

# EPiC: efficient privacy-preserving counting for MapReduce

Triet Dang Vo-Huu<sup>1</sup> · Erik-Oliver Blass<sup>2</sup> · Guevara Noubir<sup>1</sup>

Received: 6 May 2016 / Accepted: 6 June 2018 / Published online: 18 June 2018  
© Springer-Verlag GmbH Austria, part of Springer Nature 2018

**Abstract** In the face of an untrusted cloud infrastructure, outsourced data needs to be protected. We present EPiC, a practical protocol for the privacy-preserving evaluation of a fundamental operation on data sets: frequency counting. In an encrypted outsourced data set, a cloud user can specify a pattern, and the cloud will count the number of occurrences of this pattern in an oblivious manner. A pattern is expressed as a Boolean formula on the fields of data records and can specify values counting, value comparison, range counting, and conjunctions/disjunctions of field values. We show how a general pattern, defined by a Boolean formula, is arithmetized into a multivariate polynomial and used in EPiC. To increase the performance of the system, we introduce a new privacy-preserving encoding with “somewhat homomorphic” properties. The encoding is highly efficient in our particular counting scenario. Besides a formal analysis where we prove EPiC’s privacy, we also present implementation and evaluation results. We specifically target Google’s prominent MapReduce paradigm as offered by major cloud providers. Our evaluation performed both locally and in Amazon’s public cloud with up to 1 TByte data sets shows only a modest overhead of 20% compared to non-private counting, attesting to EPiC’s efficiency.

**Keywords** Privacy-preserving · MapReduce · Somewhat homomorphic encryption

---

✉ Triet Dang Vo-Huu  
vohudtr@ccs.neu.edu

Erik-Oliver Blass  
erik-oliver.blass@airbus.com

Guevara Noubir  
noubir@ccs.neu.edu

<sup>1</sup> Northeastern University, Boston, MA 02115, USA

<sup>2</sup> Airbus Group Innovations, 81663 Munich, Germany

## Mathematics Subject Classification 97R99

### 1 Introduction

Cloud computing is a promising technology for large enterprises and even governmental organizations. Major cloud computing providers such as Amazon and Google offer users to outsource their data and computation. The main advantage for users lies in the clouds' flexible cost model: users are only charged by use, e.g., total amount of storage or CPU time used. In addition, clouds "elastic" services allow the users to efficiently scale resources to satisfy dynamic load. The appeal and success of outsourcing data and operating on outsourced data is exemplified by Google's prominent MapReduce API [6]. MapReduce is typically used for analysis operations on huge amounts of (outsourced) data, e.g., scanning through data and finding patterns, counting occurrences of specific patterns, and other statistics [11].

While the idea of moving data and computation to a (public) cloud for cost savings is appealing, trusting the cloud to store and protect data against *adversaries* is a serious concern for users. Examples for adversaries can be hackers that break into the cloud (data center) to steal data, insiders such as data center administrative staff who can easily access data, and other cloud users hosted on the same data center ("multi tenancy"). Finally, as cloud providers place data centers abroad in foreign countries with unclear privacy laws, local authorities are threatening outsourced data. Such attacks are realistic and have already been reported in the real-world [10, 17, 20, 26].

The encryption of data is a viable privacy protection mechanism, but it renders subsequent operations on encrypted data a challenging problem. To address this problem, *Fully Homomorphic Encryption* (FHE) techniques have been investigated, cf. Gentry [8] or Vaikuntanathan [23] for an overview. FHE guarantees that computations can be performed on encrypted data as if ciphertexts were plaintexts, while the cloud neither learns details about the stored data nor about the results. However, today's FHE schemes are still overly inefficient [5, 9, 24], and a deployment in a real-world cloud would outweigh any cost advantage offered by the cloud. Moreover, any solution for a real-world cloud needs to be tailored to the specifics of the cloud computing paradigm, e.g., MapReduce [6].

This paper presents EPiC—Efficient PrIvacy-preserving Counting for MapReduce, an efficient, practical, yet privacy-preserving protocol for a fundamental data analysis primitive in MapReduce: *counting occurrences* of patterns. In an outsourced data set comprising a large number of encrypted data records, EPiC allows the cloud user to specify a pattern, and the cloud will count the number of occurrences of this pattern (and therefore histograms) in the stored ciphertexts without revealing the pattern and how often it occurs. A pattern is expressed as a Boolean formula on countable fields of data records and can specify a specific field value, a value comparison, a range of field values, and more complex forms of conjunctions/disjunctions among sub-patterns. For example, in an outsourced data set of patient health records, a pattern could be  $age \in [50, 70]$  and  $(diabetes = 1 \text{ or } hypertension = 1)$ . The main idea of EPiC is to transform the problem of privacy-preserving pattern counting into a summation of polynomial evaluations. Our work is inspired by Lauter et al. [15] to use *some-*

what *homomorphic* encryption to address specific privacy-preserving operations. In EPiC, we extend a previous work on cPIR protocols [22] to design a new “encoding” mechanism that exhibits homomorphic properties at the expense of growing ciphertext size (thus, the scheme is called *somewhat homomorphic*). While we call our encoding encryption in the rest of this paper, we stress that our encryption does not provide traditional IND-CPA security (INDistinguishability under Chosen Plaintext Attacks [14]), but only weaker properties suited to the context we target in this paper, i.e., the summation of polynomial evaluations. In return, our “encryption” is particularly efficient in this context. We also show how a general pattern, defined by a Boolean formula, is arithmetized into a multivariate polynomial over binary finite (Galois) field [14]— $GF(2)$ —optimizing for efficiency. EPiC evaluates this polynomial efficiently by using a new, efficient *somewhat homomorphic* encryption mechanism that is based on the Hidden Modular Group Order assumption [22]—fully homomorphic encryption is not required. In conclusion, the contributions of this paper are:

- EPiC, a new protocol to enable privacy-preserving pattern counting in MapReduce clouds. EPiC reduces the problem of counting occurrences of a Boolean pattern to the summation of a multivariate polynomial evaluated on encrypted data.
- Designing a new encoding/encryption scheme specifically addressing secure counting in a highly efficient manner.
- An implementation of EPiC together with an extensive evaluation in a realistic setting. The source code is available for download [25].

## 2 Problem statement

### 2.1 Cloud counting

We will use an example application to motivate our work. Along the lines of recent reports [21], imagine a hospital scenario where patient records are managed electronically. To reduce cost and grant access to, e.g., other hospitals and external doctors, the hospital refrains from investing into an own, local data center, but plans to outsource patient records to a public cloud. Regulatory matters require the privacy-protection of sensitive medical information, so outsourced data has to be encrypted. However, besides uploading, retrieving or editing patient records performed by multiple entities (hospitals, doctors etc.), one entity eventually wants to collect some statistics on the outsourced patient records without the necessity of downloading all of them.

