_E)}$.*

PROOF: Consider the denominator of the function $Per_{(N_E)}$,

$$\prod_{d_i \in \overline{N_E}} d_i! = (1 \times \cdots \times d_1)(1 \times \cdots \times d_2) \cdots (1 \times \cdots \times d_{|\overline{N_E}|})$$

The number of factors r different from 1 in $\prod_{d_i \in \overline{N_E}} d_i!$, is

$$r_{\left(\prod_{d_i \in \overline{N_E}} d_i!\right)} = \sum_{i=1}^{|\overline{N_E}|}(d_i - 1) = |N_E| - |\overline{N_E}|$$

Consider now the term $(|N_E| - |\overline{N_E}| + 1)!$

The number of factors r different from 1 in $(|N_E| - |\overline{N_E}| + 1)!$ is $r_{((|N_E|-|\overline{N_E}|+1)!\,)} = |N_E| - |\overline{N_E}|$

Consequently, for each factor a in $\prod_{d_i \in \overline{N_E}} d_i!$, there exists a factor b in $(|N_E| - |\overline{N_E}| + 1)!$ such that $b \geq a$.

Altogether $(|N_E| - |\overline{N_E}| + 1)! \geq \prod_{d_i \in \overline{N_E}} d_i!$ and $\frac{|N_E|!}{(|N_E|-|\overline{N_E}|+1)!}$ is a lower bound of the function $Per_{(N_E)}$. $\blacksquare$

***Lemma 4.4.** The lower bound of the function $Per_{(N_E)}$, $\frac{|N_E|!}{(|N_E|-|\overline{N_E}|+1)!}$ grows asymptotically faster than any polynomial for $|\overline{N_E}| \geq n_0$ if the condition $n_0 = \frac{|N_E|}{c}$ is fulfilled and $c \in \mathbb{R}^{>1}$.*

[1] We use $d_1 \cup d_2$ as a short hand for $\{d_1\} \cup \{d_2\}$.

PROOF: Without loss of generality we set $|\overline{N_E}| = \frac{|N_E|}{c}$ for $c \in \mathbb{R}^{>1}$. Next, we consider the function $g(x) = \frac{x!}{\left(x-\frac{x}{c}+1\right)!}$. $g(x)$ behaves like the term $A = \frac{|N_E|!}{\left(|N_E|-\frac{|N_E|}{c}+1\right)!}$ with respect to $|N_E|$ as the argument of the function. Now we analyze the limits of the denominator of the function $g(x)$: $\lim_{x\to\infty} x - \frac{x}{c} + 1 = 1$

Then $\frac{|N_E|!}{\left(|N_E|-\frac{|N_E|}{c}+1\right)!}$tends to $|N_E|!$ Consequently A grows faster than any polynomial for $|\overline{N_E}| \geq \frac{|N_E|}{c}$. $\blacksquare$

Lemma 4.4 says that for any polynomial $p(N_E)$ there exists a n_0 such that $\frac{|N_E|!}{(|N_E|-|\overline{N_E}|+1)!} \geq p(N_E)$ for all $|\overline{N_E}| \geq n_0$. In the following we give some examples to illustrate how this n_o looks like. Figure 4.2shows n_0 for the functions $p(N_E) = |N_E|^{10}$ and $Per_{(N_E)}$. We have conducted experiments with different polynomial functions, and we have always found a n_0 for which Lemma 4.4 holds.

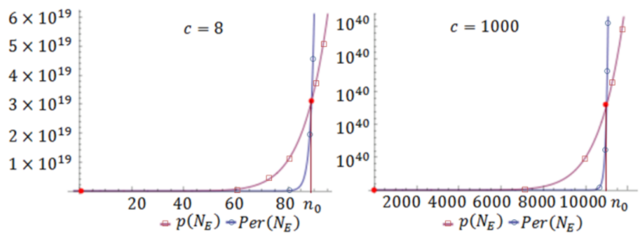

Figure 4.2. Functions $p(N_E)$ and $Per_{(N_E)}$

***Theorem 4.1.** If a graph transformation τ fulfills Definition 4.3, the probability $P_{\mathcal{A}}$ that an adversary learns the degree of a node in the graph given K1, K2 and K3 is negligible for $|\overline{N_E}| \geq \frac{|N_E|}{c}$, where $c \in \mathbb{R}^{>1}$.*

PROOF: $P_{\mathcal{A}}$ is the inverse of the function $Per_{(N_E)}$ (Lemma 4.1). From Lemma 4.2, we know that the lower bound of the function $Per_{(N_E)}$ grows asymptotically faster than any polynomial for any $|\overline{N_E}| > \frac{|N_E|}{c}$. Therefore, its inverse decreases faster than any polynomial. Then $P_{\mathcal{A}}$ is negligible. $\blacksquare$

5. OUR SECRECY APPROACH

In this section, we describe our bucketization approach for graphs. 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 the data owned by the clients. Clients apply techniques to secretize the data before passing it to the service provider, in order to maintain data secrecy.

***Definition 5.1**: Given a graph G, a **bucket** b is a finite set of edges of G. Each bucket has a $bucketID$ denoted by $bucketID(b)$. There is a maximum capacity of any bucket, denoted by $maxEdges$. The set of buckets of graph G is denoted by S_B.*

***Definition 5.2**: Given a graph G and its corresponding set of buckets S_B, the **index information** is a map of type $m\colon V \to S_B$*

that, for each node $u \in V$, contains the set of $bucketIDs$ of buckets that store at least one outgoing edge of u.

***Definition 5.3**: A **bucketization structure** B of a given graph G is a representation of G consisting of two parts, (1) a set of buckets S_B and (2) the index information. We call the set of all possible bucketization structures Bucketizations.*

Figure 5.1 illustrates a bucketization structure.

***Definition 5.4**: A **bucketization function** $buck: \mathcal{G} \rightarrow Bucketizations$ is a function that generates a bucketization structure B for a graph G.*

Section 5.5 will present the bucketization functions which we use.

***Definition 5.5**: Given bucketization structure B, an **encryption function** $enc: Bucketizations \rightarrow Bucketizations$ performs the actual 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 buckets, each edge is encrypted deterministically.*

Before outsourcing a graph G, the client applies a bucketization function on G. After encryption the bucketization structure is outsourced to the service provider. Figure 5.1 shows this process.

