Top-k Query Processing on Encrypted Databases with Strong Security Guarantees

Xianrui Meng
Amazon Web Services, Inc.
Seattle, WA, USA
Email: xianru@amazon.com

Haohan Zhu
Facebook Inc.
Menlo Park, CA, USA
Email: zhuhaohan@fb.com

George Kollios
Boston University
Boston, MA, USA
Email: gkollios@cs.bu.edu

Abstract—**Concerns about privacy in outsourced cloud databases have grown recently and many efficient and scalable query processing methods over encrypted data have been proposed. However, there is very limited work on how to securely process top-k ranking queries over encrypted databases in the cloud. In this paper, we propose the first efficient and provably secure top-k query processing construction that achieves adaptive CQA security. We develop an encrypted data structure called EHL and describe several secure subprotocols under our security model to answer top-k queries. Furthermore, we optimize our query algorithms for both space and time efficiency. Finally, we empirically evaluate our protocol using real world datasets and demonstrate that our construction is efficient and practical.**

Keywords-**database security; data privacy; privacy-preserving query processing; top-k query;**

I. INTRODUCTION

As remote storage and cloud computing services emerge, such as Amazon's EC2, Google AppEngine, and Microsoft's Azure, many enterprises, organizations, and end users may outsource their data to those cloud service providers for reliable maintenance, lower cost, and better performance. In fact, a number of database systems on the cloud have been developed recently that offer high availability and flexibility at relatively low costs. However, due to security and privacy concerns [4], many users refrain from using these services, especially users with sensitive and valuable data. Indeed, data owners and clients may not fully trust a public cloud since a hacker or the cloud's administrator with root privileges can fully access all the data. Furthermore, the cloud provider may sell its business to an untrusted company, which will also have full access to the data. One approach to address these issues is to encrypt the data before outsourcing them to the cloud. Encrypted data can bring enhanced security into the Database-As-Service environment [22]. However, it also introduces significant difficulties in querying and computing over these data.

Although top-k queries are important query types in many database applications [25], to the best of our knowledge, none of the existing works handle top-k queries on encrypted data *efficiently*. Instead of encrypting the data, Vaiyda et. al. [39] studied privacy-preserving top-k queries in which the data are vertically partitioned. Wong et. al. [41] proposed an encryption scheme for k-Nearest-Neighbors (kNN) queries

and mentioned a method of transforming their scheme to solve top-k queries, however, as shown in [42], their encryption scheme is not secure and is vulnerable to chosen plaintext attacks. In this paper, we propose the first *Chosen-Query-Attack* (CQA) secure query processing scheme over *encrypted databases* that can answer top-k queries efficiently.

We assume that the data owner and the clients are trusted, but not the cloud server. Therefore, the data owner encrypts each database relation R using some probabilistic encryption scheme before outsourcing it to the cloud. An authorized user specifies a query q and generates a *token* to query the server. Our objective is to allow the cloud to securely compute the top-k results based on a user-defined ranking function over R, and, more importantly, the cloud should not learn anything about R or q. Consider a real world example for a health medical database below:

Example 1. *An authorized doctor, Alice, wants to get the top-k results based on some ranking criteria from the* encrypted *electronic health record database* `patients` *(Table I). The encrypted* `patients` *may contain several attributes; here we only list a few in Table I:* patient name, age, id number, trestbps[1], chol[2], thalach[3].

patient name	age	id	trestbps	chol	thalach
$E(Bob)$	$E(38)$	$E(121)$	$E(110)$	$E(196)$	$E(166)$
$E(Celvin)$	$E(43)$	$E(222)$	$E(120)$	$E(201)$	$E(160)$
$E(David)$	$E(60)$	$E(285)$	$E(100)$	$E(248)$	$E(142)$
$E(Emma)$	$E(36)$	$E(956)$	$E(120)$	$E(267)$	$E(112)$
$E(Flora)$	$E(43)$	$E(756)$	$E(100)$	$E(223)$	$E(127)$

Table I: Encrypted $patients$ Heart-Disease Data

One example of a top-k query (in the form of a SQL query) can be: `SELECT * FROM patients ORDERED BY chol+thalach STOP AFTER 2`. *That is, the doctor wants the top-2 results based on $chol + thalach$ from the patient records. However, since this table contains sensitive information about patients, the data owner first encrypts the table and then delegates it to the cloud. So, Alice requests a key from the data owner and generates a query* **token** *based on the query. Then the cloud computes on the encrypted table*

---

[1]trestbps: resting blood pressure (in mm Hg)
[2]chol: serum cholesterol in mg/dl
[3]maximum heart rate achieved

to find the encrypted top-2 results that returns to Alice. In this case, the top-2 results are the records of patients David and Emma.

Our protocol extends the No-Random-Access (NRA) [19] algorithm for computing top-k queries over a probabilistically encrypted relational database. Moreover, our query processing model uses two non-colluding semi-honest clouds, which is a model that has been used in some recent works as well (see [7, 10, 11, 18]). We encrypt the database in such a way that the server can obliviously execute NRA over the encrypted database without learning the underlying data. This is accomplished with the help of a secondary independent cloud server (or Crypto Cloud). However, the encrypted database resides only in the primary cloud. We adopt two efficient state-of-art secure protocols, EncSort [7] and EncCompare [10], which are the two main building blocks we need in our top-k secure construction. We choose these two protocols mainly because of their efficiency. In addition, we propose several novel sub-routines that can securely compute the best/worst score and de-duplicate replicated data items over the encrypted database. Notice that our proposed sub-protocols can also be used as stand-alone building blocks for other applications as well. We also would like to point out that during the querying phase the computation performed by the client is very small. The client only needs to compute a simple token for the server and after receiving the results, decrypt the encrypted records. All of the relatively heavier computations are performed by the cloud side.

Below we summarize our main contributions:

- We propose a new practical protocol designed to answer top-k queries over encrypted relational databases.
- We propose a new data structure called EHL which allows the servers to homomorphically evaluate the equality relation between two objects.
- We propose several independent sub-protocols such that the main server can securely compute the best/worst scores and de-duplicate replicated encrypted objects with the use of another non-colluding server.
- We prove that our scheme is CQA secure, extending the security definition from [13]. Both *data confidentiality* and *query privacy* are protected.
- The scheme is experimentally evaluated using real-world datasets and the results show that our scheme is efficient and practical.

II. RELATED WORK

The problem of processing queries over outsourced encrypted databases is not new. The seminal work of Hacigumus et al. [22] proposed executing SQL queries over encrypted data in the Database-As-Service model using bucketization. Since then, a number of works have appeared on executing various query types over encrypted data.

A significant amount of work has been devoted on keyword search queries or boolean queries, such as [9, 12, 15, 38]. Another work [37] proposed a general framework for boolean queries of disjunctive normal form queries on encrypted data. In addition, many works have been proposed for range queries [24, 29] and graph queries [32]. Other relevant works include privacy-preserving data mining [3, 27, 30, 40].

Recent works in the cryptography community have shown that it is possible to perform arbitrary computations over encrypted data using fully homomorphic encryption [20], or Oblivious RAM [21]. However, the performance overheads of such constructions are very high in practice; thus they're not suitable for practical database queries. Some recent advancements in ORAM [36] show promise and can be potentially used in certain environments. As mentioned, [39] is the only work that studied privacy preserving execution of top-k queries. However, their approach is mainly based on the k-anonymity privacy policies, therefore, it cannot be extended to encrypted databases. Recently, differential privacy [17] has emerged as a powerful model to protect against unknown adversaries with guaranteed probabilistic accuracy. However, here we consider *encrypted* data in the outsourced model; moreover, we do not want our query answer to be perturbed by noise, instead we want the results to be exact. [28] proposed a scheme that leverages DP and leaks obfuscated access statistics to enable efficient searching. Another approach has been extensively studied is order-preserving encryption [3, 35], which preserves the order of the message. We note that, by definition, OPE directly reveals the order of the objects' ranks, thus does not satisfy our data privacy guarantee. Furthermore, [23] proposed a prototype for access control using deterministic proxy encryption. Finally, other secure database systems have been proposed by using embedded secure hardware, such as TrustedDB [6] and Cipherbase [5].

Secure kNN queries. One of the most relevant problems is answering kNN (k Nearest Neighbor) queries. Note that top-k queries should not be confused with similarity search, such as kNN queries. For kNN queries, one is interested in retrieving the k most similar objects from the database to a query object, where the similarity between two objects is measured over some metric space, for instance using the L_2 metric. Many works have been proposed to specifically handle kNN queries on encrypted data, such as [14, 41, 42].

A recent work [18] proposed secure kNN query under the same architectural setting as ours. We would like to point out that their solution does not directly solve the problem of top-k queries. In particular, [18] designed a protocol for ranking distances between the query point and the records using the L_2 metric, while we consider the top-k selection query based on a class of scoring functions using linear combinations of attribute values. Nevertheless, if we follow the similar setup

from [18], we can define the scoring function to be the sum of the squares, i.e. $\sum x_i^2(o)$, where $x_i(o)$ is the i-th attribute value for object o and is a positive value. Then one can adapt the secure kNN scheme by querying a large enough query point (say, the upper bound of the attribute value) to get the k-nearest-neighbors and therefore it can return top-k results. We show in Section X-C, that even under this particular setting, our protocol is much more efficient than [18]. The computational complexity, for each query, for [18] is at least $O(nm)$, where n is the number of records in the database and m the number of attributes. Furthermore, the communication overhead between the two clouds is also $O(nm)$. Thus, this protocol is not very efficient for even small sized databases.