More specifically, we assume that each patient record  $R$ , besides raw data such as a picture or some doctors’ notes, also includes one or more countable fields  $R.c$  containing some patterns. In practice, this field could denote the category or type of disease a patient is suffering from, e.g., “diabetes” or “hypertension”. A user (e.g., doctor)  $\mathcal{U}$  wants to extract the frequency of occurrence of pattern  $\chi$ , e.g., how many patients have  $R.disease = \chi$ . Due to the large amount of data, downloading each patient record is prohibitive, and the counting should be performed by the cloud. While encryption of data, access control, and key management in a multi-user cloud environment are clearly important topics, we focus on the problem of a-posteriori extracting information out of the outsourced data in a privacy-preserving manner.

The cloud must neither learn details about the stored data, nor any information about the counting, what is counted, the count itself, etc. Instead, the cloud processes  $\mathcal{U}$ 's counting queries “obliviously”. We will now first specify the general setup of counting schemes for public clouds and then formally define privacy requirements. Note that throughout this paper, we will assume the countable fields to be non-negative integer fields. Besides, records may contain non-countable data, e.g., pictures or doctors' notes, that can be IND-CPA (AES-CBC) encrypted—Therewith, it is of no importance for privacy defined below.

**Definition 1** (*Cloud counting*) Let  $\mathcal{R}$  denote a sequence of records  $\mathcal{R} := \{R_1, \dots, R_n\}$ . Besides some non-countable data, each record  $R_i$  contains  $m$  different countable fields. The  $k$ -th countable field of the  $i$ -th record, denoted as  $R_{i,k}$ ,  $1 \leq k \leq m$ , can take values  $R_{i,k} \in \mathcal{D}_k = \{0, 1, \dots, |\mathcal{D}_k| - 1\}$ , where  $\mathcal{D}_k$  denotes the domain of the  $k$ -th field with size<sup>1</sup>  $|\mathcal{D}_k|$ . For the “multi-domain” of  $m$  countable fields we write  $\mathcal{D} = \mathcal{D}_1 \times \dots \times \mathcal{D}_m$ . A privacy-preserving counting scheme comprises the following probabilistic polynomial time algorithms:

1.  $\text{KEYGEN}(\kappa)$ : using a security parameter  $\kappa$ , outputs a secret key  $\mathcal{S}$ .
2.  $\text{ENCRYPT}(\mathcal{S}, \mathcal{R})$ : uses secret key  $\mathcal{S}$  to encrypt the sequence of records  $\mathcal{R}$  to  $\mathcal{E} := \{E_{R_1}, \dots, E_{R_n}\}$ , where  $E_{R_i}$  denotes the encryption of record  $R_i$ .
3.  $\text{UPLOAD}(\mathcal{E})$ : uploads the sequence of encryptions  $\mathcal{E}$  to the cloud.
4.  $\text{PREPAREQUERY}(\mathcal{S}, \chi)$ : generates an encrypted query  $Q$  out of secret  $\mathcal{S}$  and the multiple-field pattern  $\chi \in \mathcal{D}$ .
5.  $\text{PROCESSQUERY}(Q, \mathcal{E})$ : uses an encrypted query  $Q$ , the sequence of ciphertexts  $\mathcal{E}$ , and outputs a result  $E_\Sigma$ . This algorithm performs the actual counting.
6.  $\text{DECODE}(\mathcal{S}, E_\Sigma)$ : takes secret  $\mathcal{S}$  and  $E_\Sigma$  to output a final result, the occurrences  $\Sigma$  (the “count”) of the specified pattern in  $\mathcal{R}$ .

According to this definition, cloud user  $\mathcal{U}$  encrypts the sequence of records and uploads them into the cloud. If  $\mathcal{U}$  wants to know the number of occurrences of  $\chi$  in the records, he prepares a query  $Q$ , which is—as we will see later—simply a fixed-length sequence of encrypted values.  $\mathcal{U}$  then sends  $Q$  to the cloud, and the cloud processes  $Q$ . Finally, the cloud sends a result  $E_\Sigma$  back to  $\mathcal{U}$  who can decrypt this result and learn the number  $\Sigma$  of occurrences of pattern  $\chi$ , i.e., the count. The idea of a performing the counting in the cloud is to put the main computational burden on the cloud side. Both storage and computational overhead for  $\text{KEYGEN}$ ,  $\text{ENCRYPT}$ ,  $\text{UPLOAD}$ ,  $\text{PREPAREQUERY}$ , and  $\text{DECODE}$  should be lightweight compared to  $\text{PROCESSQUERY}$ .

## 2.2 Privacy

In the face of an untrusted cloud infrastructure, cloud user  $\mathcal{U}$  wants to perform counting in a privacy-preserving manner. Intuitively, the data stored at the cloud as well as the counting operations must be protected against a curious cloud. Informally, we demand (1) *storage privacy*, where the cloud does not learn anything about stored data, and

<sup>1</sup> Domain size  $|\mathcal{D}_k|$  indicates the number of different values a field can take.

(2) *counting privacy*, where the cloud does not learn anything about queries and query results. The cloud, which we now call “adversary”  $\mathcal{A}$ , should only learn “trivial” privacy properties like the total size of outsourced data, the total number of patient records or the number of counting operations performed for  $\mathcal{U}$ . We formalize privacy for counting using a game-based setup. In the following,  $\epsilon(\kappa)$  denotes a negligible function in the security parameter  $\kappa$ .

**Definition 2** (*Bit mapping*) Let  $\mathcal{R} = \{R_1, \dots, R_n\}$  be a set of records, and  $R_{i,k} \in \{0, 1\}^*$  the  $k$ -th field of record  $R_i$ . Let  $\chi, \Sigma \in \{0, 1\}^*$  be bit string representations of a pattern and a count. For  $X \in \{R_{i,k}, \chi, \Sigma\}$ ,  $\text{bit}(j, X)$  denotes the  $j$ -th bit of  $X$ .

**Definition 3** (*Storage privacy*) A challenger generates two same-size same-field-types sets of records  $\mathcal{R}, \mathcal{R}'$  and two patterns  $\chi, \chi' \in \mathcal{D}$ . The challenger then uses ENCRYPT and PREPAREQUERY to compute the encrypted sets of records  $\mathcal{E}, \mathcal{E}'$  and two encrypted counting queries  $Q, Q'$  corresponding to two patterns  $\chi, \chi'$ . Using PROCESSQUERY, he evaluates  $\mathcal{E}$  with  $Q$ , and  $\mathcal{E}'$  with  $Q'$  to get encrypted results  $E_\Sigma, E'_{\Sigma'}$ . The challenger sends  $I := \{\mathcal{E}, \mathcal{E}', Q, Q', E_\Sigma, E'_{\Sigma'}\}$  to adversary  $\mathcal{A}$ . For any patterns  $\chi, \chi'$ , any  $X, X'$  such that either  $X \in \{\{R_{i,k}\}\}$  and  $X' \in \{\{R_{i,k}\}\}$  or  $X = \chi$  and  $X' = \chi'$  or  $X = \Sigma$  and  $X' = \Sigma'$ , and for any  $b = \text{bit}(j, X)$  and  $b' = \text{bit}(j', X')$ , the adversary  $\mathcal{A}$  outputs 1, if she guesses  $b = b'$ , and 0 otherwise. A protocol preserves storage privacy, *iff* for any probabilistic polynomial time (PPT) algorithm  $\mathcal{A}$ , the gain of using  $\mathcal{A}$  to produce correct output over a random guess is negligible. That is,  $|\Pr[\mathcal{A}(I) = 1|b = b'] - \frac{1}{2}| \leq \epsilon(\kappa)$  and  $|\Pr[\mathcal{A}(I) = 0|b \neq b'] - \frac{1}{2}| \leq \epsilon(\kappa)$ .