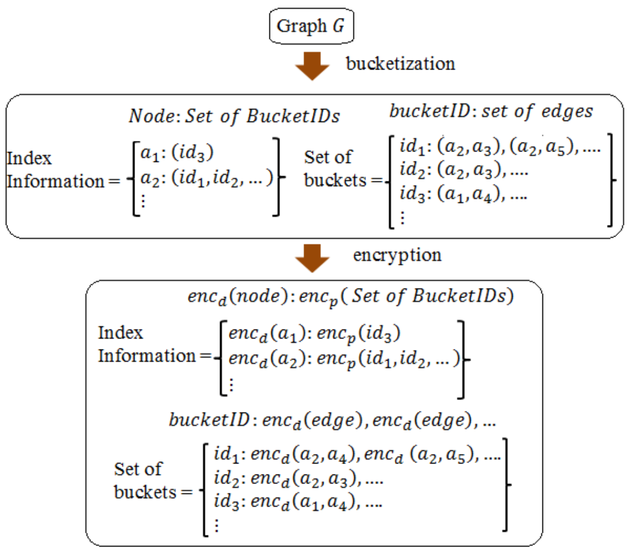
Graph G
bucketization
$Node: Set\ of\ BucketIDs$
$bucketID: set\ of\ edges$
Index Information = $\begin{bmatrix} a_1: (id_3) \\ a_2: (id_1, id_2, \dots) \\ \vdots \end{bmatrix}$
Set of buckets = $\begin{bmatrix} id_1: (a_2,a_3), (a_2,a_5), \dots \\ id_2: (a_2,a_3), \dots \\ id_3: (a_1,a_4), \dots \\ \vdots \end{bmatrix}$
encryption
$enc_d(node): enc_p(Set\ of\ BucketIDs)$
Index Information = $\begin{bmatrix} enc_d(a_1): enc_p(id_3) \\ enc_d(a_2): enc_p(id_1, id_2, \dots) \\ \vdots \end{bmatrix}$
$bucketID: enc_d(edge), enc_d(edge), \dots$
Set of buckets = $\begin{bmatrix} id_1: enc_d(a_2,a_4), enc_d(a_2,a_5), \dots \\ id_2: enc_d(a_2,a_3), \dots \\ id_3: enc_d(a_1,a_4), \dots \\ \vdots \end{bmatrix}$

Figure 5.1. Bucketization and Encryption on Graph *G*

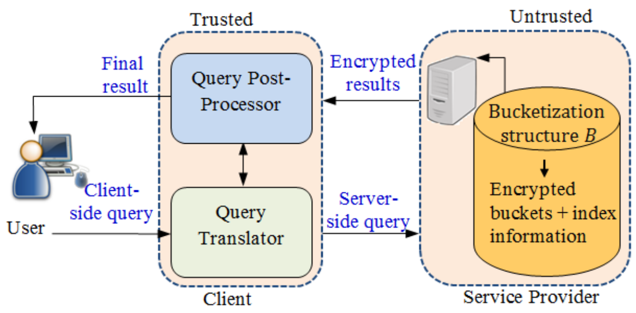

Figure 5.2. Query process

Query evaluation in the outsourced bucketization structure requires translating client-side queries to corresponding server-side queries. We assume that there are two components for query processing at the client side: (1) the query translator and (2) the query postprocessor, see Figure 5.2. The query translator translates the queries supported to server-side queries, the translation process is explained in Section 5.6. The server-side queries are sent to the server. The query post-processor is in charge of (1) receiving the encrypted results of the server-side query from the server, (2) unencrypting the results and (3) filtering any false positive, by applying the original client-side query. The final result is sent to the user.

5.2 Bucketization – Challenges

Even though using an encryption function solves the problem of chosen-plaintext attacks, this does not yield secrecy guarantees against frequency attacks. We for our part use bucketization to solve this problem. However, this is challenging in order to facilitate both good query performance as well as data secrecy. This is because it is not obvious how to assign edges to buckets, see Examples 5.1 and 5.2.

***Definition 5.6**: The **frequency of a bucket** is the number of edges that the bucket stores.*

***Example 5.1:** Consider 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 2 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 Alice's edges. Then the overall query processing effort and the client workload are rather large, i.e., the client has to filter more data.*

***Example 5.2:** Consider the email network from Example 5.1. 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. If an adversary knows that Alice is the user that has sent more emails than any other user, followed by Carol, the adversary can conclude that the bucket with four edges corresponds to Alice and the one with 3 edges to Carol. So the adversary has learned the actual degree of Alice and Carol. Moreover the adversary can learn that Alice has sent an email to Carol.*

So assigning edges to buckets randomly is likely to bog down query performance. 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. Although encryption offers some secrecy guarantees, they are not enough. To avoid information leakage, as illustrated in Example 5.2, buckets should be undistinguishable. We aim for an equal frequency of buckets, i.e., all buckets should reach their maximal capacity. Since a simple assignment may not always yield full buckets, 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 query-processing time significantly because the client must filter more false positives.

We proceed now to formalize our bucketization problem.

5.3 The Bucketization Problem

The bucketization problem is as follows:

Given as input a graph $G = (V, E)$ we search for a bucketization B that meets Constraints c1-c4:

- *c1 Each edge $(u, v) \in E$ is assigned to one bucket.*
- *c2 Each bucket stores at most $maxEdges$ edges.*
- *c3 Edges adjacent to the same node are placed in as few buckets as possible. Formally, let the function $ind: V \times Bucketizations \to \mathbb{N}$ be as follows:*
  $ind(u, B) := |\{b \in B | \exists\, x \in V | (u, x) \in b\}|$.
  Then $\forall B' \in Bucketizations: ind(u, B) \leq ind((u, B')$
- *c4 The total number of buckets should be as small as possible (while prioritizing Constraint c3).*

We prioritize Constraint c3 over c4, so that query performance is not affected.

***Definition 5.7**: An **optimal bucketization** is a bucketization that meets Constraints c1 to c4.*

In the next subsection we show that the problem of finding the bucketization defined by Constraints c1-c4 is NP-hard.

5.4 Hardness Result

Our bucketization problem is NP-hard. To prove this, we reduce the Bin-packing problem (BP problem) [20] to our problem. The BP problem has been proven to be NP-hard in [20]. We start by introducing the BP problem.

***Definition 5.8**: Let a set of n bins $C = \{c_1, c_2, \dots, c_n\}$ and the same number of n items $I = \{a_1, a_2, \dots, a_n\}$ 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 function $BP: I \to C$ that maps each item in I to one bin in C such that the following Constraints bp1, bp2 and bp3 are met.*

bp1 An item is assigned to only one bin.

bp2 The sum of the weight of all items assigned to a bin does not exceed the bin capacity w_c. $\forall c_j \in C: W_{c_j} \leq w_c$ where $W_{c_j} = \sum_{i \in |\{a \in I | BP(a) = c_j\}|} w_{a_i}$.

bp3 The number m of bins used is as small as possible, i.e., $m = \sum_{c_j \in C} min(1, |BP(c_j)|)$.

For the hardness proof, we introduce Lemmas 5.1 and 5.2. They 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.

We start by identifying the steps required to construct the *transformed BP*.

***Input construction process**:* Given 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=$\{a_i, a_{i1}, a_{i2}, \cdots, a_{iw_{a_i}}\}$ and the set of edges E_i=$\{(a_i, a_{i1}), (a_i, a_{i2}), \cdots, (a_i, a_{iw_{a_i}})\}$.
- The graph *is* $(\bigcup_{i=1}^{n} V_i, \bigcup_{i=1}^{n} E_i)$.