III. PRELIMINARIES

A. Problem Definition

Consider a data owner that has a database relation R of n objects, denoted by $o_1, ..., o_n$, and each object o_i has M attributes. For simplicity, we assume that all M attributes take numerical values. Thus, the relation R is an $n \times M$ matrix. The data owner would like to outsource R to a third-party cloud S_1 that is untrusted. Therefore, the data owner encrypts R and sends the encrypted relation ER to the cloud. After that, any authorized client should be able to get the results of a top-k query over this encrypted relation directly from S_1, by specifying k and a score function over the M (encrypted) attributes. We consider monotone scoring (ranking) functions that are weighted linear combinations over all the attributes, that is, $F_W(o) = \sum w_i \times x_i(o)$, where each $w_i \geq 0$ is a user-specified weight for the i-th attribute and $x_i(o)$ is the local score (value) of the i-th attribute for object o. Note that we consider monotone linear functions because they are the most important and widely used scoring functions for top-k queries [25]. The results of a top-k query are the objects with the highest k F_W values. For example, consider an authorized client, Alice, who wants to run a top-k query over the encrypted relation ER. Consider the following query: `q = SELECT * FROM ER ORDER BY` $F_W(\cdot)$ `STOP AFTER k;` That is, Alice wants to get the top-k results based on her scoring function F_W, for a specified set of weights. Alice first has to request the keys from the data owner, then generates a *query token* tk. Alice sends the tk to the cloud server. The cloud server storing the encrypted database ER processes the top-k query and sends the encrypted results back to Alice that she decrypts using the secret key. In the real world scenarios, the authorized clients can locally store the keys for generating the token and decryption.

B. Our Architecture and Security Model

We consider secure computations on the cloud under the *semi-honest* (or *honest-but-curious*) adversarial model.

Furthermore, our model assumes the existence of two *different* non-colluding semi-honest cloud providers, S_1 and S_2, where S_1 stores the encrypted database ER and S_2 holds some secret keys and provides crypto services. We refer to the server S_2 as the *Crypto Cloud* and we assume that it is isolated from S_1. The two parties S_1 and S_2 do not trust each other, and therefore, they have to execute secure computations on encrypted data. In fact, crypto clouds have been built and used in some industrial applications today (e.g., the pCloud Crypto[4] or boxcryptor[5]). This model is not new and has already been widely used in related work, such as [7, 10, 11, 18, 31]. We emphasize that these cloud services are typically provided by some large companies, such as Amazon, Google, and Microsoft, who have also commercial interests not to collude. The intuition behind such an assumption is as follows. Most of the cloud service providers in the market are well-established IT companies, such as Amazon AWS, Microsoft Azure and Google Cloud. Therefore, a collusion between them is highly unlikely as it will damage their reputation which effects their revenues. When S_1 receives the query token, S_1 initiates the secure computation protocol with S_2. Figure 1 shows an overview of the architecture.
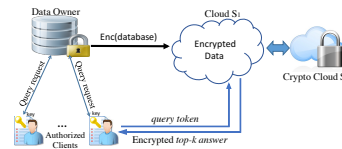

Figure 1: An overview of our model

Security Model. We adapt a security definition from the searchable encryption literature ([12, 13, 15]), which has been widely accepted in the prior privacy preserving database query works. In this paper, we capture the security requirement using a simulation-based security definition, which we refer as CQA (*Chosen-Query-Attack*) security, extending the definition from [13, 15]. We choose the CQA because it is more natural to our problem and better describes the security properties of our construction. Throughout the paper, the non-colluding clouds S_1 and S_2 are semi-honest adversarial servers. In our construction, S_1 and S_2 learn nothing about the data except a small amount of leakage that we explicitly describe. We give the details in Section VIII and the proof of security can be found in the full version of this paper [33]. We would like to emphasize that, during the execution of the query processing, neither of the servers S_1 or S_2 can retrieve the original data.

---

[4]https://www.pcloud.com/encrypted-cloud-storage.html
[5]https://www.boxcryptor.com/en/provider

C. Cryptographic Tools

Paillier Cryptosystem. The Paillier cryptosystem [34] is a semantically secure public key encryption scheme. The message space $\mathcal{M}$ for the encryption is $\mathbb{Z}_N$, where N is a product of two large prime numbers p and q. For a message $m \in \mathbb{Z}_N$, we denote $\mathsf{Enc}_{\mathsf{pk}}(m) \in \mathbb{Z}_{N^2}$ to be the encryption of m with the public key pk. When the key is clear in the text, we simply use $\mathsf{Enc}(m)$ to denote the encryption of m and $\mathsf{Dec}_{\mathsf{sk}}(c)$ to denote the decryption of a ciphertext c. The details of encryption and decryption algorithm can be found in [34]. It has the following homomorphic properties:

- *Addition:* $\forall x, y \in \mathbb{Z}_N$, $\mathsf{Enc}(x) \cdot \mathsf{Enc}(y) = \mathsf{Enc}(x + y)$
- *Scalar Multiplication:* $\forall x, a \in \mathbb{Z}_N$, $\mathsf{Enc}(x)^a = \mathsf{Enc}(a \cdot x)$

Damgård-Jurik cryptosystem. Our construction also relies on a Damgård-Jurik (DJ) cryptosystem introduced by Damgård and Jurik [16], which is a generalization of Paillier encryption. The message space $\mathcal{M}$ expands to $\mathbb{Z}_{N^s}$ for $s \geq 1$, and the ciphertext space is under the group $\mathbb{Z}_{N^{s+1}}$. As mentioned in [2], this generalization allows one to *doubly* encrypt messages and use the additive homomorphism of the inner encryption layer under the same secret key. In particular, let $\mathcal{E}^2(x)$ denote an encryption of the DJ scheme for a message $x \in \mathbb{Z}_{N^2}$ (when $s = 2$) and $\mathsf{Enc}(x)$ be a normal Paillier encryption.

This extension allows a ciphertext of the first layer to be treated as a plaintext in the second layer. Moreover, this nested encryption preserves the structure over inner ciphertexts and allows one to manipulate it as follows: $\mathcal{E}^2\big(\mathsf{Enc}(m_1)\big)^{\mathsf{Enc}(m_2)} = \mathcal{E}^2\big(\mathsf{Enc}(m_1) \cdot \mathsf{Enc}(m_2)\big) = \mathcal{E}^2\big(\mathsf{Enc}(m_1 + m_2)\big)$. We note that this is the only homomorphic property that our construction relies on. Throughout this paper, we use $\sim$ to denote that the plaintext under encryption Enc are the same, i.e., $\mathsf{Enc}(x) \sim \mathsf{Enc}(y) \Rightarrow x = y$. We summarize the notation throughout this paper in Table II. Note that in our application, we need one layered encryption; that is, given $\mathcal{E}^2\big(\mathsf{Enc}(x)\big)$, we want a normal Paillier encryption $\mathsf{Enc}(x)$. As introduced in [7], this could simply be done with the help of S_2. However, we need a protocol $\mathsf{RecoverEnc}$ to securely remove one layer of encryption.

D. No-Random-Access (NRA) Algorithm

The NRA algorithm [19] finds the top-k answers by exploiting only sorted accesses to the relation R. The input to the NRA algorithm is a set of sorted lists S, each ranks the "same" set of objects based on different attributes. The output is a ranked list of these objects ordered by the aggregate ranking scores. We opted to use this algorithm because it provides a scheme that leaks minimal information to the cloud server. We assume that each column (attribute) is sorted independently to create a set of sorted lists S. The set of sorted lists is equivalent to the original relation, but the objects in each list L are

Notation	Definition
n	Size of the relation R, i.e. $|R| = n$
M	Total number of attributes in R
m	Total number of attributes for the query q
$\mathsf{Enc}(m)$	Paillier encryption of m
$\mathsf{Dec}(c)$	Paillier decryption of c
$\mathcal{E}^2(m)$	Damgård-Jurik (DJ) encryption of m
$\mathsf{Enc}(x) \sim \mathsf{Enc}(y)$	Denotes $x = y$, i.e. $\mathsf{Dec}(\mathsf{Enc}(x)) = \mathsf{Dec}(\mathsf{Enc}(y))$
$\mathsf{EHL}(o)$	Encrypted Hash List of the object o
$\ominus, \odot$	EHL randomized operations, see Section V-A.
I_i^d	The data item in the ith sorted list L_i at depth d
$E(I_i^d)$	Encrypted data item I_i^d
$B^d(o)$	The best score (upper bound) of o at depth d
$W^d(o)$	The worst score (lower bound) of o at depth d

Table II: Notation Summarization

sorted in descending order according to their local scores (attribute values). After sorting, R contains M sorted lists, denoted as $S = \{L_1, L_2, \ldots, L_M\}$. Each sorted list consists of n data items, denoted as $L_i = \{I_i^1, I_i^2, \ldots, I_i^n\}$. Each data item is a object/value pair $I_i^d = (o_i^d, x_i^d)$, where o_i^d and x_i^d are the object id and local score at the depth d (when d objects have been accessed under sorted access in each list) in the ith sorted list respectively. Since NRA produces the top-k answers using upper/lower bounds it may not report the exact object scores. The score lower bound of some object o, $W(o)$, is obtained by applying the ranking function on o's known scores and the minimum possible values of o's unknown scores. The score upper bound of o, $B(o)$, is obtained by applying the ranking function on o's known scores and the maximum possible values of o's unknown scores, which are the same as the last seen scores in the corresponding ranked lists. The algorithm reports a top-k object even if its score is not precisely known. Specifically, if the score lower bound of an object o is not below the score upper bounds of all other objects (including unseen objects), then o can be safely reported as the next top-k object.