**Definition 4** (*Counting privacy*) A challenger generates two same-size same-field-types sets of records  $\mathcal{R}, \mathcal{R}'$ , and two patterns  $\chi, \chi'$ , uses ENCRYPT, PREPAREQUERY, and PROCESSQUERY, and sends encrypted  $I := \{\mathcal{E}, \mathcal{E}', Q, Q', E_\Sigma, E'_{\Sigma'}\}$ , to  $\mathcal{A}$ . Now,  $\mathcal{A}$  outputs 1, if  $\chi = \chi'$ , and 0 otherwise. A protocol preserves counting privacy, *iff* for any PPT algorithm  $\mathcal{A}$  the probability of correct output is not better than a random guess:  $|\Pr[\mathcal{A}(I) = 1|\chi = \chi'] - \frac{1}{2}| \leq \epsilon(\kappa)$  and  $|\Pr[\mathcal{A}(I) = 0|\chi \neq \chi'] - \frac{1}{2}| \leq \epsilon(\kappa)$ .

Similar to traditional indistinguishability, *storage privacy* and *counting privacy* captures the intuition that, by storing data and counting, the cloud should not learn anything about the content it stores. In addition, the cloud should not learn anything about the counting performed, such as which pattern is counted, whether a pattern is counted twice or what the resulting count is.

### 2.3 MapReduce

The efficiency of counting relies on the performance of PROCESSQUERY which involves processing huge amounts of data in the cloud. Cloud computing usually processes data in parallel via multiple nodes in the cloud data center based on some computation paradigm. For efficiency, PROCESSQUERY has to take the specifics of that computation into account. One of the most widespread, frequently used framework for distributed computation that is offered by major cloud providers today is MapReduce [6].

In the following, we will give a very compressed overview about MapReduce, only to understand EPiC. For more details, refer to Dean and Ghemawat [6]. In the

MapReduce framework, a user uploads his data into the cloud. During upload, data is automatically split into pieces (*InputSplits*) and distributed among the nodes in the cloud's data center. If the user wants the cloud to perform an operation on the outsourced data, he uploads an implementation of his operation, e.g., Java .class files, to the cloud. More precisely, the user has to provide implementations of two functions, the so called *map* function and the *reduce* function—these two functions will be executed by the cloud on the user's data.

EPiC's counting "job" runs in two phases. First, in the "mapping" phase, *Mapper* nodes scan data through *InputSplits* (data pieces split automatically by MapReduce framework) and evaluate the counting's *map* function on the data. These operations are performed by all Mappers in parallel. The outputs of each *map* function are sent to one *Reducer* node, which, in the "reducing" phase, aggregates them and produces a final output that is sent back to the user. This setup takes advantage of the parallel nature of a cloud data center and allows for scalability and elasticity.

### 3 EPiC protocol

To motivate the need for a more sophisticated protocol like EPiC, we briefly discuss why possible straightforward solutions do not work in our application scenario.

*Precomputed counters* One could imagine that the cloud user, in the purpose of counting a value  $\chi_k$  in a single countable field  $\mathcal{D}_k$ , simply stores encrypted counters for each possible value of  $\chi_k$  in domain  $\mathcal{D}_k$  in the cloud. Each time records are added, removed or updated, the cloud user updates the encrypted counters. However, this approach does not scale very well in our scenario where multiple cloud users (different "doctors") perform updates and add or modify records. An expensive user side locking mechanism would be required to ensure consistency of the encrypted counters. Moreover, in the case of complex queries involving multiple fields, all possible combinations of counters need to be updated by users involving a lot of user side computation.

*Per-record counters ("Voting")* Alternatively and similarly to a naive voting scheme, each encrypted record stored in the cloud could be augmented with an encrypted "voting" field containing  $|\mathcal{D}_k|$  subsets, each of  $\log_2 n$  bits. If a record's countable value in field  $\mathcal{D}_k$  matches the value corresponding to a subset, then the according subset is set to 1. To find the count, the cloud sums the encrypted voting fields (using additive homomorphic encryption) for all records. Again, such an approach requires heavy locking mechanism and recomputation of counters for each operation of adding, removing, or modifying a record. In conclusion, these straightforward solutions require heavy user-side computation and do not provide efficient, practical, and flexible solutions for multi-user, multiple field data sets.

#### 3.1 EPiC overview

For ease of understanding, we *initially* introduce EPiC for the simpler case of counting on only a single countable field  $\mathcal{D}_k$  in a multiple countable fields data set where values are in  $\text{GF}(q)$ . Subsequently, we extend EPiC to support counting on Boolean combinations of multiple countable fields  $\mathcal{D}_1, \dots, \mathcal{D}_m$  over  $\text{GF}(q)$ . Finally, for perfor-

mance improvement, we further optimize our mechanisms by considering conversion of (generic) finite fields  $GF(q)$  into binary finite fields  $GF(2)$ .

EPiC’s main rationale is to perform the counting in the cloud by evaluating an *indicator polynomial*  $P_\chi(\cdot)$ , as query  $Q$ , specific to the pattern  $\chi$  the cloud user  $\mathcal{U}$  is interested in. Conceptually, the cloud evaluates  $P_\chi(\cdot)$  on the countable fields’ values of each record. The outcome of all individual polynomial evaluations is a (large) set of values of either “1” or “0”. The cloud now adds these values and sends the sum back to  $\mathcal{U}$ , who learns the number of occurrences of  $\chi$  in the investigated set of records.

### 3.2 Counting on a single field

Without loss of generality, we assume a user  $\mathcal{U}$  wishes to count occurrences of  $\chi$  in the first field  $\mathcal{D}_1$  in an oblivious manner. The idea is to prepare a univariate indicator polynomial  $P_\chi(x)$  such that

$$P_\chi(x) = \begin{cases} 1, & \text{if } x = \chi \\ 0, & \text{otherwise} \end{cases}, \tag{1}$$

and scan through the data set  $\mathcal{R} = \{R_1, \dots, R_n\}$  of all records to compute the sum  $\sum_{i=1}^n P_\chi(R_{i,1})$ . The result is the number of occurrences of  $\chi$  in the first field in the data set. The idea for generating  $P_\chi(x)$  is to construct the polynomial in the Lagrange interpolation form

$$P_\chi(x) := \sum_{j=0}^{|\mathcal{D}_1|-1} a_j \cdot x^j := \prod_{\alpha \in \mathcal{D}_1, \alpha \neq \chi} \frac{x - \alpha}{\chi - \alpha}. \tag{2}$$

The polynomial  $P_\chi(x)$  is of degree  $|\mathcal{D}_1| - 1$ , and its coefficients  $a_j$  are uniquely determined.