***Lemma 5.1. Input transformation**. 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 the *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. However, $\sum_{i=1}^{n}(w_{a_i} + 1) \leq (w_c + 1) \cdot n$ and w_c is a constant, so the construction is still polynomial. Then an *initial BP* can be transformed to a *transformed BP* in polynomial time. ■

Example 5.3 illustrates the construction of the *transformed BP*.

***Example 5.3:** 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 5.3 *shows the transformed BP.*

Once we have built the *transformed BP,* we can run an algorithm that solves the bucketization problem, by setting w_c to $maxEdges$. The solution of the *transformed BP* is a bucketization B. Figure 5.4.a shows the set of buckets S_B of Example 5.3, b_1 and b_2 are the $bucketIDs$. Since we set $maxEdges = w_c$, it holds for all buckets $b \in S_B$ that $|b| \leq w_c$.

The next lemma, Lemma 5.2, states that a solution of the *initial BP* can be constructed in polynomial time from a solution of the *transformed BP.* Before moving to Lemma 5.2, we first explain the solution construction process.

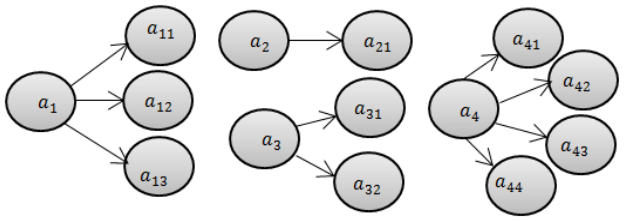

Figure 5.3. *Transformed BP* of Example 5.3.

a) The set of buckets S_B solution of the *transformed BP*

$$S_B = \begin{Bmatrix} \{(a_1, a_{11}), (a_1, a_{12}), (a_1, a_{13}), (a_3, a_{31}), (a_3, a_{32})\}_{b_1} \\ \{(a_2, a_{21}), (a_4, a_{41}), (a_4, a_{42}), (a_4, a_{43}), (a_4, a_{44})\}_{b_2} \end{Bmatrix}$$

b) The solution of the *initial BP*

$C = \{c_1, c_2, c_3, c_4\}$ $\quad c_1 = \{a_1, a_3\}$

$m = |S_B| = 2$ $\quad c_2 = \{a_2, a_4\}$

Figure 5.4. Solution of Example 5.3

Solution transformation process: A solution of the *initial BP* can be constructed from a solution of the *transformed BP* as follows:

- Identify the number of bins m needed to store the items. Each bucket represents one bin. Then $m = |S_B|$.
- Identify the set of items that each bin will store. c_i=$\{x | \exists (x, y) \in b_i\}$. Figure 5.4.b shows the solution constructed for the initial BP of Example 5.3.

***Lemma 5.2. Output transformation.** A solution of the transformed BP can be transformed to one of the initial BP in polynomial time.*

Proof: Consider a bucketization of the *transformed BP* that fulfills Constraints c1-c4. 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 c1 and c3 of the bucketization problem. Constraint c1 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 c2 of the bucketization problem. For all bins $c_i \in C$, $|c_i| = |b_i|$ and $maxEdges = w_c$, then $|b_i| \leq w_c$, which fulfils bp1.

Third, bp3 is fulfilled because of Constraint c4. 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. For all buckets $b_i \in S_B$, $|b_i| = |c_i|$ and $\sum_{i=1}^{n}|c_i| = \sum_{i=1}^{n} w_{a_i}$. Then the complexity of the reconstruction is $O(m)$, where m is the total number of edges and m=$\sum_{i=1}^{n} w_{a_i}$. ■

Theorem 5.1. *Finding an optimal bucketization that meets Constraints c1 - c4 is NP-hard.*

PROOF: With Lemmas 5.1 and 5.2 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 [20], the bucketization problem is NP-hard as well. ■

In the next section we present our bucketization approach.

5.5 The Bucketization Algorithm

Due to the complexity of the problem, we use heuristics to find an approximate solution to an optimal bucketization.

The bucketization algorithm consists of (1) partitioning the edges of a graph G into buckets with the constraints established in Section 5.3 and (2) creating the corresponding index information. The algorithm has an initialization phase and a merging phase.

5.5.1 The Initialization Phase

Algorithm 1 is the initialization phase of our bucketization approach for a graph G. It starts by padding the labels of the nodes to ensure that all strings that represent an edge have the same length. Then the algorithm follows the next steps: (1) create the buckets needed to store the edges of G, (2) assign $maxEdges$ edges belonging to the same node to each bucket randomly and (3) generate the index information.

We justify the need for padding in Section 5.7. Example 5.4 illustrates how the assignment of edges works, and Example 5.5 explains the need for randomness with this assignment.

Example 5.4: *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 buckets, 10 random edges are chosen from the remaining ones for the second bucket, and the 7 remaining edges go to the third bucket.*

Example 5.5: *For the sake of an easy example, image a setting where emails can be revoked without difficulty. For this example, we consider the buckets in* Figure 5.5. *Assume that the bucketization algorithm does not assign edges randomly. If an adversary knows that user A has sent 4 emails, learns that the system has revoked one of the emails and sees that Bucket b2 is deleted, although the edges are encrypted, she will learn that the revoked email was the last one, i.e., the email sent to user* E*. A random assignment of edges reduces the probability of the adversary learning extra information.*

Algorithm 1: Initialization()

```
INPUT: Graph: $G(V,E)$, int: maxEdges
OUTPUT: initial bucketization: $B_0$
1:  //Step 0: pad the length of all nodes
2: pad.labelOfNodes();
3:  //Step 1: create a sufficient number of buckets for each node
4:  for each $v$ in $V$ {
5:      create (ceil(1, v.numberOfEdges()/maxEdges)) buckets;
6: //Step 2: assign edges of each node to a corresponding bucket
7:      assign randomly up to maxEdges edges of $v$ to each bucket;
8: generate the corresponding index information;
9: }
```

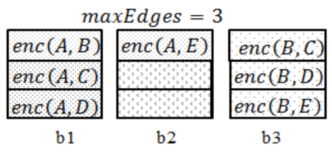

Figure 5.5 Illustration of Example 5.5

Definition 5.9*: Given a graph* G*, the* ***initial bucketization*** B_0 *is the result of the initialization phase of the bucketization algorithm applied to* G*.*

After the initialization phase of the bucketization process, all edges have been placed into their buckets. At this point, some buckets may not have reached their maximal capacity. Even if we encrypt the buckets at this stage, the initial bucketization is not secure. Recall that edges are encrypted individually. Then an adversary can learn the frequency of buckets. Furthermore, if the degree of a node is less than or equal to $maxEdges$, its edges have been placed in one bucket exclusively. An adversary could now gain extra knowledge by analyzing the initial bucketization, see Example 5.2.

5.5.2 The Merging Phase

Definition 5.10*:* ***Bucket merge*** *is a process 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 in order to fulfill Constraint c4.