Security Remark. We choose to use NRA because it does not reveal the access patterns. It provides a scheme that leaks minimal information to the cloud server (since during query processing there is no need to access intermediate objects). We never access the actual records in our protocol and the user can decide how to access them at the end (we will discuss later).

IV. SCHEME OVERVIEW

In this section, we give an overview of our scheme.

Definition 2. *Let* $\mathsf{SecTopK} = (\mathsf{Enc}, \mathsf{Token}, \mathsf{SecQuery})$ *be the secure top-k query scheme containing three algorithms* Enc, Token *and* $\mathsf{SecQuery}$.

- $\mathsf{Enc}(\lambda, R)$: *is a probabilistic encryption algorithm that takes relation R and security parameter λ as its inputs and outputs encrypted relation ER and secret key K.*
- $\mathsf{Token}(K, q, k)$: *takes a secret key K, a query q, the parameter k for top-k, and outputs a token tk_q.*

- SecQuery(tk, ER) *is the query processing algorithm that takes* tk *and* ER *and securely computes top-k results based on the* tk.

As mentioned earlier, our encryption scheme takes advantage of the NRA algorithm. The idea of Enc is to encrypt and permute the set of sorted lists for R, so that the server can execute a variation of the NRA algorithm using only sequential accesses to the encrypted lists. To do this encryption, we design a new encrypted data structure, called EHL. The Token computes a token that serves as a 'trapdoor' so that the clouds know which list to access. In SecQuery, S_1 scans the encrypted lists depth by depth, for each targeted list, maintaining a list of encrypted objects per depth until there are k encrypted objects satisfying the NRA halting condition. During this process, S_1 and S_2 learn nothing about the underlying scores and objects. At the end of the protocol, the encrypted object ids can be reported to the client that can decrypt them. As we discuss next, there are two options after that. Either the records are retrieved and decrypted by the client, or the client retrieves the records using oblivious RAM [21] that does not even reveal the location of the actual encrypted records. In the first case, the server can get some additional information by observing the access patterns, i.e. the encryptions of different queries. However, there are schemes that address this access leakage [26, 28] and is beyond the scope of this paper. The second approach may be very expensive but is more secure.

V. Database Encryption

A. Encrypted Hash List (EHL)

We propose a new computation- and space- efficient data structure called encrypted hash list (EHL) to encrypt each object. The main purpose of this structure is to allow a server to homomorphically compute equality between two objects, whereas it is computationally hard for the server to figure out which these objects are. The intuition of designing the EHL is to use Pseudo-Random Function (PRF) F to first compute s 'secure hashes' of the object o and homomorphically encrypt them. Let $\mathsf{EHL}(o)$ be the encrypted list of an object o and let $\mathsf{EHL}(o)[i]$ denote the ith encryption in the list. In particular, we initialize an empty list EHL of length s. Then, we generate s secure keys $\kappa_1, ..., \kappa_s$ and object o is hashed to EHL as follows: for $1 \le i \le s$, let $\mathsf{EHL}(o)[i] = \mathsf{Enc}(F_{\kappa_i}(o))$.

Lemma 3. *Given two objects o_1 and o_2, their $\mathsf{EHL}(o_1)$ and $\mathsf{EHL}(o_2)$ are computationally indistinguishable.*

It is easy to see that Lemma 3 holds since the encrypted values in the EHL are encrypted by the semantically secure Paillier encryption. Given $\mathsf{EHL}(x)$ and $\mathsf{EHL}(y)$, we define the *randomized operation* $\ominus$ between $\mathsf{EHL}(x)$ and $\mathsf{EHL}(y)$ below:

$$\mathsf{EHL}(x) \ominus \mathsf{EHL}(y) \stackrel{\text{def}}{=} \prod_{i=0}^{s-1} \big(\mathsf{EHL}(x)[i] \cdot \mathsf{EHL}(y)[i]^{-1}\big)^{r_i} \quad (1)$$

where each r_i is some random value in $\mathbb{Z}_N$.

Lemma 4. *Let* $\mathsf{Enc}(b) = \mathsf{EHL}(x) \ominus \mathsf{EHL}(y)$, *then $b = 0$ if $x = y$ (two objects are the same), otherwise b is uniformly distributed in the group $\mathbb{Z}_N$ with high probability.*

Proof Sketch: If x and y are the same, then for all $i \in [s]$, $x_i = y_i$. Then,

$$\prod_{i=0}^{s-1} (\mathsf{EHL}(x)[i] \cdot \mathsf{EHL}(y)[i]^{-1})^{r_i} = \mathsf{Enc}\big(\sum_{i=0}^{s-1} (r_i(x_i - y_i)) \big) = \mathsf{Enc}(0)$$

If $x \ne y$, w.h.p. it must be true that there exists some $i \in [s]$ such that $F_{\kappa_i}(x) \ne F_{\kappa_i}(y)$, i.e. the plaintexts are different in $\mathsf{EHL}(x)[i]$ and $\mathsf{EHL}(y)[i]$. Therefore, $\big(\mathsf{EHL}(x)[i] \cdot \mathsf{EHL}(y)[i]^{-1}\big)^{r_i} = \mathsf{Enc}(r_i(F_{\kappa_i}(x) - F_{\kappa_i}(y))$. Hence, based on the definition $\ominus$, it follows that b becomes a random value uniformly distributed in the group $\mathbb{Z}_N$. ∎

False Positive Rate. Assuming F is a Pseudo-Random Function, the probability that the secure hashed values collide is at most $\frac{1}{N^s}$. Taking the union bound gives that the FPR is at most $\binom{n}{2}\frac{1}{N^s} \le \frac{n^2}{N^s}$. Notice that $N \approx 2^\lambda$ is large number as N is the product of two large primes p and q in the Paillier encryption and λ is the security parameter. For instance, if we set N to be a 1024 bit number and $s = 4$ or 5, then the FPR is negligible even for billions of records.

Remark. Note that we cannot simply encrypt the objects with Pallier encryption and then homomorphically evaluate whether they're equal or not. As we can see in the following sections, one of the servers needs to learn the equality bit. For example, if we homomorphically subtract the two objects, the difference reveals a lot of information besides the equality bit. The proposed EHL guarantee that we only reveal the equality bit if $x = y$ when we decrypt $\mathsf{EHL}(x) \ominus \mathsf{EHL}(y)$.

Notation. We introduce some notation that we use in our construction. Let $\mathbf{x} = (x_1, \ldots, x_s) \in \mathbb{Z}_N^s$ and let the encryption $\mathsf{Enc}(\mathbf{x})$ denotes the concatenation of $\mathsf{Enc}(x_1)...\mathsf{Enc}(x_s)$. Also, we denote by $\odot$ the block-wise multiplication between $\mathsf{Enc}(\mathbf{x})$ and $\mathsf{EHL}(y)$; that is, $\mathbf{c} \leftarrow \mathsf{Enc}(\mathbf{x}) \odot \mathsf{EHL}(y)$, where $\mathbf{c}_i \leftarrow \mathsf{Enc}(x_i) \cdot \mathsf{EHL}(y)[i]$ for $i \in [1, s]$.

B. Database Encryption

We describe the database encryption procedure Enc in this section. Given a relation R with M attributes, the data owner first encrypts the relation using Algorithm 1.

In ER, each data item $I_i^d = (o_i^d, x_i^d)$ at depth d in the sorted list L_i is encrypted as $E(I_i^d) = \langle \mathsf{EHL}(o_i^d), \mathsf{Enc}_{\mathsf{pk_p}}(x_i^d) \rangle$. As all the score has been encrypted under the public key $\mathsf{pk_p}$, for the rest of the paper, we simply use $\mathsf{Enc}(x)$ to denote $\mathsf{Enc}_{\mathsf{pk_p}}(x)$ under the public key $\mathsf{pk_p}$. Besides the size of the database and M, the encrypted ER doesn't reveal anything. In Theorem 5, we demonstrate this by showing that two encrypted databases are indistinguishable if they have the same size and number of attributes.

Algorithm 1: Database encryption

1. Generate a public/secret key pk_p, sk_p for the Paillier encryption and secret keys $\kappa_1, \ldots, \kappa_s$ for EHL;
2. Given the relation R, sort each L_i in descending order based on the attribute's value for $1 \leq i \leq M$;
3. Do sorted access in parallel to each of the M sorted lists L_i;
4. **foreach** *data item* $I_i = \langle o_i^d, x_i^d \rangle \in L_i$ **do**
5.   **foreach** *depth* d **do**
6.   Compute $\mathsf{EHL}(o_i^d)$ using the keys $\kappa_1, \ldots, \kappa_s$;
7.   Compute $\mathsf{Enc}_{pk_p}(x_i^d)$ using pk_p;
8.   Store $E(I_i^d) = \langle \mathsf{EHL}(o_i^d), \mathsf{Enc}_{pk_p}(x_i^d) \rangle$ at depth d;
9. Generate a secret key k_p for a pseudorandom permutation P. For $1 \leq i \leq M$, permute L_i as $L_{P_{k_p}(i)}$;
10. Finally, output all lists of permuted encrypted items as the encrypted relation as ER;
11. The data owner securely uploads the keys pk_p, sk_p to the S_2, and only pk_p to S_1. Furthermore, it securely distribute the secrete keys (sk_p, k_p and $\kappa_1, \ldots, \kappa_s$) to its authorized client.

Theorem 5. *Given two relations R_1 and R_2 with $|R_1| = |R_2|$ and same number of attributes. The ER_1 and ER_2 output by the algorithm Enc are indistinguishable.*

It's easy to see that the theorem holds based on Lemma 3 and the Paillier encryption scheme.