**Encrypted polynomial** In EPiC, each countable value  $R_{i,k}$  is encrypted to  $E_{R_{i,k}}$ . The above indicator polynomial based counting method for plaintext values can be applied to ciphertexts in a similar manner. User  $\mathcal{U}$  prepares the indicator polynomial based on plaintext  $\chi$ , but  $\mathcal{U}$  encrypts coefficients  $a_j$  to  $E_{a_j}$  before sending them to the cloud, which now computes the encrypted sum

$$E_\Sigma := \sum_{i=1}^n P_\chi(E_{R_{i,1}}) = \sum_{i=1}^n \sum_{j=0}^{|\mathcal{D}_1|-1} E_{a_j} \cdot (E_{R_{i,1}})^j. \tag{3}$$

Note that the polynomial *coefficients* are encrypted (and potentially large), but the polynomial *degree* remains  $|\mathcal{D}_1| - 1$ . In order for the cloud to compute  $E_\Sigma$  and user  $\mathcal{U}$  to decrypt it later, additively and multiplicatively homomorphic properties are required for the encryption, which we describe in Sect. 3.5. As a final step,  $\mathcal{U}$  simply receives back  $E_\Sigma$  and only decrypts the count  $\sigma := \text{DEC}(E_\Sigma) = P_\chi(x)$ . This does not require high computational costs at the user, suiting the cloud computing paradigm well.

**Cloud computation cost** The above technique requires  $n \cdot |\mathcal{D}_1|$  additions,  $n \cdot |\mathcal{D}_1|$  multiplications, and  $n \cdot (|\mathcal{D}_1| - 1)$  exponentiations. We can improve efficiency by rearranging the order of computations in Eq. (3) as follows:

$$E_\Sigma := \sum_{i=1}^n P_\chi(E_{R_{i,1}}) = \sum_{i=1}^n \sum_{j=0}^{|\mathcal{D}_1|-1} E_{a_j} \cdot (E_{R_{i,1}})^j = \sum_{j=0}^{|\mathcal{D}_1|-1} \left( E_{a_j} \cdot \sum_{i=1}^n (E_{R_{i,1}})^j \right). \tag{4}$$

Therewith, the number of multiplications is reduced to  $|\mathcal{D}_1|$ . We also note that in the case of a binary domain ( $|\mathcal{D}_1| = 2$ ), Eq. (4) becomes  $E_\Sigma = E_{a_0} \cdot n + E_{a_1} \cdot \sum_{i=1}^n E_{R_{i,1}}$ , i.e., there are no exponentiations. This observation motivates our optimization described later in Sect. 3.4.

**Oblivious counting** *First*, the query is submitted to the cloud as a sequence of encrypted coefficients of the indicator polynomial; *second*, no matter what query is made, exactly  $|\mathcal{D}_1|$  coefficients (including 0-coefficients) are sent, thus preventing the cloud to infer query information based on the query size. We will discuss the user privacy in details in Sect. 3.7.

### 3.3 Counting patterns defined by a Boolean formula

We now extend the indicator polynomial based counting technique towards a general solution for counting patterns defined by any Boolean combination of *multiple* fields in the data set. That is, each  $R_i$  record can contain multiple countable fields, such as  $R_{i,1}, R_{i,2}, \dots, R_{i,m}$ . First, we present EPiC’s construction of the indicator polynomial for *conjunctive* combinations of fields (Imagine  $\mathcal{U}$  being interested in counting the number of records where  $R_i.c_1 = \chi_1$  and  $R_i.c_2 = \chi_2$ ). Besides supporting conjunctive combinations, we extend EPiC to *disjunctive* combinations. Therewith, we generalize EPiC for any Boolean combination of the countable fields. The key technique for defining an indicator polynomial corresponding to an arbitrary Boolean expression among multiple fields is to transform Boolean operations to arithmetic operations, which is similar to *arithmetization* [3, 18].

**Conjunctive counting** Assume cloud user  $\mathcal{U}$  is interested in counting the number of records that have their  $m$  countable fields set to the pattern  $\chi = (\chi_1, \dots, \chi_m)$ . Here,  $\chi_k$  denotes the queried value in the  $k$ -th field ( $1 \leq k \leq m$ ). Let  $\varphi = (x_1 = \chi_1 \wedge \dots \wedge x_m = \chi_m)$  be the conjunction among  $m$  fields in the data set. User  $\mathcal{U}$  can now construct

$$P_\varphi(\mathbf{x}) = \prod_{k=1}^m P_{\chi_k}(x_k), \tag{5}$$

where  $\mathbf{x} = (x_1, \dots, x_m)$  denotes the variables in the multivariate polynomial  $P_\varphi(\mathbf{x})$ , and  $P_{\chi_k}(x_k)$  is the univariate indicator polynomial (as defined in Sect. 3.2) for counting  $\chi_k$  in the  $k$ -th field. Therewith,  $P_\varphi(\mathbf{x})$  satisfies the property of indicator polynomial as in Eq. (1), i.e.,  $P_\varphi(\mathbf{x})$  yields 1 only when  $\chi$  is matched. Note that the size of the multi-domain  $\mathcal{D}$  is  $|\mathcal{D}| = \prod_{k=1}^m |\mathcal{D}_k|$ , and the degree of  $P_\varphi(\mathbf{x})$  is  $\sum_{k=1}^m (|\mathcal{D}_k| - 1)$ . The product of univariate polynomials in Eq. (5) can be expanded to

$$\prod_{k=1}^m P_{\chi_k}(x_k) = \sum_{\forall \mathbf{j}=(j_1, \dots, j_m)} a_{\mathbf{j}} \cdot x_1^{j_1} \cdot x_2^{j_2} \cdots x_m^{j_m},$$

in which each coefficient  $a_{\mathbf{j}}$  is uniquely determined by  $a_{\mathbf{j}} = a_{1,j_1} \cdot a_{2,j_2} \cdots a_{m,j_m}$ , where  $a_{k,j_k}$  is a coefficient of the  $k$ -th univariate indicator polynomial  $P_{\chi_k}(x_k)$  corresponding to the exponent  $x_k^{j_k}$ . User  $\mathcal{U}$  prepares all coefficients  $a_{\mathbf{j}}$  of the indicator polynomial and sends their encryption  $E_{a_{\mathbf{j}}}$  to the cloud, which then computes the encrypted sum  $E_{\Sigma}$  and replies back to  $\mathcal{U}$  for decrypting the count.