Definition 5.11*: Given a graph* G*, a* ***final bucketization*** B *is a bucketization resulting from the initialization and merging phases applied to* ***G****.*

Algorithm 2 identifies pairs of buckets that can be merged in order to obtain buckets with the same frequency, to address the secrecy issues from Section 5.5.1. Different heuristics are conceivable at this stage. We choose a First Fit Decreasing approach (FFD) [8]. We will justify this decision after having explained the algorithm. When the algorithm starts, Lines 1-3, it creates three sets: (1) B', which contains buckets that do not yet have $maxEdges$; B' is sorted based on the frequency of each bucket in descending order, (2) B_f, which contains full buckets, and (3) B_m, an auxiliary set that contains 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, Lines 4-6. If it finds one, the function $merge(b_i, b_j)$, Line 7, creates a new Bucket b to store the edges of b_i and b_j. The Buckets b_i and b_j are removed from B' and B_m, and the *index information* is updated. If the new bucket b reaches its maximal capacity, b is placed in B_f, Lines 8-9. Otherwise it is placed in B_m so that it can be considered again for a merge, Line 11. If there is not a Bucket b_j available for a merge, b_i is placed in B_m, Lines 15-16. Once the merging process has finished, *dummy edges* are added to the buckets that have not reached $maxEdges$, Lines 18-20. 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.

Algorithm 2: Merge buckets()

```
INPUT: initial bucketization  $B_0$, int: maxEdges
OUTPUT: final bucketization:  $B_f$
1: Initialize: $B'$:={ $b \in B_i$| b.numberOfEdges() < maxEdges}
2: Initialize: $B_f := B_i \backslash B'$; $B_m := \{ \}$
3: Order $B'$ by number of edges in decreasing order;
4:  for each $b_i \in B'${
5:     for each $b_j \in B_m$ {
6:        if $b_i$ fits in $b_j$
7:           $b \leftarrow$ merge($b_i, b_j$);
8:           if b.numberOfEdges() = maxEdges
9:              add b to $B_f$;
10:          else
11:             add b to $B_m$;
12:          delete $b_i$, $b_j$;
13:          break the loop and continue with the next $b_i$;
14:    }
15:       if $b_i$ does not fit in any available $b_j \in B_m$
16:          move $b_i$ to $B_m$;
17: }
18: for each b in $B_m${
19:    addDummyEdges();
20:    add b to $B_f$;
21: }
22:  enc($B_f$);
```

Analysis of the Merging Phase: Because finding an optimal bucketization solution is computationally intractable (NP-hard), we introduce a heuristic to solve the problem. However, different heuristics are conceivable for the merging. We for our part use a First Fit Decreasing (FFD) approach. Garey et al. have demonstrated in [8] that the worst case solution for the bin packing problem with the FFD approach is far of the optimal by a factor of $\frac{11}{9}$. Other approaches, such as Best Fit (BF) and Next Fit (NF), have a worse approximation ratio, $\frac{17}{10}$ and 2 respectively [8].

5.6 Query Transformation

Unlike other approaches such as [18], our bucketization approach does not lose any information regarding the original graph. Consequently, there is no limitation regarding the kind of query we can process in principle. However, with respect to the client workload, our approach is more efficient answering neighbor and adjacency queries than answering other queries such as finding a path between two given nodes. In the following, we discuss the processing of neighbor and adjacency queries. These queries are essential information needs regarding graphs [17].

Algorithm 3: Neighbor Query Processing given a bucketization structure *B*

```
INPUT: $Q_{Neighbor}(G, u)$, key
OUTPUT: Edges:= {}
1: Initialize: EncBucketIDs:={}, BucketIDs:={}, EncEdges:={},
   EdgesTem :={}, Edges:={};
2: encNode ←client.$enc_d^{key}(u)$;
3: if server.indexInformation.node=encNode
4:    EncBucketIDs ← indexInformation.BucketIDs(encNode);
5:     for each $b_i$in EncBucketIDs
6:        $b$ ←client. $dec_p^{key}(b_i)$;
7:        add $b$ to BucketIDs;
8:     for each $b$ in BucketIDs
9:        for each bucket in server.SetOfBuckets
10:          if bucket.bucketID = $b$
11:            eEdge ← SetOfBuckets.Edge(bucket);
12:             add eEdge to EncEdges;
13:    for each eEdge in EncEdges
14:       edge ←client. $dec_d^{key}$(eEdge);
15:             add edge to EdgesTemp;
16:    for each edge in EdgesTemp
17:       if (edge.isFalsePositive!=true)
18:       add edge to Edges;
19: return Edges;
```

Algorithm 4: Adjacency Query Processing given a bucketization structure B

```
INPUT: $Q_{Adjacency}(G, u, v), key$
OUTPUT: Boolean isEdge ∈ $\{true, false\}$
1: Initialize: Boolean isEdge = $false$;
2: encEdge ←client.$enc_d^{key}(u, v)$
3: if server.SetOfBuckets.Edge contains encEdge
4:    isEdge= $true$;
5: return isEdge;
```

Client-side queries are transformed to server-side queries. Algorithm 3 shows the transformation process for neighbor queries, $Q_{Neighbor}(G, u)$. The client starts by encrypting Node u and generating the server-side query, Lines 2-3. This query retrieves the set of b$ucketIDs$ of the encrypted node $enc_d^{key}(u)$ from the *index information* and returns it to the client, Line 4. The client unencrypts it and generates a new server-side query Lines 5-7. This new query retrieves the edges stored in buckets whose $bucketID$ corresponds to one of the unencrypted b$ucketIDs$, Lines 8-12. Finally the client unencrypts the edges and filters false positives, Lines 13-19. 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 $enc_d^{key}(u, v)$, Line 2. The server-side query searches in the *set of buckets* for this encrypted edge, Lines 3-4. Iff there exists such an edge, the nodes are adjacent.

5.7 Our Bucketization Approach is Ind-INP

Recall that the output of the bucketization algorithm consists of two parts, the *index information* and the set of *buckets*. See Figure 5.6 where $|C_{(node)}|$ is the length of the ciphertext representing an encrypted node, $|C_{(bucketIDs)}|$ is the length of the ciphertext representing the set of $bucketIDs$, $|C_{(edge)}|$ is the length of the ciphertext representing an encrypted edge, $|N_{(enc)}|$ is the number of encrypted nodes and $|EB|$ is the number of buckets. In this section we will prove that our Bucketization Approach fulfills the secrecy notion Ind-INP defined in Section 4.2. To facilitate the proof, we first prove that our bucketization algorithm is Ind-INP with respect to the *set of buckets* output, Lemma 5.3, and with respect to the *index information* output, Lemma 5.4.