VI. QUERY TOKEN

Consider the SQL-like query `q = SELECT * FROM ER ORDERED BY` $F_W(\cdot)$ `STOP BY k`, where $F_W(\cdot)$ is a weighted linear combination of all attributes. In this paper, to simplify our presentation of the protocol, we consider binary weights and therefore the scoring function is just a sum of the values of a subset of attributes. However, notice that for non $\{0,1\}$ weights the client should provide these weights to the server and the server can simply adapt the same techniques by using the scalar multiplication property from Paillier before it performs the rest of the protocol which we discuss next. On input the key k_p and query q, the Token algorithm is quite simple and works as follows: the client specifies the scoring attribute set M of size m, i.e. $|\mathsf{M}| = m \leq M$, then requests the key k_p from the data owner, where k_p is the key corresponds to the Pseudorandom Permutation P. Then the client computes the $P_{k_p}(i)$ for each $i \in \mathsf{M}$ and sends the following query token to the cloud server S_1: $tk_q =$ `SELECT * FROM ER ORDERED BY` $\{P_{k_p}(i)\}_{i \in \mathsf{M}}$ `STOP BY k`.

VII. TOP-K QUERY PROCESSING

As mentioned, our query processing protocol is based on the NRA algorithm. However, the technical difficulty is to execute the algorithm on the encrypted data while S_1 does not learn any object id or any score and attribute value of the data. We incorporate several cryptographic protocols to achieve this. Our query processing uses two state-of-the-art efficient and secure protocols: EncSort introduced by [7]

Algorithm 2: Top-k Query Processing: SecQuery

1. S_1 receives tk_q from the client, parses tk_q and let $L_i = L_{P_K(j)}$ for $j \in \mathsf{M}$;
2. **foreach** *depth d at each list* **do**
3.   **foreach** $E(I_i^d) = \langle \mathsf{EHL}(o_i^d), \mathsf{Enc}(x_i^d) \rangle \in L_i$ **do**
4.   Compute $\mathsf{Enc}(W_i^d)) \leftarrow \mathsf{SecWorst}(E(I_i^d), H, pk_p, sk_p)$, where $H = \{E(I_j^d)\}_{j \in m, i \neq j}$;
5.   Compute $\mathsf{Enc}(B_i^d)) \leftarrow \mathsf{SecBest}(E(I_i^d), \{j\}_{j \neq i}, pk_p, sk_p)$;
6.   Run $\Gamma_d \leftarrow \mathsf{SecDedup}(\{E(I_i^d)\}, pk_p, sk_p)$ with S_2 and get the local encrypted list Γ_d;
7.   Run $T^d \leftarrow \mathsf{SecUpdate}(T^{d-1}, \Gamma_d, pk_p, sk_p)$ with S_2 and get T^d;
8.   If $|T^d| < k$ elements, go to the next depth. Otherwise, run $\mathsf{EncSort}(T^d)$ by sorting on $\mathsf{Enc}(W_i)$, get first k items as T_k^d;
9.   Let the kth and the $(k+1)$th item be $E(I_k')$ and $E(I_{k+1}')$, S_1 then runs $f \leftarrow \mathsf{EncCompare}(E(W_k'), E(B_{k+1}'))$ with S_2, where $E(W_k')$ is the worst score for $E(I_k')$, and $E(B_{k+1}')$ is the best score for $E(I_{k+1}')$ in T^d;
10.   **if** $f = 0$ **then**
11.   **Halt** and return the encrypted first k item in T_k^d

---

and EncCompare introduced by [10] as building blocks. We skip the detailed description of these two protocols since they are not the focus of this paper. Here we only describe their functionalities:

1). EncSort: S_1 has a list of encrypted keyed-value pairs $(\mathsf{Enc}(k_1), \mathsf{Enc}(a_1))...(\mathsf{Enc}(k_m), \mathsf{Enc}(a_m))$ and a public key pk, and S_2 has the secret key sk. At the end, S_1 obtains a list *new* encryptions $(\mathsf{Enc}(k_1'), \mathsf{Enc}(a_1'))...(\mathsf{Enc}(k_m'), \mathsf{Enc}(a_m'))$, where the key/value list is sorted based on the order $a_1' \leq a_2'... \leq a_m'$ and the set $\{(k_1, a_1), ..., (k_m, a_m)\}$ is the same as $\{(k_1', a_1'), ..., (k_m', a_m')\}$.

2). EncCompare($\mathsf{Enc}(a), \mathsf{Enc}(b)$): S_1 has a public key pk and two encrypted values $\mathsf{Enc}(a), \mathsf{Enc}(b)$, while S_2 has the secret key sk. At the end of the protocol, S_1 obtains the bit f such that $f := (a < b)$. Several protocols have been proposed for the functionality above. We choose the one from [10] mainly because it is efficient and perfectly suits our requirements.

A. Query Processing: SecQuery

Notations. We denote each *encrypted item* by $E(I) = \langle \mathsf{EHL}(o), \mathsf{Enc}(x) \rangle$, where I is the item with object id o and score x. During the query processing, the server S_1 needs to maintain the encrypted item with its current best or worst scores, and we denote by $\mathbf{E}(I) = (\mathsf{EHL}(o), \mathsf{Enc}(W), \mathsf{Enc}(B))$ the *encrypted score item* I with object id o with best score B and worst score W. We first give the overall description of the top-k query processing SecQuery at a high level. Then in Section VII-B, we describe in details the secure sub-routines that we use in the query processing: SecWorst, SecBest, SecDedup, and SecUpdate.

As mentioned, SecQuery makes use of the NRA algorithm but is different from the original NRA, because SecQuery cannot maintain the global worst/best scores in plaintext. Instead, SecQuery has to run secure protocols depth by depth and homomorphically compute the worst/best scores based on the items at each depth. In particular, upon receiving

	R_1		R_2		R_3		T^1 Score List		
								Lower Bound	Upper Bound
depth 1	EHL(X_1)	Enc(10)	EHL(X_2)	Enc(8)	EHL(X_4)	Enc(8)			
	EHL(X_2)	Enc(8)	EHL(X_3)	Enc(7)	EHL(X_3)	Enc(6)	EHL(X_1)	Enc(10)	Enc(26)
	EHL(X_3)	Enc(5)	EHL(X_1)	Enc(3)	EHL(X_1)	Enc(2)	EHL(X_2)	Enc(8)	Enc(26)
	EHL(X_4)	Enc(3)	EHL(X_4)	Enc(2)	EHL(X_5)	Enc(1)	EHL(X_4)	Enc(8)	Enc(26)
	EHL(X_5)	Enc(1)	EHL(X_5)	Enc(1)	EHL(X_2)	Enc(0)			

(a) SecWorst & SecBest at depth 1. T^1 maintains the encrypted scores after depth 1.

	R_1		R_2		R_3		T^2 Score List		
								Lower Bound	Upper Bound
depth 2	EHL(X_1)	Enc(10)	EHL(X_2)	Enc(8)	EHL(X_4)	Enc(8)	EHL(X_2)	Enc(16)	Enc(22)
	EHL(X_2)	Enc(8)	EHL(X_3)	Enc(7)	EHL(X_3)	Enc(6)	EHL(X_3)	Enc(13)	Enc(21)
	EHL(X_3)	Enc(5)	EHL(X_1)	Enc(3)	EHL(X_1)	Enc(2)	EHL(X_1)	Enc(10)	Enc(23)
	EHL(X_4)	Enc(3)	EHL(X_4)	Enc(2)	EHL(X_5)	Enc(1)	EHL(X_4)	Enc(8)	Enc(23)
	EHL(X_5)	Enc(1)	EHL(X_5)	Enc(1)	EHL(X_2)	Enc(0)	EHL(R_2)	Enc(Z)	Enc(22)
							EHL(R_3)	Enc(Z)	Enc(23)

(b) SecWorst & SecBest at depth 2. T^2 maintains the sorted encrypted scores based on their worst scores after depth 2. Note that, after SecDedup, the duplicated objects X_1, X_2 do not appear in the top-k list twice.

	R_1		R_2		R_3		T^3 Score List		
								Lower Bound	Upper Bound
depth 3	EHL(X_1)	Enc(10)	EHL(X_2)	Enc(8)	EHL(X_4)	Enc(8)	EHL(X_3)	Enc(18)	Enc(18)	top-2
	EHL(X_2)	Enc(8)	EHL(X_3)	Enc(7)	EHL(X_3)	Enc(6)	EHL(X_2)	Enc(16)	Enc(18)
	EHL(X_3)	Enc(5)	EHL(X_1)	Enc(3)	EHL(X_1)	Enc(2)	EHL(X_1)	Enc(15)	Enc(15)
	EHL(X_4)	Enc(3)	EHL(X_4)	Enc(2)	EHL(X_5)	Enc(1)	EHL(X_4)	Enc(8)	Enc(16)
	EHL(X_5)	Enc(1)	EHL(X_5)	Enc(1)	EHL(X_2)	Enc(0)	EHL(R_2)	Enc(Z)	Enc(22)
							EHL(R_3)	Enc(Z)	Enc(23)