**Disjunctive counting** Assume  $\mathcal{U}$ 's objective is to count the number of records that have value  $\chi_1$  in  $\mathcal{D}_1$  or value  $\chi_2$  in  $\mathcal{D}_2$ . The multivariate indicator polynomial for this disjunction is computed as

$$P_{\chi_1 \vee \chi_2}(\mathbf{x}) = P_{\chi_1}(x_1) + P_{\chi_2}(x_2) - P_{\chi_1 \wedge \chi_2}(\mathbf{x}), \tag{6}$$

where  $P_{\chi_1}(x_1)$ ,  $P_{\chi_2}(x_2)$  are univariate indicator polynomials for  $\mathcal{D}_1$ ,  $\mathcal{D}_2$ , respectively, and  $P_{\chi_1 \wedge \chi_2}(\mathbf{x})$  is a multivariate indicator polynomial for conjunctive counting between  $\mathcal{D}_1$  and  $\mathcal{D}_2$ . This method can be easily generalized to design counting query for disjunctions of  $m$  fields.

**Complement counting**  $\mathcal{U}$  can count records that do not satisfy a condition among fields by ‘‘flipping’’ the satisfying indicator polynomial:  $P_{\neg\varphi}(\mathbf{x}) = 1 - P_{\varphi}(\mathbf{x})$ .

**Integer range counting** Assume  $\mathcal{U}$  wants to count records having a field  $\mathcal{D}_k$  lying in an integer range  $[a, b]$ , i.e.,  $\varphi = (x_k = a \vee x_k = a + 1 \vee \cdots \vee x_k = b)$ . Based on disjunctive constructing method, we have  $P_{[a,b]}(x_k) = P_a(x_k) + P_{a+1}(x_k) + \cdots + P_b(x_k) - P_{a \wedge a+1} - \dots$ ; Since  $(x_k = u)$  and  $(x_k = v)$  are exclusive disjunctions for any  $u \neq v \in [a, b]$ ,  $P_{[a,b]}(x_k)$  reduces to:  $P_{[a,b]}(x_k) = \sum_{\chi_k=a}^b P_{\chi_k}(x_k)$ .

**Integer comparison counting** Integer comparisons can be constructed based on integer range counting:  $P_{\chi_k \leq a}(x_k) = P_{[0,a]}(x_k)$ , or  $P_{\chi_k > a}(x_k) = P_{[a+1, |\mathcal{D}_k|-1]}(x_k)$ . Our initial motivating example in Sect. 1 can be expressed as:

$$P_{\varphi}(\mathbf{x}) = \left( \sum_{\text{age}=50}^{70} P_{\text{age}}(\mathbf{x}) \right) \cdot (P_{\text{dia}=1}(\mathbf{x}) + P_{\text{hyp}=1}(\mathbf{x}) - P_{\text{dia}=1}(\mathbf{x}) \cdot P_{\text{hyp}=1}(\mathbf{x})).$$

**Privacy** Although the user-defined queries are different in construction, the encrypted queries  $Q$  always have exactly  $|\mathcal{D}| = \prod_{k=1}^m |\mathcal{D}_k|$  encrypted coefficients as we include zero coefficients also. As mentioned in Sect. 3.2, this prevents the cloud to differentiate queries based on query sizes.

**Efficiency** The user-side computation involving constructing the query’s coefficients is carried on plain-text before encryption, hence it introduces much lower computation cost compared to the computation burden on the cloud. To improve the user-side performance, one could apply optimizing techniques for reducing complex expressions, but this is out of scope of our work. To improve the cloud’s performance, we rearrange the order of computations for the sequence of encrypted fields  $E(R_i) = (E_{R_{i,1}}, \dots, E_{R_{i,m}})$  and coefficients  $a_{\mathbf{j}}$ ,  $\mathbf{j} = (j_1, \dots, j_m) \in \mathcal{D}$  to achieve  $E_{\Sigma}$  as

$$\begin{aligned} \sum_{i=1}^n P_{\chi}(E(R_i)) &= \sum_{i=1}^n \sum_{\mathbf{j} \in \mathcal{D}} E_{a_{\mathbf{j}}}(E_{R_{i,1}})^{j_1} \cdots (E_{R_{i,m}})^{j_m} \\ &= \sum_{\mathbf{j} \in \mathcal{D}} \left( E_{a_{\mathbf{j}}} \sum_{i=1}^n \prod_{k=1}^m (E_{R_{i,k}})^{j_k} \right) \end{aligned}$$

where  $a_{\mathbf{j}}$  are the query’s coefficients corresponding to monomials  $x_1^{j_1} \cdot x_2^{j_2} \cdots x_m^{j_m}$ .

### 3.4 Optimization through arithmetization in GF(2)

EPiC’s efficiency relies on the computations performed by the cloud. As discussed in Sect. 3.2, there are *no exponentiations* required for counting on a binary field. Consequently, we optimize EPiC by converting generic (non-binary) fields into multiple binary fields, thereby avoiding costly exponentiations. Note that as the conversion preserves Boolean expression output, results shown in Sect. 3.3 still hold, and protocol details discussed later in Sect. 3.6 remain unchanged.

Our idea is to store every generic field  $\mathcal{D}_k$  as separate binary fields  $\mathcal{D}_{k,1}, \mathcal{D}_{k,2}, \dots, \mathcal{D}_{k,\|\mathcal{D}_k\|}$ .<sup>2</sup> Therefore,  $m$  generic fields  $\mathcal{D}_1, \dots, \mathcal{D}_m$  become  $\sum_{k=1}^m \|\mathcal{D}_k\|$  binary fields  $\mathcal{D}_{1,1}, \dots, \mathcal{D}_{1,\|\mathcal{D}_1\|}, \dots, \mathcal{D}_{m,1}, \dots, \mathcal{D}_{m,\|\mathcal{D}_m\|}$ . The indicator polynomial for counting  $\chi_k$  in each field  $\mathcal{D}_k$  becomes

$$P_{\chi_{k,1} \wedge \dots \wedge \chi_{k,\|\mathcal{D}_k\|}}(x_{k,1}, \dots, x_{1,\|\mathcal{D}_k\|}) = \prod_{l=1}^{\|\mathcal{D}_k\|} P_{\chi_{k,l}}(x_{k,l}),$$

where  $x_{k,l}$  represents the  $l$ -th bit in the generic field  $\mathcal{D}_k$ , and  $\chi_{k,l}$  denotes the corresponding queried bit value. Applying arithmetization to “transform” from Boolean to multivariate polynomials, Boolean expressions of  $m$  generic fields can be converted into equivalent multiple binary fields. For convenience in later sections, we call the conversion to binary fields “GF(2) arithmetized” (shortly “G”), while the original is “Basic” (shortly “B”). We note that although the number of coefficients of the GF(2) arithmetized multivariate indicator polynomial corresponding to each query remains the same as in the generic case, the (multivariate) degree of the GF(2) arithmetized polynomial is much lower at

$$\deg(P^{(G)}) = \sum_{k=1}^m \|\mathcal{D}_k\| = \sum_{k=1}^m \lceil \log_2 |\mathcal{D}_k| \rceil \ll \sum_{k=1}^m (|\mathcal{D}_k| - 1) = \deg(P^{(B)}).$$

This implies a significant improvement for computational costs on the cloud. We refer to EPiC’s evaluation in Sect. 4 for details.