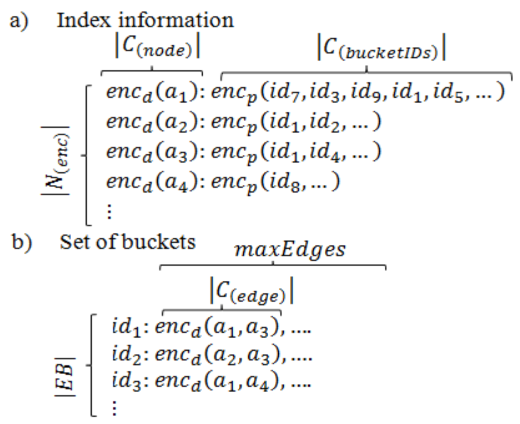
a) Index information
$|C_{(node)}|$
$|C_{(bucketIDs)}|$
$|N_{(enc)}|$
$enc_d(a_1)$: $enc_p(id_7, id_3, id_9, id_1, id_5, \ldots)$
$enc_d(a_2)$: $enc_p(id_1, id_2, \ldots)$
$enc_d(a_3)$: $enc_p(id_1, id_4, \ldots)$
$enc_d(a_4)$: $enc_p(id_8, \ldots)$
b) Set of buckets
$maxEdges$
$|C_{(edge)}|$
$|EB|$
id_1: $enc_d(a_1, a_3)$, ...
id_2: $enc_d(a_2, a_3)$, ...
id_3: $enc_d(a_1, a_4)$, ...

Figure 5.6. Abstract output of the bucketization algorithm

Lemma 5.3. *Given the set of buckets the Bucketization algorithm has generated, an adversary $\mathcal{A}$ cannot distinguish whether the graph G or the permuted graph $\mathfrak{p}(G)$ has been the input of the bucketization algorithm.*

PROOF: All buckets Index information have the same frequency, namely $maxEdges$. So buckets are undistinguishable. The only characteristic that can change in the *set of buckets* when the input of the bucketization algorithm changes is the number of buckets $|EB|$. $|EB|$ depends on the number of nodes $|N|$ and the multiset of degrees of the nodes N_E. With knowledge K1, K2 and K3, (K2 and K3, to be exact) an adversary has access to $|N|$ and N_E. However, since $\mathfrak{p}(G)$ is just a permutation of G, $|N|$ and N_E of both graphs are identical. Consequently, an adversary cannot distinguish whether G or $\mathfrak{p}(G)$ has been the input of the bucketization algorithm, given knowledge K1, K2 and K3. ■

Lemma 5.4. *Given the index information the Bucketization algorithm has generated, an adversary $\mathcal{A}$ cannot distinguish whether the graph G or the permuted graph $\mathfrak{p}(G)$ has been the input of the bucketization algorithm.*

PROOF: The only characteristic that can change in the *index information* when the input of the algorithm changes are the number of encrypted nodes $|N_{(enc)}|$ and the length of the ciphertext of the set of $bucketIDs$, $|C_{(bucketIDs)}|$. $|N_{(enc)}|$ and $|C_{(bucketIDs)}|$ depend on the number of nodes $|N|$ and the multiset of degrees N_E. With knowledge K2 and K3, an adversary has access to $|N|$ and N_E. However, the permutation does not modify the nodes in the graph. Also, since $\mathfrak{p}(G)$ is just the permutation of G, N_E of both graphs are identical. Consequently, an adversary cannot distinguish whether G or $\mathfrak{p}(G)$ has been the input of the algorithm given knowledge K1, K2 and K3. ■

Theorem 5.2. *Our bucketization algorithm fulfills the secrecy notion Ind-INP (Definition 4.3).*

PROOF: The bucketization algorithm produces two outputs, the *index information* and the *set of buckets.* The only characteristics that change when the input of the algorithm changes are $|EB|$, $|N_{(enc)}|$ and $|C_{(bucketIDs)}|$. We have proven in Lemmas 5.3 and 5.4 that the three characteristics are identical for graphs G or $\mathfrak{p}(G)$. Then any combination of these characteristics does not violate Definition 4.3. Thus, our algorithm is Ind-INP. ■

6. PERFORMANCE MODEL

We start this section by describing scale-free networks. Then we present our performance model which consists of (1) the Number-of-Buckets Model and (2) the Query-Cost Model.

A performance model is important because it allows predicting the behavior of an algorithm and facilitates meaningful comparisons or evaluations of algorithms. In our approach, the number of buckets obtained after applying our bucketization algorithm to a graph G is a crucial parameter for query performance.

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 a given graph. In the next section, due to the importance of scale-free networks, we review some of their characteristics. Based on these properties, we 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. These 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 scale-free power-law distribution [3].

Growth. Real-world networks often are the result of a continuous growth process. At each time step, a new node is added to the network and connected with existing nodes.

Preferential attachment. Nodes with higher degree will have higher probability to be connected to a new node. This property has the effect that most nodes in the network will have only few edges, and a few nodes gradually turn into hubs, i.e., their degree greatly exceeds the average.

Barabasi et al. have introduced a model capturing the properties of scale-free networks, the Barabasi-Albert Model (BA) [3]. In the following, we review some important BA parameters.

Degree Exponent, γ. The degree exponent 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. Barabasi et al. have observed that the degree exponent of many real networks is between 2 and 3.

Growing 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 degree exponent and the growing parameter, it is possible to calculate the probability that a randomly chosen node has degree of k [3]. The probability is

$$\rho_k = \frac{2m(m+1)}{k(k+1)(k+2)} \tag{6.1}$$

Edges. The number of edges $|E|$ in the BA is $|E| = m \cdot N$, where N is the number of nodes.

Largest node degree, k_{max}. The expected value of the largest node degree in the BA is $k_{max} \sim N^{\frac{1}{\gamma-1}}$.

***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}.

Based on these BA characteristic, we derive a model of the expected number of buckets with our bucketization algorithm.

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 6.1 captures the number of buckets that have reached their maximal capacity after the initialization phase of the algorithm.

Lemma 6.1. *The number of full buckets after the initialization phase of the algorithm is*

$$Bucket_{\textit{Full-ini}} = \sum_{k=k_{min}}^{k_{max}} \left(N \cdot \rho_k \cdot \left\lfloor \frac{k}{maxEdges} \right\rfloor \right) \quad (6.2)$$

Proof: Given a node $u \epsilon V$ in a graph G with degree k_u, the number of full buckets generated for u after the initialization phase is $EB_{Full_u} = \left\lfloor \frac{k_u}{maxEdges} \right\rfloor$. EB_{Full_u} is calculated regardless of the other nodes in G. Next, it is required to calculate EB_{Full_u} for all nodes $u \in V$. According to the BA properties, the probability that a randomly chosen node has degree of k is given by ρ_k. Then the total number of nodes with degree k is $N \cdot \rho_k$. For all the nodes with degree k, the total number of buckets is $N \cdot \rho_k \cdot \left\lfloor \frac{k_u}{maxEdges} \right\rfloor$. 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 6.2.

Lemma 6.2. *The number of edges that have been assigned to buckets that are not full is*

$$Edges_{NFB} = |E| - Bucket_{\textit{Full-ini}} \cdot maxEdges \quad (6.3)$$

Proof: The number of edges already stored in full buckets after the initialization phase is $Bucket_{\textit{Full-ini}} \cdot maxEdges$. 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 Bucket Estimation Model, see Theorem 6.1.