(c) SecWorst & SecBest at depth 3. T^3 maintains the sorted encrypted scores based on their worst scores after depth 3 . The SecQuery halts (based on line 11) in Algorithm 2
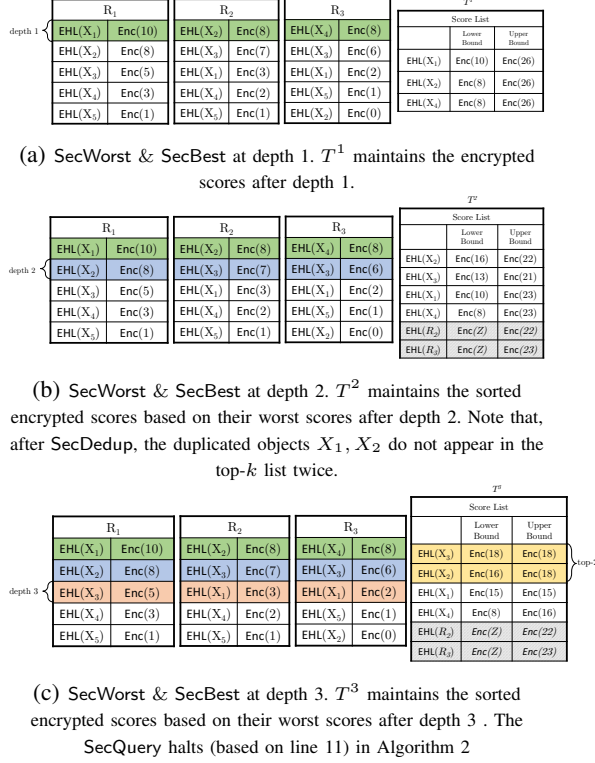
Figure 2: An example of securely computing the top-2 query for SecQuery. The table has three attributes, and the score function f is the sum of all the attributes.

the query token tk $=$ SELECT $\star$ FROM E(R) ORDERED BY $\{P_K(i)\}_{i \in M}$ STOP BY k, S_1 begins to process the query. The token tk contains $\{P_K(i)\}_{i \in M}$ which informs S_1 which lists to perform the sequential access. By maintaining an encrypted list T, which includes items with their encrypted global best and worst scores, S_1 updates the list T depth by depth. Let T^d be the state of the encrypted list T after depth d. At depth d, S_1 first computes the local encrypted worst/best scores for each item appearing at this depth by running SecWorst and SecBest. Then S_1 securely replaces the duplicated encrypted objects with large encrypted worst scores Z by running SecDedup. Next, S_1 updates the encrypted global list from state T^{d-1} to state T^d by adapting SecUpdate. S_1 utilizes EncSort to sort the distinct encrypted objects with their scores in T^d to obtain the first k encrypted objects which are essentially the top-k objects based on their worst scores so far. The protocol halts if at some depth, the encrypted best score of the $(k+1)$-th object, $\text{Enc}(B_{k+1})$, is less than the k-th object's encrypted worst score $\text{Enc}(W_k)$. This can be checked by calling the protocol $\text{EncCompare}(\text{Enc}(W_k), \text{Enc}(B_{k+1}))$. We describe the detailed query processing in Algorithm 2.

B. Building Blocks

In this section, we present the detailed description of the protocols SecWorst, SecBest, SecDedup, and SecUpdate.

1) Secure Worst Score: At each depth, for each encrypted data item, server S_1 should obtain the encryption $\text{Enc}(W)$, which is the *worst score* (lower bound) based on the items at the current depth *only*. Note that this is different than the normal NRA algorithm as it computes the global worst possible score for each encountered objects until the current depth. We formally describe the protocol setup below:

Protocol SecWorst. *Server S_1 has the input $E(I) = \langle \text{EHL}(o), \text{Enc}(x) \rangle$, a set of encrypted items H, i.e. $H = \{E(I_i)\}_{i=[|H|]}$, where $E(I_i) = \langle \text{EHL}(o_i), \text{Enc}(x_i) \rangle$, and the public key pk_p. Server S_2's inputs are pk_p and sk_p. SecWorst securely computes the encrypted worst ranking score based on L, i.e., S_1 outputs $\text{Enc}(W(o))$, where $W(o)$ is the worst score based on the list H.*

The technical challenge here is to homomorphically evaluate the encrypted score only based on the objects' equality relation. We present the detailed protocol description of SecWorst in Algorithm 3.

Example 6. *(Figure 2a) At depth 1, to compute the worst score (lower bound) for X_1, SecWorst takes the encryptions $\text{Enc}(8)$, $\text{Enc}(8)$ at the same depth from columns R_2 and R_3 and finally outputs the $\text{Enc}(10)$ as 10 is the lower bound for X_1 so far after depth 1.*

Algorithm 3: $\text{SecWorst}(E(I), H = \{E(I_i)\}_{i \in [|H|]}, \text{pk}_p, \text{sk}_p)$

S_1's input: $E(I)$, $H = \{E(I_j)\}$, pk_p
S_2's input: pk_p, sk_p

1   **Server S_1:**
2   Let $|H| = m$. Generate a random permutation $\pi : [m] \to [m]$;
3   For the set of encrypted items $H = \{E(I_j)\}$, permute each $E(I_j)$ in H as $E(I_{\pi(j)}) = \text{EHL}(o_{\pi(j)}), \text{Enc}(x_{\pi(j)})$.;
4   **for** *each permuted item in $E(I_{\pi(j)})$* **do**
5   compute $\text{Enc}(b_j) \leftarrow \text{EHL}(o) \ominus \text{EHL}(o_{\pi(j)})$, send $\text{Enc}(b_j)$ to S_2
6   Receive $\mathcal{E}^2(t_i)$ from S_2 and evaluate:
$$\mathcal{E}^2(\text{Enc}(x_i')) := \mathcal{E}^2(t_i)^{\text{Enc}(x_i)} \cdot \left(\mathcal{E}^2(1)\mathcal{E}^2(t_i)^{-1}\right)^{\text{Enc}(0)};$$
7   Run $\text{Enc}(x_i') \leftarrow \text{RecoverEnc}(\mathcal{E}^2(\text{Enc}(x_i')))$;
8   Set the worst score $\text{Enc}(W) \leftarrow (\prod_{i=1}^m \text{Enc}(x_i'))$;
9   Output $\text{Enc}(W)$.
10   **Server S_2:**
11   **for** *each $\text{Enc}(b_i)$ received from S_1* **do**
12   Decrypt to get b_i, set $t_i \leftarrow (b_i = 0 ? 1 : 0)$;
13   Send $\mathcal{E}^2(t_i)$ to S_1.
14   **Procedure** $\text{RecoverEnc}(\mathcal{E}^2(\text{Enc}(c)))$
15   **Server S_1:**
16   Generate $r \xleftarrow{\$} \mathbb{Z}_N$, compute and send
$\mathcal{E}^2(\text{Enc}(c + r)) \leftarrow \mathcal{E}^2(\text{Enc}(c))^{\text{Enc}(r)}$ to S_2.
17   **Server S_2:**
18   Decrypt as $\text{Enc}(c + r)$ using sk_p and send back to S_1
19   **Server S_1:**
20   Receive $\text{Enc}(c + r)$, compute $\text{Enc}(c) = \text{Enc}(c + r) \cdot \text{Enc}(r)^{-1}$, and output $\text{Enc}(c)$.

---

2) Secure Best Score: The secure computation for the best score is different from computing the worst score. Below we describe the protocol SecBest between S_1 and S_2, and Algorithm 4 describes the protocol in detail.

Protocol SecBest. *Server S_1 takes the inputs of the public key $\mathsf{pk_p}$, $E(I) = \langle \mathsf{EHL}(o), \mathsf{Enc}(x) \rangle$ for the object o in list L_i, and a set of pointers $\mathcal{P} = \{j\}_{i \neq j, j \in \mathsf{M}}$ to the list in ER. The protocol SecBest securely computes the encrypted best score at the current depth d, i.e., S_1 finally outputs $\mathsf{Enc}(B(o))$, where $B(o)$ is the best score for the o at current depth.*

Example 7. *(Figure 2b) At depth 2, to compute the best score (upper bound) for X_4, SecBest takes the encryptions seen so far, then based on the scores it outputs $\mathsf{Enc}(23)$ as 23 is the upper bound for X_4 after depth 2.*

Algorithm 4: $\mathsf{SecBest}(E(I_i), \mathcal{P}, \mathsf{pk_p}, \mathsf{sk_p})$ Secure Best Score.

S_1's input: $E(I_i)$ in list L_i, $\mathcal{P} = \{j\}_{i \neq j}$, $\mathsf{pk_p}$
S_2's input: $\mathsf{pk_p}$, $\mathsf{sk_p}$

1 **Server S_1:**
2   **foreach** *list L_i* **do**
3   maintain $\mathsf{Enc}(\underline{x}_i^d)$ for L_i, where $\mathsf{Enc}(\underline{x}_i^d)$ is the encrypted score at depth d.
4   Let $l = |\{L_i\}|$, i.e. the size of the list, then generate a random permutation $\pi : [l] \to [l]$;
5   Permute each L_i as $L_{\pi(i)} = \mathsf{EHL}(o_{\pi(i)}), \mathsf{Enc}(x_{\pi(i)})$;
6   **foreach** *permuted $E(I_{\pi(i)})$* **do**
7   compute $\mathsf{Enc}(b_i) \leftarrow \mathsf{EHL}(o) \ominus \mathsf{EHL}(o_i)$
8   send $\mathsf{Enc}(b_i)$ to S_2 receive $\mathcal{E}^2(t_i)$ and compute:
$$\mathcal{E}^2(\mathsf{Enc}(x_i')) := \mathcal{E}^2(t_i)^{\mathsf{Enc}(x_i)} \cdot \left(\mathcal{E}^2(1) \mathcal{E}^2(t_i)^{-1} \right)^{\mathsf{Enc}(0)};$$
9   run $\mathsf{Enc}(x_i') \leftarrow \mathsf{RecoverEnc}(\mathcal{E}^2(\mathsf{Enc}(x_i')))$ with S_2;
10   compute $\mathcal{E}^2(\mathsf{Enc}(\underline{x}_i'^d)) \leftarrow (1 - \prod_{i=1}^d \mathcal{E}^2(t_i))^{\mathsf{Enc}(\underline{x}_i^d)}$;
11   run $\mathsf{Enc}(\underline{x}_i'^d) \leftarrow \mathsf{RecoverEnc}(\mathcal{E}^2(\mathsf{Enc}(\underline{x}_i'^d)))$ with S_2;
12   set $\mathsf{Enc}(B_i) \leftarrow \mathsf{Enc}(\underline{x}_i'^d) \cdot (\prod_{i=1}^l \mathsf{Enc}(x_i'))$;
13   compute $\mathsf{Enc}(B) \leftarrow \prod_{i=1}^m \mathsf{Enc}(B_i)$ and output $\mathsf{Enc}(B)$;
14 **Server S_2:**
15   **for** $\mathsf{Enc}(b_i)$ *received from S_1* **do**
16   Decrypt to get b_i. If $b_i = 0$ set $t_i = 1$, otherwise $t_i = 0$. Send $\mathcal{E}^2(t_i)$ to S_1.