<sup>2</sup>  $\|X\| = \lceil \log_2 |X| \rceil$  denotes size in bits of  $X$ .

### 3.5 Encryption

Since EPiC’s indicator polynomial based counting technique involves additions and multiplications on ciphertexts, an appropriate homomorphic encryption scheme is needed as a building block. While there already exist various schemes, their computational complexities are often high [5, 8, 15, 24], rendering their use in current clouds impractical. Although EPiC can seamlessly integrate related work, we derive a somewhat homomorphic encryption scheme from the computational Private Information Retrieval (cPIR) technique of Trostle and Parrish [22]. The scheme is a secret key encryption scheme, where the cloud does not have the secret key to decrypt the data, but instead blindly performs operations on outsourced data. As we will see, this scheme does not enjoy the same security properties, i.e., IND-CPA, as related work, but only security with respect to Definitions 3 and 4 as required in the specific context of EPiC. Due to its weaker security properties, our scheme is especially practical in the settings we target.

Again, we stress that any (somewhat) homomorphic encryption scheme can be used as an exchangeable building block in EPiC. In the time between submission and publication of this work, new research results have pointed out security issues with the Hidden Modular Group Order assumption [16], and future work might consider adopting, e.g. NTRU [12] for efficient (somewhat) homomorphic encryption.

#### 3.5.1 Encryption scheme

- **Key generation**— $\text{KEYGEN}(s_1, s_2, n, \mathcal{D})$ : Parameters  $s_1, s_2 \in \mathbb{N}$  are security parameters,  $n \in \mathbb{N}$  is the upper bound for the total number of records in the data set, and  $\mathcal{D} = \mathcal{D}_1 \times \dots \times \mathcal{D}_m$  is the multi-domain of  $m$  countable fields.  $\text{KEYGEN}$  computes a random prime  $q$ , a random prime  $p$ , and a random (maybe non-prime)  $b \in \mathbb{Z}_p$ . The secret key, the output of  $\text{KEYGEN}$ , is defined as  $K := \{p, b\}$ . The prime  $p$  and  $q$  are generated based on the following conditions (cf. Sect. 3.5.3):

$$q > n \tag{7}$$

$$\|p\| \geq s_1 + \|n\| + \|q\| + \sum_{k=1}^m (s_2 + \|q\|) \cdot (|\mathcal{D}_k| - 1). \tag{8}$$

- **Encryption**— $\text{ENC}(\mathcal{P})$ : Selects a random number  $r$ ,  $\|r\| \leq s_2$ , and encrypts the plaintext  $\mathcal{P}$  to  $\mathcal{C} = \text{ENC}(\mathcal{P}) := b \cdot (r \cdot q + \mathcal{P}) \bmod p$ .
- **Decryption**— $\text{DEC}(\mathcal{C})$ : Decrypts  $\mathcal{C}$  to  $\mathcal{P} = \text{DEC}(\mathcal{C}) := b^{-1} \cdot \mathcal{C} \bmod p \bmod q$ .
- **Security** The security of our encryption scheme (cf. Sect. 3.7) is based on the Hidden Modular Group Order hardness assumption and the cPIR protocol in [22]. The rationale is that, for appropriate security parameters, more than half of the bits of  $p$  are still secret against any PPT adversary; and if a PPT adversary can break the cPIR protocol, the Hidden Group Order  $p$  is also revealed, violating the assumption.

### 3.5.2 Arithmetic

The addition and multiplication operations on ciphertexts take place in the integers. There is no modulo reduction, as the cloud does not know  $p$ . One can verify that this scheme provides additively and multiplicatively homomorphic properties. Let  $\|\mathcal{C}\|$  denote the size in bits of a ciphertext  $\mathcal{C}$ . The size of ciphertext grows after each operation (thus, this scheme is only somewhat homomorphic).

- Addition:  $\|\mathcal{C}_1 + \mathcal{C}_2\| = \begin{cases} \|\mathcal{C}_1\| + 1, & \text{if } \|\mathcal{C}_1\| = \|\mathcal{C}_2\|. \\ \max(\|\mathcal{C}_1\|, \|\mathcal{C}_2\|), & \text{otherwise.} \end{cases}$
- Multiplication:  $\|\mathcal{C}_1 \cdot \mathcal{C}_2\| = \|\mathcal{C}_1\| + \|\mathcal{C}_2\| - 1$ .
- Scalar multiplication:  $\|n \cdot \mathcal{C}\| = \|n\| + \|\mathcal{C}\| - 1$ .
- Exponentiation:  $\|\mathcal{C}^j\| = j \cdot \|\mathcal{C}\| - (j - 1)$ .

Note that a multiplication of ciphertexts will increase the exponent of  $b$  in the result. Therefore, in general, a decryption of a ciphertext after  $(j - 1)$  multiplications has to be  $\mathcal{P}^j = \text{DEC}(\mathcal{C}^j, j) := b^{-j} \cdot \mathcal{C}^j \bmod p \bmod q$ . Note the soundness of our scheme:

$$\begin{aligned} \text{DEC}(\mathcal{C}^j, j) &= b^{-j} \cdot \mathcal{C}^j \bmod p \bmod q = b^{-j} \cdot [b \cdot (r \cdot q + \mathcal{P})]^j \bmod p \bmod q \\ &= (r \cdot q + \mathcal{P})^j \bmod q = \mathcal{P}^j. \end{aligned}$$

So, contrary to related work, decryption in our scheme requires a number of multiplications performed on the ciphertext. Also, EPiC’s encryption scheme does not allow addition of two ciphertexts that have different exponents of  $b$ . In the particular context of EPiC, we can accept these limitations, and we gain high computation efficiency in return.

### 3.5.3 Selection of $p$ and $q$

Since ciphertexts increase for every multiplication and addition, this scheme requires a careful selection of  $q$  and  $p$  in advance such that the decryption is sound. Let  $e(\mathcal{P}) = r \cdot q + \mathcal{P}$ . Consider a ciphertext  $\mathcal{C} = \text{ENC}(\mathcal{P}) = b \cdot e(\mathcal{P}) \bmod p$  and its decryption procedure:  $\text{DEC}(\mathcal{C}) = b^{-1} \cdot \mathcal{C} \bmod p \bmod q = e(\mathcal{P}) \bmod p \bmod q \stackrel{(a)}{=} e(\mathcal{P}) \bmod q \stackrel{(b)}{=} \mathcal{P}$ . For the equalities (a) and (b) to hold, the plaintext  $\mathcal{P}$  needs to satisfy: 1)  $\mathcal{P} \in GF(q)$  for (b) to hold; 2)  $e(\mathcal{P}) \in GF(p)$  for (a) to hold.

In EPiC, user  $\mathcal{U}$  receives the encrypted final sum  $E_\Sigma$  from the cloud. In this context,  $\mathcal{P} = \sigma = \text{DEC}(E_\Sigma)$  is the plaintext count to be decrypted from  $E_\Sigma$ . The first condition requires that  $\sigma \in GF(q)$ . Since  $\sigma$  is the count value, it is at most the number of records in the data set, i.e.,  $\sigma \leq n$ . This implies  $q > n$  as in Eq. (7).