Theorem 6.1. *Given a graph $G = (V, E)$ that follows the BA Model, the expected number of buckets, E_B, is in the range:*

$$Bucket_{\textit{Full-ini}} + \left\lceil \frac{Edges_{NFB}}{maxEdges} \right\rceil \leq E_B \leq$$
$$Bucket_{\textit{Full-ini}} + \frac{11}{9} \cdot \left\lceil \frac{Edges_{NFB}}{maxEdges} \right\rceil \quad (6.4)$$

The lowest value of the range represents an optimal solution, and the highest value represents the worst case scenario of our bucketization algorithm.

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, $Bucket_{\textit{Full-ini}}$, 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 6.1 gives a range of the expected number of dummy edges.

Corollary 6.1. *Given a graph $G = (V, E)$ that follows the BA Model, the expected number of dummy edges, D_E, is in the range*

$$\left(Bucket_{\textit{Full-ini}} + \left\lceil \frac{Edges_{NFB}}{maxEdges} \right\rceil \right) \cdot maxEdges - |E| \leq D_E \leq$$
$$\left(Bucket_{\textit{Full-ini}} + \frac{11}{9} \cdot \left\lceil \frac{Edges_{NFB}}{maxEdges} \right\rceil \right) \cdot maxEdges - |E| \quad (6.5)$$

Proof: The lower bound of the expected number of buckets from Theorem 6.1 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 for the upper bound. ■

The number-of-buckets model helps us to predict the query performance. Depending on the type of queries, the query workload is divided between the client and the server, e.g., with neighbor queries the client has to filter false positives. Lemma 6.1 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 $Bucket_{\textit{Full-ini}}$ to the expected number of buckets from Theorem 6.1. 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 number of edges per node, but also on parameter $maxEdges$. As mentioned, adjacency queries do not require work at the client. However, dummy edges affect the query performance at the server. Preliminary experiments of ours show that more dummy edges increase the query execution time at the server proportionally.

6.3 Query-Cost Model

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

Definition 6.1*: The **query performance ratio** of a given query Q, an original graph G and its corresponding bucketization structure B at the server side is $SP_Q = SR_{Q\text{-}B} / SR_{Q\text{-}G}$ and the query performance ratio at the client side is $CP_Q = CR_{Q\text{-}B} / CR_{Q\text{-}G}$.*

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 in the original graph G has a complexity of $O(|E|)$. In the bucketization structure, a single lookup in the *index information* has complexity of $O(|V|)$ and a single lookup in the *set of buckets* has a complexity of $O(|E| + |D_E|)$, where $|D_E|$ is the number of dummy edges.

Lemma 6.3. *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 as follows, with $deg(u)$ being the degree of u:*

$$SP_{Q_{Neighbor}(G,u)} = \frac{O\left(\left\lceil\frac{deg(u)}{maxEdges}\right\rceil\cdot(|E|+|D_E|)\right)}{O(|E|)} \quad (6.6)$$

$$CP_{Q_{Neigbhor}(G,u)} = O\left(\left\lceil\frac{deg(u)}{maxEdges}\right\rceil \cdot maxEdges\right) \quad (6.7)$$

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 u. Then the effort of executing a neighbor query on the server side is $SR_{Q_{Neighbor(G,u)}} = O(|E|)$. 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 $O(1)$.
2. Retrieve the set of $bucketIDs$ of node u from the *index information*. This step has a complexity of $O(|V|)$. Using the BA model, we can write $|V|$ as $\frac{|E|}{m}$.
3. Decrypt the set of $bucketIDs$. The decryption operation has a complexity of $O\left(\left\lceil\frac{deg(u)}{maxEdges}\right\rceil\right)$.
4. For each $bucketID$, one lookup in the *set of buckets* 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 $O\left(\left\lceil\frac{deg(u)}{maxEdges}\right\rceil \cdot (|E| + |D_E|)\right)$.
5. Decrypt and filter the $\left\lceil\frac{k_u}{maxEdges}\right\rceil \cdot maxEdges$ edges. The decryption and filtering is in $O\left(\left\lceil\frac{k_u}{maxEdges}\right\rceil \cdot maxEdges\right)$.

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:

$$SR_{Q_{Neighbor(G,u)\text{-}B}} = O\left(\left\lceil\frac{deg(u)}{maxEdges}\right\rceil \cdot (|E| + |D_E|)\right)$$

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

Finally,

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

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

Lemma 6.4. *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 $SP_{Q_{Adjacency}(G,u,v)} = \frac{O((|E|+|D_E|))}{O(|E|)}$ and $CP_{Q_{Neigbhor}(G,u)} = O(1)$.*

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

In the transformed graph the following steps are required:

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

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

Then $SP_{Q_{Adjacency}(G,u,v)} = \frac{O((|E|+|D_E|))}{O(|E|)}$ $\blacksquare$

From the Query-Cost Model we can learn that for adjacency and neighbor queries the parameter $maxEdges$ plays an important role in the query performance effort at client and server. If $maxEdges$ increases, the number of dummy edges increases and the server requires more effort in order to answer queries. At the client-side, for answering neighbor queries if $maxEdges$ increases, the workload at the client increases too, because the client has to filter more false positives.

7. EXPERIMENTS

In this section we present experiments to evaluate (1) the accuracy of our number-of-buckets model and (2) the performance of our bucketization approach.

7.1 Experiment Setup

7.1.1 Input datasets

In our experiments we use synthetic and real datasets.

Synthetic datasets: We have used Networkx [11] to generate 8 different undirected graphs that follow the BA Model. Table 7.1 shows the characteristics of the data, where N is the number of nodes, m the growing parameter and E the number of edges.

Table 7.1. Characteristics of the synthetic data

Synthetic Data	N	m	E
G1	5000	6	29964
G2	5000	8	39936
G3	10000	6	59964
G4	10000	8	79936
G5	40000	8	319936
G6	40000	10	399900
G7	150000	8	1199936
G8	150000	10	1499900

Real datasets: We have used as real datasets the actor network [14] and the Web network [16]. Barabasi et al. have proven that both networks are scale-free. The actor network contains 1048575 edges and 1137725 nodes, 89150 nodes represent actors and 1048575 nodes represent movies. An edge connects a movie with an actor who has played in it. The actor network exhibits the preferential attachment feature. This is because, if an actor has played in more movies, a casting director is more familiar with his or her skills. Then an actor with higher degree has higher chances to be considered for a new role. The Web network contains 2381903 nodes and 2312497 edges, and its growing parameter m is 5. The nodes in the Web network are web pages, and the edges represent hyperlinks between them.

7.1.2 Queries

Based on initial experiments, we observe that node degree plays an important role in the query performance evaluation. Therefore, the nodes that will be part of the experiments sample should be carefully selected in order to have a representative sample 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 in two groups $Random\ Q_{Neighbor}(G, u)$ and $Hub\ Q_{Neighbor}(G, u)$. For $Random\ Q_{Neighbor}(G, u)$, we select the input node u randomly from the set of nodes V without considering the hubs in the graph. For $Hub\ Q_{Neighbor}(G,u)$, we identify the hubs in the graph and use them as input.