---

3) Secure Deduplication: At each depth, some of the objects might be repeatedly computed since the same objects may appear in different sorted list at the same depth. S_1 cannot identify duplicates since the items and their scores are probabilistically encrypted. We now present a protocol that deduplicates the encrypted objects in the following, and, due to space limit, the detailed protocol SecUpdate can be found in [33].

Protocol SecDedup. *Let the $\boldsymbol{E}(I)$ be an encrypted scored item such that $\boldsymbol{E}(I) = (\mathsf{EHL}(o), \mathsf{Enc}(W), \mathsf{Enc}(B))$, i.e. the $\boldsymbol{E}(I)$ is associated with $\mathsf{EHL}(o_i)$, its encrypted worst and best score $\mathsf{Enc}(W_i)$, $\mathsf{Enc}(B_i)$. S_1's inputs are public key $\mathsf{pk_p}$, and a set of encrypted scored items $Q = \{\boldsymbol{E}(I_i)\}_{i \in [\ell]}$, where $\ell = |Q|$. At the end of the protocol, S_1 outputs a new list of items $\boldsymbol{E}(I_1'), ..., \boldsymbol{E}(I_\ell')$, and there does not exist $i, j \in [\ell]$ with $i \neq j$ such that $o_i = o_j$. Moreover, the new encrypted list should not affect the final top-k results.*

Example 8. *(Fig 2b) After scanning depth 2, SecDedup deduplicates the repeated objects in the list T^2. X_1 and X_2 are the repeated objects. SecDedup replaces those the objects with random ids R_1 and R_2 and replaces the worst scores with large number Z so that they do not appear in the top-2 list.*

4) Secure Update: At each depth d, we need to update the current list of objects with the latest global worst/best scores. At a high level, S_1 has to update the encrypted list Γ^d from the state T^{d-1} (previous depth) to T^d, and appends the new encrypted items at this depth. Let Γ^d be the list of encrypted items with the encrypted worst/best scores S_1 get at depth d. Specifically, for each encrypted item $E(I_i) \in T^{d-1}$ and each $E(I_j) \in \Gamma^d$ at depth d, we update I_i's worst score by adding the worst from I_j and replace its best score with I_j's best score if $I_i = I_j$ since the worst score for I_j is the in-depth worst score and best score for I_j is the most updated best score. If $I_i \neq I_j$, we then simply append $E(I_j)$ with its scores to the list. Finally, we get the fresh T^d after depth d. Due to space limit, the detailed protocol SecUpdate is described in [33].

Complexity analysis. We analyze the efficiency of query execution. Suppose the client chooses m attributes for the query, therefore at each depth there are m objects. At depth d, it takes S_1 $O(m)$ for executing SecWorst, $O(md)$ for executing SecBest, $O(m^2)$ for SecDedup, and $O(m^2 d)$ for the SecUpdate. The complexities for S_2 are similar. In addition, the EncSort has time overhead $O(m \log^2 m)$; however, we can further reduce to $O(\log^2 m)$ by adapting parallelism (see [7]). On the other hand, the SecDupElim only takes $O(u^2)$, where u is the number of distinct objects at this depth. Notice that most of the computations are multiplication (homomorphic addition), therefore, the cost of query is relatively small.

VIII. Security

The following describes the high level intuition of our security definition for querying encrypted databases:

Informal Definition. *No efficient adversary can learn any partial information about the data or the queries, beyond what is explicitly allowed by the* leakage functions*, even for queries that are adversarially-influenced and generated adaptively.*

Since our construction supports a more complex query type than searching, the security has to capture the fact that the adversarial servers also get the 'views' from the data and meta-data during the query execution. Formally, the CQA security model in our scheme defines a **Real** and an **Ideal** world. In the real world, the protocol between the adversarial servers and the client executes just like the SecTopK scheme. In the ideal world, we assume that there exists two simulator Sim_1 and Sim_2 who get the leakage profiles from an ideal functionality and try to simulate the execution for the real world. We say the scheme is CQA secure if, after polynomial many queries, no ppt distinguisher can distinguish between the two worlds only with non-negligibly probability. Formally, see Definition 9.

Definition 9. *Consider the scheme* SecTopK $=$ (Enc, Token, SecQuery) *and the following probabilistic*

experiments where $\mathcal{E}$ is an environment, $\mathcal{C}$ is a client, S_1 and S_2 are two non-colluding semi-honest servers, Sim_1 and Sim_2 are two simulators, and $\mathcal{L}_{\mathsf{Setup}}$, $\mathcal{L}_{\mathsf{Query}} = (\mathcal{L}^1_{\mathsf{Query}}, \mathcal{L}^2_{\mathsf{Query}})$ are (stateful) leakage functions:

Ideal(1^λ): $\mathcal{E}$ outputs a relation R of size n and sends to the client $\mathcal{C}$. $\mathcal{C}$ submits R to $\mathcal{F}_{\mathsf{topk}}$, i.e. the **ideal** top-k functionality. $\mathcal{F}_{\mathsf{topk}}$ outputs $\mathcal{L}_{\mathsf{Setup}}(R)$ and 1^λ, and gives $\mathcal{L}_{\mathsf{Setup}}(R)$, 1^λ to Sim_1. Given $\mathcal{L}_{\mathsf{Setup}}(R)$ and 1^λ, Sim_1 generates an encrypted ER. $\mathcal{C}$ adaptively generates poly. number of queries $(q_1, \ldots, q_m)$. For each q_i, $\mathcal{C}$ submits q_i to $\mathcal{F}_{\mathsf{topk}}$, $\mathcal{F}_{\mathsf{topk}}$ sends $\mathcal{L}^1_{\mathsf{Query}}(\mathsf{ER}, q_i)$ to Sim_1 and $\mathcal{L}^2_{\mathsf{Query}}(\mathsf{ER}, q_i)$ to Sim_2. Finally, $\mathcal{C}$ outputs $\mathsf{OUT}'_{\mathcal{C}}$, Sim_1 outputs $\mathsf{OUT}_{\mathsf{Sim}_1}$, Sim_2 outputs $\mathsf{OUT}_{\mathsf{Sim}_2}$.

Real$_\mathcal{A}(1^\lambda)$: $\mathcal{E}$ outputs R of size n and sends to $\mathcal{C}$. $\mathcal{C}$ computes $(K, \mathsf{ER}) \leftarrow \mathsf{Enc}(1^\lambda, R)$ and sends the encrypted ER to S_1. $\mathcal{C}$ adaptively generates a poly number of queries $(q_1, \ldots, q_m)$. For each q_i, $\mathcal{C}$ computes $\mathsf{tk}_i \leftarrow \mathsf{Token}(K, q_i)$ and sends tk_i to S_1. S_1 runs $\mathsf{SecQuery}(\mathsf{tk}_i, \mathsf{ER})$ with S_2. Finally, S_1 sends the encrypted results to $\mathcal{C}$. $\mathcal{C}$ outputs $\mathsf{OUT}_{\mathcal{C}}$, S_1 outputs OUT_{S_1}, S_2 outputs OUT_{S_2}. We say that $\mathsf{SecQuery}$ is adaptively $(\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}})$-secure against Chosen Query Attack (CQA) if the following holds:

1) For all $\mathcal{E}$, for all S_1, there exists a ppt simulator Sim_1 such that the following two distributions are computationally indistinguishable

$$\langle \mathsf{OUT}_{S_1}, \mathsf{OUT}_{\mathcal{C}} \rangle \cong \langle \mathsf{OUT}_{\mathsf{Sim}_1}, \mathsf{OUT}'_{\mathcal{C}} \rangle$$

2) For all $\mathcal{E}$, for all S_2, there exists a ppt simulator Sim_2 such that the following two distributions are computationally indistinguishable

$$\langle \mathsf{OUT}_{S_2}, \mathsf{OUT}_{\mathcal{C}} \rangle \cong \langle \mathsf{OUT}_{\mathsf{Sim}_2}, \mathsf{OUT}'_{\mathcal{C}} \rangle$$

We formally define the leakage in $\mathsf{SecTopK}$. Let the setup leakage $\mathcal{L}_{\mathsf{Setup}} = (|R|, |M|)$. $\mathcal{L}_{\mathsf{Setup}}$ is the leakage profile revealed to S_1 after the execution of Enc. During query processing, we allow $\mathcal{L}_{\mathsf{Query}} = (\mathcal{L}^1_{\mathsf{Query}}, \mathcal{L}^2_{\mathsf{Query}})$ revealed to the servers. Note that $\mathcal{L}^1_{\mathsf{Query}}$ is the leakage function for S_1, while $\mathcal{L}^2_{\mathsf{Query}}$ is the leakage function for S_2. In our scheme, $\mathcal{L}^1_{\mathsf{Query}} = (\mathsf{QP}, D_q)$, where QP is the *query pattern* indicating whether a query has been repeated or not, and D_q is the halting depth for query q. For any query q, we define the equality pattern as follows: suppose that there are m number of objects at each depth,

- *Equality pattern* $\mathsf{EP}_d(q)$: a symmetric binary $m \times m$ matrix M^d, where $M^d[i, j] = 1$ if there exist $o_{\pi(i')} = o_{\pi(j')}$ for some random permutation π such that $\pi(i') = i$ and $\pi(j') = j$, otherwise $M^d[i, j] = 0$.