Since  $e(\sigma) = \sum_{j \in \mathcal{D}} (e(a_j) \cdot \sum_{i=1}^n \prod_{k=1}^m (e(R_{i,k}))^{j_k})$  and the second condition requires  $e(\sigma) \in GF(p)$ . Therewith, we establish the lower bound on the size of  $p$  as  $\|p\| \geq \|n\| + \|q\| + \sum_{k=1}^m (s_2 + \|q\|) \cdot (|\mathcal{D}_k| - 1)$ . For the Hidden Modular Group Order assumption, a security parameter  $s_1$  has to be added to the size of  $p$  [22], therewith yielding the requirement for  $p$  as in Eq. (8).

### 3.5.4 Encrypting data and query

We employ this encryption scheme to encrypt both countable fields and query. Using the secret key  $K = \{p, b\}$  returned from `KEYGEN`, the user encrypts each countable field  $R_{i,k}$  to  $E_{R_{i,k}} = \text{ENC}(R_{i,k})$  before uploading his encrypted data to the cloud. The encryption of a counting query  $Q$ , as a sequence of encrypted coefficients  $E_{a_j}$ , however, is more difficult. A simple method of encrypting  $a_j$  to  $E_{a_j} = \text{ENC}(a_j)$  does not allow the decryption of the final sum  $E_\Sigma$  of Eq. (7) due to different exponents of  $b$  in the multivariate monomials, explained below.

Consider the cloud computation in Eq. (7). For every  $\mathbf{j} = (j_1, \dots, j_m) \in \mathcal{D}$ ,  $j_k \in \mathcal{D}_k$ , the corresponding multivariate monomial  $\prod_{k=1}^m (E_{R_{i,k}})^{j_k}$  has a multi-degree dependent on  $\mathbf{j}$ . More precisely, let  $d_1(\mathbf{j}) = \text{deg} \prod_{k=1}^m (E_{R_{i,k}})^{j_k}$  denote the multi-degree of the multivariate monomial  $\prod_{k=1}^m (E_{R_{i,k}})^{j_k}$ , then  $d_1(\mathbf{j}) = \sum_{k=1}^m j_k$ . If  $a_j$  was simply encrypted similarly to the encryption of countable fields, i.e.,  $E_{a_j} = \text{ENC}(a_j)$ , each product  $E_{a_j} \cdot \sum_{i=1}^n \prod_{k=1}^m (E_{R_{i,k}})^{j_k}$  would contain  $b$  of different degree equal to  $1 + d_1(\mathbf{j})$ . This would prohibit additions among them to obtain the *decryptable* final sum  $E_\Sigma$ , as our encryption scheme only allows adding ciphertexts containing the same exponents of  $b$ . Our approach is to “augment” the exponents of  $b$  in the coefficients. Specifically, we encrypt  $a_j$  to

$$E_{a_j} := \text{ENC}(a_j) \cdot (\text{ENC}(1))^{d_2(\mathbf{j})} = b^{d_2(\mathbf{j})+1} \cdot e(a_j) \pmod p, \tag{9}$$

where  $d_2(\mathbf{j}) = \sum_{k=1}^m (|\mathcal{D}_k| - j_k - 1)$ . Therewith,  $E_\Sigma$  will contain  $b$  of degree  $d = 1 + d_1(\mathbf{j}) + d_2(\mathbf{j}) = 1 + \sum_{k=1}^m (|\mathcal{D}_k| - 1)$ , which is independent of  $\mathbf{j}$ . All multivariate monomials now have the same exponents of  $b$ , allowing successful decryption of  $E_\Sigma$ .

### 3.6 Detailed protocol description

With all ingredients ready, we now describe EPiC using the notation of Sect. 2.1.

`KEYGEN`( $\kappa$ ): Based on security parameter  $\kappa$ , cloud user  $\mathcal{U}$  chooses  $s_1, s_2$  for the somewhat homomorphic encryption, determines an upper bound  $n$  for the total number of records that might be stored and the appropriate multi-domain  $\mathcal{D}$  for the countable fields.  $\mathcal{U}$  generates a secret key  $K$  from the somewhat homomorphic encryption `KEYGEN`( $s_1, s_2, n, \mathcal{D}$ ) and a symmetric key  $K'$  for a block cipher such as AES used for non-countable data. The secret key  $\mathcal{S} := \{K, K'\}$  is used throughout EPiC.

`ENCRYPT`( $\mathcal{S}, \mathcal{R}$ ): Assume  $\mathcal{U}$  wants to store  $n$  records  $\mathcal{R} = \{R_1, \dots, R_n\}$ . Each record  $R_i$  is encrypted separating the countable values  $R_{i,k}$  from the rest of the record.  $R_{i,k}$  is encrypted using the somewhat homomorphic encryption mechanism, i.e.,  $E_{R_{i,k}} := \text{ENC}(\{p, b\}, R_{i,k})$ . For the rest of the record  $R_i$ , a random initialization vector  $IV$  is chosen and the record is  $\text{AES}_{K'}\text{-CBC}$  encrypted. In conclusion, a record  $R_i$  encrypts to  $E_{R_i} := \{E_{R_{i,1}}, \dots, E_{R_{i,m}}, IV, \text{AES}_{K'}\text{-CBC}(R_{i,rest})\}$ . The output of `ENCRYPT` is the sequence of encrypted records.  $\mathcal{E} := \{E_{R_1}, \dots, E_{R_n}\}$ .

`UPLOAD`( $\mathcal{E}$ ): Upload simply sends all records as one large file to the MapReduce cloud where the file is automatically split into *InputSplits*.

---

**Algorithm 1:** PROCESSQUERY

---

For each Mapper  $M$ :

```

init  $s_j := 0, \forall j \in \mathcal{D}$ 
forall  $E_{R_i}$  in  $InputSplit(M)$  do
  read  $\{E_{R_{i,1}}, \dots, E_{R_{i,m}}\}$ 
  forall  $j = (j_1, \dots, j_k) \in \mathcal{D}$  do
     $s_j := s_j + \prod_{k=1}^m (E_{R_{i,k}})^{j_k}$ 
  end
end
emit  $\{j, s_j\}, \forall j \in \mathcal{D}$ 

```

Reducer  $R$ :

```

init  $E_{\Sigma} := 0, S_j := 0, \forall j \in \mathcal{D}$ 
forall  $\{j, s_j\}$  in  $MappersOutput$  do
   $S_j := S_j + s_j$ 
end
forall  $j$  in  $\mathcal{D}$  do
   $E_{\Sigma} := E_{\Sigma} + E_{a_j} \cdot S_j$ 
end
write  $\{E_{\Sigma}\}$ 

```

---

PREPAREQUERY( $\mathcal{S}, \chi$ ): To prepare a query for  $\chi$ ,  $\mathcal{U}$  computes the  $|\mathcal{D}|$  coefficients  $a_j, j \in \mathcal{D}$ , of the indicator polynomial  $P_{\chi}(\mathbf{x})$  as described in Sect. 3.3. Coefficients  $a_j$  are encrypted and sent to the cloud. The cloud will be using these coefficients to perform the evaluation of  $P_{\chi}(\mathbf{x})$ . Consequently in EPiC, the output  $Q$  of PREPAREQUERY sent to the cloud is  $Q := \{E_{a_j}, j \in \mathcal{D}\}$ .