For adjacency queries, $Q_{Adjacency}(G,u,v)$, the nodes u,v were selected randomly from the set of nodes V. The execution time of adjacency queries depends on the total number of edges including dummy edges (Section 6.3). So a distinct consideration of hubs is not necessary in this case.

7.1.3 Evaluation Measures

We use six metrics which let us evaluate the accuracy of the NBM and the performance of the bucketization approach.

The NBM metrics are:

$\boldsymbol{E_{TotalEB}}$: This metric quantifies the number of buckets obtained when applying our bucketization algorithm to Graph G.

$\boldsymbol{E_{dummy}}$: This is the percentage of dummy edges when applying our bucketization algorithm to Graph G.

$\boldsymbol{E_{Buckets_{Full-ini}}}$: This is the percentage of buckets that are full after the initialization of the bucketization algorithm on Graph G.

The bucketization performance metrics are:

$\boldsymbol{P_{TQprocessing}}$: This metric quantifies the total query processing time using our bucketization structure B, i.e., it adds up the processing time at the client and at the server.

$\boldsymbol{P_{CQprocessing}}$: This metric quantifies the client query processing time when using a bucketization structure, i.e., the time required by the client to decrypt the results returned from the server and filter false positives.

$\boldsymbol{P_{RQprocessing}}$: This is the ratio of the total query processing time using a bucketization structure B and its original graph G.

7.2 Results

We now present the results of the experiments. First, the evaluation of the NBM is discussed, then performance. We study the effect of each parameter from Section 7.1.3 one by one.

7.2.1 NBM Evaluation

$\boldsymbol{E_{TotalEB}}$**:** Figure 7.1 shows the numbers of buckets obtained with the synthetic data. Figure 7.2 shows the numbers of buckets obtained with the real datasets. For both types of datasets we have considered different values for 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 6.1.

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.

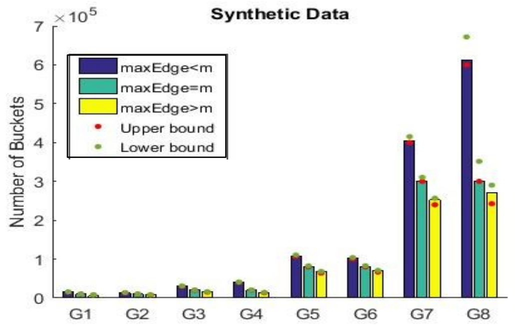

Figure 7.1. $\boldsymbol{E_{TotalEB}}$ **obtained for the synthetic data**

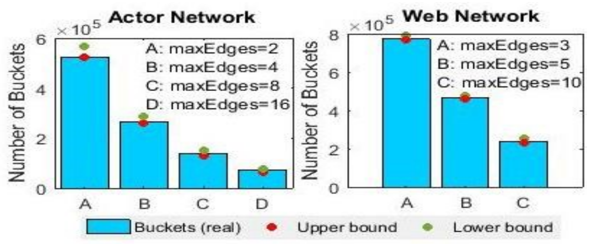

Figure 7.2. $\boldsymbol{E_{TotalEB}}$ **obtained for the real datasets**

$\boldsymbol{E_{dummy}}$**:** We calculate the percentage of dummy edges in comparison with the size of the original graph for the synthetic data and real datasets. For space constraints, Table 7.2 shows the average percentage of dummy edges for the synthetic data, i.e., eight datasets.

Table **7.3** shows the exact percentage of dummy edges for the real datasets. The number of dummy edges needed increases, as parameter $maxEdges$ takes greater values than m. More dummy edges means a larger database. This is likely to affect the efficiency of the querying process on the server as well, this will be examined in Section 7.2.2.

Table 7.2. $\boldsymbol{E_{dummy}}$ **for the synthetic datasets**

Synthetic Data	1<maxEdges<m	maxEdges=m	maxEdges>m
	1.217%	0.889%	26.513%

Table 7.3. $\boldsymbol{E_{dummy}}$ **for the real datasets**

Actor network	maxEdges=2	maxEdges=4	maxEdges=16
	0.028%	0.748%	7.629%
Web network	maxEdges=3	maxEdges=5	maxEdges=10
	0.223%	1.112%	6.349%

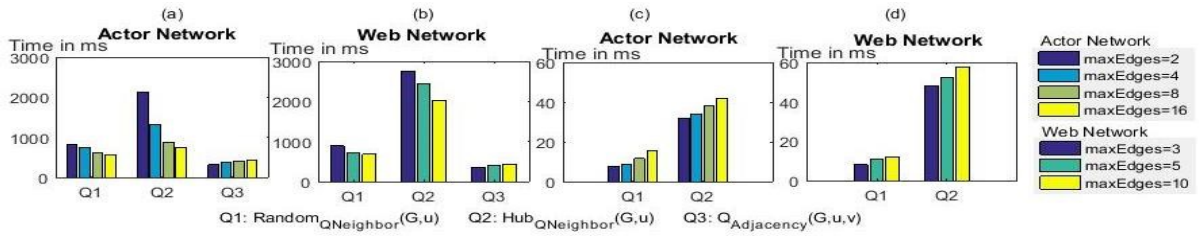

Figure 7.3. $P_{TQprocessing}$ and $P_{CQprocessing}$ in the Actor and Web Networks

$\boldsymbol{E_{Buckets_{Full\text{-}ini}}}$: Table 7.4 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 7.5. The number of full buckets decreases, as parameter $maxEdges$ takes greater values than m. If there are edges that belong to different nodes inside a bucket, then 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 implies fewer buckets for the merging process, fewer dummy edges and fewer false positives when querying.

Table 7.4. $E_{Buckets_{Full\text{-}ini}}$ for the synthetic datasets

Synthetic Data	1<maxEdges<m	maxEdges=m	maxEdges>m
	88.79%	86.81%	46.65%

Table 7.5. $E_{Buckets_{Full\text{-}ini}}$ for the real datasets

Actor network	maxEdges=2	maxEdges=4	maxEdges=16
	59.15%	55.78%	14.49%
Web network	maxEdges=3	maxEdges=5	maxEdges=10
	81.38%	80.18%	47.96%

7.2.2 Performance Evaluation

As in the previous section, we have conducted experiments with synthetic and real datasets. The result analysis is the same for both cases. For space constraints, we only present the results on the real data.

$\boldsymbol{P_{TQprocessing}}$: Figure 7.3(a) and (b) shows the total average query processing time for the three groups of queries defined in Section 7.1.2 in the actor and Web networks. For $Random\ Q_{Neighbor}(G,u)$ and $Hub\ Q_{Neighbor}(G,u)$ the total execution time increases as $maxEdges$ decreases. This is because, if $maxEdge$ decreases, the edges of a node will be distributed in more buckets, and when executing a query, more buckets have to be retrieved from the server. $Hub\ Q_{Neighbor}(G,u)$ requires greater processing time than $Random\ Q_{Neighbor}(G,u)$, because hubs are nodes with many edges. In contrast, $Q_{Adjacency}(G,u,v)$ increases as $maxEdges$ takes higher values. The increase is due to the dummy edges inserted. Our experiments show that the number of dummy edges needed grows as $maxEdges$ increases.