Then, let $\mathcal{L}^2_{\mathsf{Query}} = (\{\mathsf{EP}_d(q)\}_{i=1}^{D_q})$, i.e. at depth $d \leq D_q$ the equality pattern indicates the number of equalities between objects. Note that $\mathsf{EP}^d(q)$ does not leak the equality relations between objects at any depth in R, i.e. the server never know which objects are same.

Theorem 10. *Suppose the function in* EHL *is a pseudorandom function and Paillier encryption is CPA-secure, then the scheme* $\mathsf{SecTopK} = (\mathsf{Enc}, \mathsf{Token}, \mathsf{SecQuery})$ *we proposed is* $(\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}})$*-CQA secure.*

We skip the proof of Theorem 10 due to space limit. The proof can be found in the full version of this paper [33].

IX. OPTIMIZATIONS

In this section, we present some optimizations that improve the performance of our protocol. The optimizations are two-fold: 1) we optimize the efficiency of the protocol $\mathsf{SecDedup}$ at the expense of some additional privacy leakage, and 2) we propose batch processing of $\mathsf{SecDupElim}$ and $\mathsf{EncSort}$ to further improve the $\mathsf{SecQuery}$.

SecDupElim The idea for $\mathsf{SecDupElim}$ is that instead of keeping the same number encrypted items m, $\mathsf{SecDupElim}$ *eliminates* the duplicated objects. In this way, the number of encrypted objects gets reduced, especially if there are many duplicated objects.

Now by adapting $\mathsf{SecDupElim}$, if there are many duplicated objects appear in the list, we have much fewer encrypted items to sort. The $\mathsf{SecDupElim}$ leaks additional information to the server S_1. S_1 learns the *uniqueness pattern* $\mathsf{UP}^d(q_i)$ at depth d, where $\mathsf{UP}^d(q_i)$ denotes the number of the unique objects that appear at current depth d. The distinct encrypted values at depth d are independent from all other depths, therefore, this protocol still protects the distribution of the original R. In addition, due to the 're-encryptions' during the execution of the protocol, all the encryptions are fresh ones, i.e., there are not as the same as the encryptions from ER. Finally, we emphasize that nothing on the objects and their values have been revealed since they are all encrypted.

Batch Processing In the query processing $\mathsf{SecQuery}$, we observe that we do not need to run the protocols $\mathsf{SecDupElim}$ and $\mathsf{EncSort}$ for every depth. Since $\mathsf{SecDupElim}$ and $\mathsf{EncSort}$ are the most costly protocols in $\mathsf{SecQuery}$, we can perform *batch processing* and execute them after a few depths. Our observation is that there is no need to deduplicate repeated objects at each scanned depth. If we perform the $\mathsf{SecDupElim}$ after certain depths of scanning, then the repeated objects will be eliminated, and those distinct encrypted objects with updated worst and best scores will be sorted by running $\mathsf{EncSort}$. The protocol remains correct. We introduce a parameter p such that $p \geq k$. The parameter p specifies where we need to run the $\mathsf{SecDupElim}$ and $\mathsf{EncSort}$ in the $\mathsf{SecQuery}$. That is, the server S_1 runs the $\mathsf{SecQuery}$ the same as in Algorithm 2, except that every p depths we run line 8-11 in Algorithm 2 to check if the algorithm could halt. Furthermore, we can replace the $\mathsf{SecDupElim}$ with the $\mathsf{SecDedup}$ in the batch processing for better privacy but at the cost of some efficiency.

Security. Compared to the optimization from SecDupElim, we show that the batching strategy provides more privacy than just running the SecDupElim alone. For query q, assuming that we compute the scores over m attributes. Recall that the $\mathsf{UP}^p(q)$ at depth p has been revealed to S_1 while running SecDupElim, therefore, after the first depth, in the worst case, S_1 learns that the objects at the first depth is the same object. To prevent this worst case leakage, we perform SecDupElim every p depth. Then S_1 learns there are p distinct objects in the worst case. After depth p, the probability that S_1 can correctly locate those distinct encrypted objects' positions in the table is at most $\frac{1}{(p!)^m}$. This decreases fast for bigger p. However, in practice this leakage is very small as many distinct objects appear every p depth. Similar to all our protocols, the encryptions are fresh due to the 're-encryption' by the server. Even though S_1 has some probability of guessing the distinct objects' location, the object id and their scores have not been revealed since they are all probabilistically encrypted.

X. EXPERIMENTS

To evaluate the performance of our protocol, we conduct a set of experiments using real and synthetic datasets. We used the `HMAC-SHA-256` as the Pseudo-Random Function (PRF) for EHL encoding and the security parameter for the Paillier & DJ encryptions is set to 1024. All experiments are implemented using C++. We implemented the scheme $\mathsf{SecTopK} = (\mathsf{Enc}, \mathsf{Token}, \mathsf{SecQuery})$, including SecWorst, SecBest, EncSort, and EncCompare and their optimizations. We run our experiments on a 24-core machine who serves as the cloud, running Scientific Linux with 128GB memory and 2.9GHz Intel Xeon.

DataSets. We use the following real world dataset downloaded from UCI ML Repository. `insurance`: a benchmark dataset that contains 5822 customers' information on an insurance company and we extracted 13 attributes from the original dataset, `diabetes`: a patients' dataset containing 101767 patients' records (i.e. data objects), where we extracted 10 attributes, and `PAMAP`: a physical activity monitoring dataset that contains 376416 objects, and we extracted 15 attributes. We also generated a synthetic dataset `synthetic` with 1 million objects and 10 integer attributes that takes values from a Gaussian distribution.

A. Evaluation of the Encryption

We implemented the EHL and instantiated the PRF by using the `HMAC` keyed secure hash function. We set the number of secure hash function `HMAC` to be $s = 5$, and, as discussed in the previous section, we obtained negligible false positive rate in practice. The encryption is independent of the characteristics of the dataset and depends only on the size. Therefore, when encrypting each dataset, we used 64 threads on the machine that we discussed before. Table III shows that, both in terms of time and space, the cost of

the encryption is reasonable and scales linearly to the size of the database. Finally, we emphasize that the encryption only incurs a one-time off-line construction overhead.

Dataset	EHL	
	time (sec.)	size (MB)
`insurance`	0.31	0.65
`diabetes`	5.50	11.32
`PAMAP`	20.41	41.82
`synthetic`	54.22	111.37

Table III: EHL size and time

B. Query Processing Performance

1) Query Evaluation and Methodology: We evaluate the performance of the secure query processing and their optimizations that we discussed before. In particular, we use the query algorithm without any optimization but with full privacy, denoted as Qry_F; the query algorithm running SecDupElim instead of SecDedup at each depth, denoted as Qry_E; the one using the batching strategies, denoted as Qry_Ba. We evaluate the query processing performance using all the datasets and use EHL^+ to encrypt all of the object ids. Notice that the performance of the NRA algorithm depends on the distribution of the dataset among other things. To present a clear comparison of the different methods, we measure the average time per depth for the query processing, i.e. $\frac{T}{D}$ where T is the total time that the program spends on executing a query and D is the total number of depths the program scanned before halting. In most of our experiments the value of D ranges between a few hundreds and a few thousands. For each query, we randomly choose the number of attributes m that are used for the ranking function ranging from 2 to 8, and we also vary k between 2 and 20. The ranking function F that we use is the sum function.