$\boldsymbol{P_{CQprocessing}}$: For this part of the evaluation we only consider two groups of queries, $Random\ Q_{Neighbor}(G,u)$ and $Hub\ Q_{Neighbor}(G,u)$. We omit adjacency queries because they do not require any post-processing. See Figure 7.3(c) and (d). The time at the client increases as $maxEdges$ takes larger values.

From the experiments and analysis of $P_{TQprocessing}$ and $P_{CQprocessing}$, we can see that the best value to set $maxEdges$ is the growing parameter m. In scale-free networks, most nodes have a degree equal to m, so most buckets will be full after initialization. For the last experimental results, $P_{RQprocessing}$, we set $maxEdges$ to the best option, i.e., $maxEdges = m$.

$\boldsymbol{P_{RQprocessing}}$: In Figure 7.4, each two boxes of each plot show for each type of query executed, the total query processing time with our bucketization and the original graphs. The plots are for three kinds of queries, i.e., $Random\ Q_{Neighbor}(G,u)$, $Hub\ Q_{Neighbor}(G,u)$ and $Q_{Adjacency}(G,u,v)$. We deem the total execution time on the original graph the optimum. So we evaluate our approach depending on how much query processing time increases in comparison with the original graph. Regarding the actor network, $Random\ Q_{Neighbor}(G,u)$ with our bucketization approach is on average 3.44 times slower than with the original graph, $Hub\ Q_{Neighbor}(G,u)$ is 5.12 times slower and $Q_{Adjacency}(G,u,v)$ is 2.88 times slower. Regarding the Web graph, $Random\ Q_{Neighbor}(G,u)$ with our bucketization approach is 2.90 times slower than with the original graph on average, $Hub\ Q_{Neighbor}(G,u)$ is 10.15 times slower and $Q_{Adjacency}(G,u,v)$ 4.76 times. Except for $Hub\ Q_{Neighbor}(G,u)$, with our approach the query execution time is 3.5 times slower than with the original graph. In our opinion, this is a reasonable price for secrecy guarantees. So our bucketization approach is effective and feasible for graph-structured data secrecy.

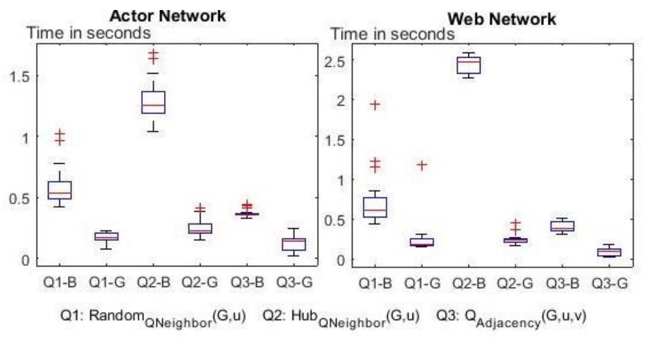

Figure 7.4. Total query processing time real datasets

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 [7] [23] [22], deal with different types of adversaries. In graph-structured data not only the node labels but also the edges of a node can reveal sensitive information. Therefore, our approach offers secrecy so that an adversary will not find out the edges and the degree of a node. 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 $\frac{11}{9}$ off the optimal solution. Next, to facilitate query planning, 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.

9. REFERENCES

[1] G Aggarwal et al. Two can keep a secret: A distributed architecture for secure database services. In *Conference on Innovative Data Systems Research (CIDR)*, 2005, pp. 186-199.

[3] A. L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, vol. 286, pp. 509-512, October 1999.

[4] J. Cao, F.-Y. Rao, M. Kuzu, E. Bertino, and M. Kantarcioglu. Efficient tree pattern queries on encrypted XML documents. In *Joint EDBT/ICDT 2013 Workshops*, ACM, 2013, pp. 111-120.

[5] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology-ASIACRYPT 2010*, Springer Berlin Heidelberg, 2010, pp. 577-594.

[6] R. Curmola, J. Garay, S. Kamara, and R. Ostrovsk. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, vol. 19, no. 5, pp. 895-934, January 2011.

[7] Z. Fan et al. Structure-Preserving subgraph query services. *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 8, pp. 2275-2290, August 2015.

[8] Michael Garey and David Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, 1st ed. United States of America: W. H. Freeman, 1979.

[9] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *2002 ACM SIGMOD international conference on Management of data*, ACM, 2002, pp. 216-227.

[10] H. Hacigümüs, B. Iyer, and S. Mehrotra. Query optimization in encrypted database systems. In *Database Systems for Advanced Applications*. Beijing, China: Springer Berlin Heidelberg, 2005, pp. 43-55.

[11] A. Hagberg, D. Schult, and P. Swart. Exploring network structure, dynamics, and function using NetworkX. In *7th Python in Science Conference (SciPy2008)*, Pasadena, CA USA, 2008, pp. 11-15.

[12] X. He, J. Viadya, B. Shafiq, N. Adam, and X. Lin. Reachability analysis in privacy-preserving perturbed graphs. In *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 2010, pp. 691-694.

[2] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *30th International Conference on Very Large Databases*, 2004, pp. 720-731.

[13] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Boca Raton, United States: Chapman & Hall, 2008.

[14] KONECT. (2016, October) Movies network dataset. [Online]. http://konect.uni-koblenz.de/networks/dbpedia-starring

[15] K. LeFevre, D. DeWitt, and R. Ramakrishnan. Mondrian multidimensional k-anonymity. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference*, 2006, p. 25.

[16] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster Sizes and the absence of large well-defined clusters. *Internet Mathematics*, vol. 6, no. 1, pp. 29--123, 2009.

[17] H. Masserrat and J. Pei. Neighbor query friendly compression of social networks. In *16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, DC, USA, 2010, pp. 533-542.

[18] X. Meng, S. Kamara, K. Nissim, and G. Kollios. GRECS: Graph encryption for approximate shortest distance queries. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2015, pp. 504- 517.

[19] G. S. Poh, M. S. Mohamad, and M. R. Z'aba. Structured encryption for conceptual graphs. In *Advances in Information and Computer Security*, Berlin, November 2012, pp. 105-122.

[20] V. V. Vazirani, *Approximation algorithms*. Springer Science&Business Media, 2013.

[21] J. W. and X. Du. A secure multi-dimensional partition based index in DAS. In *Asia-Pacific Web Conference*, Springer, 2008, pp. 319-330.

[22] H. Wang and L. Lakshmanan. Efficient secure query evaluation over encrypted XML databases. In *32nd international conference on Very large data bases*, 2006, pp. 127-138.

[23] Y. Zhang, S. Sen, W. Yulong, C. Weifeng, and Y. Fangchun. Privacy-assured substructure similarity query over encrypted graph-structured data in cloud. *Security and Communication Networks*, vol. 7, no. 11, pp. 1933-1944, 2014.