Qry_F, Qry_E, and Qry_Ba. We report the query processing performance without any query optimization. Figure 3 shows Qry_F query performance. The results are very promising considering that the query is executed completely on encrypted data. For a fixed number of attributes $m = 3$, the average time is about 1.30 seconds for the largest dataset `synthetic` running top-20 queries. When fixing $k = 5$, the average time per depth for all the dataset is below 1.2s. As we can see that, for fixed m, the performance scales linearly as k increases. Similarly, the query time also linearly increases as m gets larger for fixed k. The experiments show that the SecDupElim improves the efficiency of the query processing. Figure 4 shows the querying overhead for exactly the same setting as before. Since Qry_E eliminates all the duplicated the items for each depth, Qry_E has been improved compared to the Qry_F above. As k increases, the performance for Qry_E is up to 5 times faster than Qry_F when k increase to 20. On the other hand, fixing $k = 5$, the
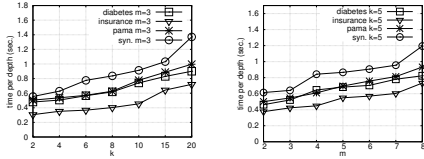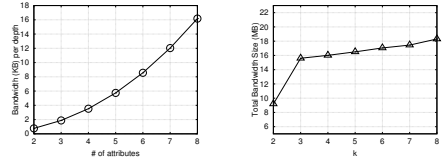
(a) varying k (b) varying m
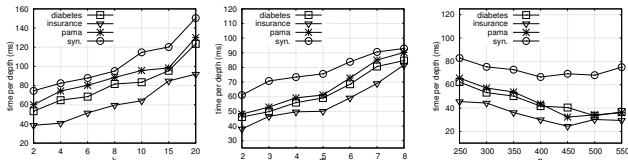
Figure 3: Qry_F performance
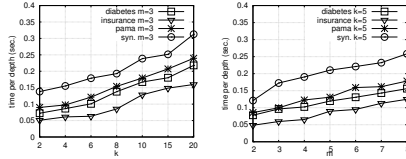
(a) varying k (b) varying m

Figure 4: Qry_E performance

(a) band. per depth   (b) varying k

Figure 5: Comm. bandwidth

(a) varying k (b) varying m (c) varying p

Figure 6: Qry_Ba performance

Dataset	Qry_Ba	Qry_E	Qry_F
insurance	28.2	53.8	247.6
diabetes	31.4	73.2	465.5
PAMAP	33.1	75.1	516.8
synthetic	65.4	115.3	959.5

Table IV: Time per depth (milliseconds)

Dataset	bandwidth (MB)	latency (sec.)
insurance	8.87	1.41
diabetes	12.45	1.99
PAMAP	15.72	2.5152
synthetic	17.3	2.768

Table V: Comm. bandwidth & latency ($k = 20$, $m = 4$)

performance of Qry_E is up to around 7 times faster than Qry_F as m grows to 20. In general, the experiments show that Qry_E effectively speed up the query time 5 to 7 times over the basic approach.

We evaluate the effectiveness of batching optimization for the Qry_Ba queries. Figure 6 shows the query performance of the Qry_Ba for the same settings as the previous experiments. The experiments show that the batching technique further improves the performance. In particular, for fixed batching parameter $p = 150$, i.e. every 150 depths we perform SecDupElim and EncSort in the SecQuery, and we vary our k from 2 to 20. Compared to the Qry_E, the average time per depth for all of the datasets have been further improved. For example, when $k = 2$, the average time for the largest dataset synthetic is reduced to 74.5 milliseconds. while for Qry_F it takes more than 500 milliseconds. For diabetes, the average time is reduced to 53 milliseconds when $k = 2$ and 123.5 milliseconds when k increases to 20. As shown in figure 6a, the average time linearly increases as k gets larger. In Figure 6c, we further evaluate parameter p. Ranging p from 200 to 550, the experiments show that the proper p can be chosen for better query performance. In general, for different datasets, there are different p's that can achieve the best query performance. When p gets larger, the number of calls for EncSort and SecDupElim are reduced, however, the performance for these two protocols also slow down as there're more encrypted items. We finally compare the three queries' performance. Table IV shows the query performance when fixing $k = 5$, $m = 3$, and $p = 500$. Clearly, as we can see, Qry_Ba significantly improves the performance.

2) Communication Bandwidth and Cost: We evaluate the communication cost of our protocol. In particular, we evaluate the communication of the fully secure and unop-

timized Qry_F queries on the largest dataset synthetic. The speed of the network between two clouds depends on the location and the technology of the clouds. Furthermore, with recent networking advances [1], we expect that the connections between clouds (inter-clouds) will be much higher [8]. However, even if we assume that the communication between the two clouds is about 50 Mbps, the total cost of the communication at each depth is below 1 ms! Thus, communication is not a bottleneck for our protocol. In Figure 5a, we report the actual bandwidth per depth. In Figure 5b, we show the total bandwidth when executing the top-20 by fixing $m = 4$. Also, assuming a standard 50 Mbps LAN setting, we show in Table V the total network latency between servers S_1 and S_2 when $k = 20$ and $m = 4$. Based on the discussion above, we can see that the communication cost of our protocol is very moderate for any reasonable assumptions about the connectivity between the two clouds.

C. Related works on secure kNN

As discussed in the section II, although existing work [18] on secure kNN does not directly solve our problem, we can adopt their techniques to obtain top-k results by restricting our scoring function to be $\sum x_i^2(o)$. We then use as a query a point with large enough values in each attribute and run their secure k-nearest-neighbor scheme. In order to compare the experiment from [18], during our encryption setup in our SecTopK, the data owner needs to encrypt the additional squares of the values, i.e. $\mathsf{Enc}(x_i^2(o))$, then the scoring function would simply be the sum of all the attributes. We emphasize that the execution of the rest of our protocol remains the same.

We can now use the results from the experiments in [18]. It is clear that the protocol in [18] is very inefficient. For example, it is reported that it would take more than 2 hours to return 10 nearest neighbors for a database of only 2,000 records. On the other hand, with our scheme, we can return 10 nearest neighbors over a database with the same characteristics of 1 million records in less than 30 minutes. Moreover, as [18] needs to sends all of the encrypted records for each query execution, the communication bandwidth is very large even for small dataset that has 2,000 records. On the other hand, in our approach, we show that the bandwidth cost is low and will not affect the performance much.

XI. Conclusion

We propose the first secure scheme that executes top-k ranking queries over encrypted databases. First, we describe a secure probabilistic data structure called encrypted hash list (EHL). We then propose a number of building blocks that can securely compute top-k objects based on their ranks. We also provide a clean and formal security analysis of our proposed scheme where we explicitly state the leakage of various schemes. We experimentally evaluated using real-world datasets to show the scheme is efficient and practical.

References

[1] http://www.cisco.com/c/en/us/products/cloud-systems-management/intercloud-fabric/index.html.

[2] B. Adida and D. Wikström. How to shuffle in public. In *TCC*, pages 555–574, 2007.

[3] C. C. Aggarwal and P. S. Yu, editors. *Privacy-Preserving Data Mining-Models and Algorithms*, volume 34 of *Advances in Database Systems*. 2008.

[4] D. Agrawal, A. El Abbadi, F. Emekci, A. Metwally, and S. Wang. Secure data management service on cloud computing infrastructures. In *New Frontiers in Information and Software as Services*, volume 74, pages 57–80. 2011.

[5] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy. Transaction processing on confidential data using cipherbase. In *ICDE*, pages 435–446, 2015.

[6] S. Bajaj and R. Sion. Trusteddb:a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, pages 205–216, 2011.

[7] F. Baldimtsi and O. Ohrimenko. Sorting and searching behind the curtain. In *FC*, 2014.

[8] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[9] R. Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *ACM SIGSAC CCS*, pages 1143–1154, 2016.

[10] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

[11] S. Bugiel, S. Nürnberger, A. Sadeghi, and T. Schneider. Twin clouds: Secure cloud computing with low latency. In *CMS*, pages 32–44, 2011.

[12] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, 2013.

[13] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT*, pages 577–594, 2010.

[14] S. Choi, G. Ghinita, H. Lim, and E. Bertino. Secure knn query processing in untrusted cloud environments. *TKDE*, 26:2818–2831, 2014.

[15] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *CCS 2006*, pages 79–88. ACM, 2006.

[16] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *PKC*, pages 119–136, 2001.

[17] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

[18] Y. Elmehdwi, B. K. Samanthula, and W. Jiang. Secure k-nearest neighbor query over encrypted data in outsourced environments. In *ICDE*, pages 664–675, 2014.

[19] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[20] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[21] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43:431–473, 1996.

[22] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.

[23] I. Hang, F. Kerschbaum, and E. Damiani. ENKI: access control for encrypted query processing. In *SIGMOD*, pages 183–196, 2015.

[24] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multi-dimensional range queries over outsourced data. *VLDB J.*, 21(3):333–358, 2012.

[25] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[26] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[27] C. C. Jaideep Vaidya, Murat Kantarcioglu. Privacy-preserving naïve bayes classification. *VLDB J.*, 17(4):879–898, 2008.

[28] M. Kuzu, M. S. Islam, and M. Kantarcioglu. Efficient privacy-aware search over encrypted databases. CODASPY, pages 249–256, 2014.

[29] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar. Fast range query processing with strong privacy protection for cloud computing. *PVLDB*, 7(14):1953–1964, 2014.

[30] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *CRYPTO*, pages 36–54, 2000.

[31] A. Liu, K. Zheng, L. Li, G. Liu, L. Zhao, and X. Zhou. Efficient secure similarity computation on encrypted trajectory data. In *ICDE*, pages 66–77, 2015.

[32] X. Meng, S. Kamara, K. Nissim, and G. Kollios. GRECS: graph encryption for approximate shortest distance queries. In *CCS*, pages 504–517, 2015.

[33] X. Meng, H. Zhu, and G. Kollios. Top-k query processing on encrypted databases with strong security guarantees. *CoRR*, arXiv:1510.05175, 2018.

[34] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[35] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[36] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious ram. In *USENIX Security*, 2015.

[37] B. K. Samanthula, W. Jiang, and E. Bertino. Privacy-preserving complex query evaluation over semantically secure encrypted data. In *ESORICS*, pages 400–418, 2014.

[38] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *Oakland S & P*, pages 44–55, 2000.

[39] J. Vaidya and C. Clifton. Privacy-preserving top-k queries. In *ICDE*, pages 545–546, 2005.

[40] J. Vaidya, C. Clifton, M. Kantarcioglu, and A. S. Patterson. Privacy-preserving decision trees over vertically partitioned data. *TKDD*, 2(3), 2008.

[41] W. K. Wong, D. W.-l. Cheung, B. Kao, and N. Mamoulis. Secure knn computation on encrypted databases. In *SIGMOD*, pages 139–152, 2009.

[42] B. Yao, F. Li, and X. Xiao. Secure nearest neighbor revisited. In *ICDE*, pages 733–744, 2013.