PROCESSQUERY( $Q, \mathcal{E}$ ): Based on the data set size and the cloud configuration, the MapReduce framework selects  $M$  Mapper nodes and 1 Reducer node. Algorithm 1 depicts the specification of EPiC’s *map* and *reduce* functions that will be executed by the cloud. In the mapping phase, for each input record  $R_i$  in their locally stored InputSplits, the Mappers compute in parallel all monomials  $\prod_{k=1}^m (E_{R_{i,k}})^{j_k}$  of the countable fields and add the same-degree monomials together. After the Mappers finish scanning over all records, the sums  $s_j$  of monomials are output as key-value pairs. These pairs contain the multi-degree  $j$  as key, and the computed sum  $s_j$  as value. In MapReduce, the output of the Mappers is then automatically sent (“emitted”) to the Reducer. Based on the sums received from all Mappers, the Reducer combines them together to obtain the *global* sums  $S_j$ , i.e., the sums over all records in the data set. In a last step, the Reducer uses the coefficients  $E_{a_j}$  received from  $\mathcal{U}$  to evaluate the polynomial by computing the inner product with the global sums. The result  $E_{\Sigma}$  is sent back to  $\mathcal{U}$  and can be decrypted to obtain the count value.

DECODE( $\mathcal{S}, E_{\Sigma}$ ):  $\mathcal{U}$  receives  $E_{\Sigma}$  and computes the counting result  $\sigma = \text{DEC}(E_{\Sigma})$ .

**3.7 Privacy analysis**

We now formally prove Storage and Counting privacy for EPiC and its underlying encryption. We stress that, below, we neither target nor prove that our encryption provides traditional IND-CPA security. Instead, we show that, in combination with other details of our protocol, it provides security according to Definitions 3 and 4.

**Lemma 1** (Storage privacy) *Based on the security of the cPIR scheme by Trostle and Parrish [22], EPiC preserves storage privacy.*

*Proof* cPIR-security by Trostle and Parrish [22] can be summarized as follows. With a  $u \times u$  bit database, a user wants to retrieve an  $y$ -th row and sends an encrypted PIR request to the cloud:  $P = \{E_{v_1}, \dots, E_{v_u}\}$ , where  $E_{v_k} = \text{ENC}(v_k)$ , cf. Sect. 3.5,

and  $v_k = 1$ , if  $k = y$ , and  $v_k = 0$  otherwise. This cPIR protocol is secure *iff* for all PPT adversaries  $\mathcal{A}^*$ , the probability of finding  $y$  is negligible more than guessing, i.e.,  $\Pr[\mathcal{A}^*(P) = y] \leq 1/u + \epsilon^*(\kappa)$ . We now prove our lemma by reduction from cPIR security. We show that, for security parameter  $\kappa$ , any PPT  $(t(\kappa), \epsilon(\kappa))$ -adversary  $\mathcal{A}$  breaking EPiC’s storage privacy (Definition 3) in  $t(\kappa)$  steps with non-negligible advantage  $\epsilon(\kappa)$  can be used to construct a  $(t^*(\kappa), \epsilon^*(\kappa))$ -adversary  $\mathcal{A}^*$  as a subroutine breaking the cPIR protocol in [22]. We construct  $\mathcal{A}^*$  based on the *parity* of  $u$ .

**1.  $u$  is odd.** First,  $\mathcal{A}^*$  receives as input the PIR request  $P$  and splits  $P$  into two halves  $\mathcal{E} = \{E_{v_1}, \dots, E_{v_{\lfloor u/2 \rfloor}}\}$ ,  $\mathcal{E}' = \{E_{v_{\lfloor u/2 \rfloor + 1}}, \dots, E_{u-1}\}$ , i.e., treating the PIR request as two EPiC data sets of the same size ( $\lfloor u/2 \rfloor$  records). Since  $E_{v_k}$  are either encryptions of 0 or 1,  $\mathcal{E}$  and  $\mathcal{E}'$  are now viewed as single-binary-field data sets, where each record contains only 1 countable binary field.  $\mathcal{A}^*$  randomly selects  $l_1, l_2, l'_1, l'_2 \in [1, u]$  and creates two EPiC counting queries  $Q = \{E_{v_{l_1}}, E_{v_{l_2}}\}$ ,  $Q' = \{E_{v_{l'_1}}, E_{v_{l'_2}}\}$ . These are two valid queries, because for single-binary-field data sets  $\mathcal{E}$ ,  $\mathcal{E}'$ , any EPiC query contains exactly 2 encrypted coefficients of 0 or 1, cf. Sect. 3.3. Then  $\mathcal{A}^*$  runs PROCESSQUERY on  $\mathcal{E}$  with  $Q$ , and  $\mathcal{E}'$  with  $Q'$ , thereby obtaining  $E_\Sigma$  and  $E'_\Sigma$ .  $\mathcal{A}^*$  forwards  $I = \{\mathcal{E}, \mathcal{E}', Q, Q', E_\Sigma, E'_\Sigma\}$  to  $\mathcal{A}$ .  $\mathcal{A}^*$ ’s output depends on  $\mathcal{A}$ ’s output as follows.

If  $\mathcal{A}$  outputs 0,  $\mathcal{A}^*$  outputs  $u$ . The intuition is that, since  $\mathcal{A}$  “believes” the two halves  $\mathcal{E}$  and  $\mathcal{E}'$  are the same,  $\mathcal{A}$  concludes that the requested element must not belong to either  $\mathcal{E}$  or  $\mathcal{E}'$ , i.e.,  $v_u = 1$ . If  $\mathcal{A}$  outputs 1,  $\mathcal{A}^*$  randomly selects  $k \in [1, u - 1]$  and outputs  $k$ . The intuition is that “ $\mathcal{A}$  outputs 1” indicates the requested row index is between 1 and  $u - 1$ , and  $\mathcal{A}^*$  simply makes a random guess for it. The probability for  $\mathcal{A}^*$  to output correctly is  $\Pr[\mathcal{A}^*(P) = y] = \Pr[\mathcal{A} = 0 | y = u] \cdot \Pr[y = u] + \Pr[\mathcal{A} = 1, k = y | y < u] \cdot \Pr[y < u] = \left(\frac{1}{2} + \epsilon(\kappa)\right) \cdot \frac{1}{u} + \left(\frac{1}{2} + \epsilon(\kappa)\right) \cdot \frac{u-1}{u} = \frac{1}{u} + \frac{2\epsilon(\kappa)}{u}$ . Therewith,  $\mathcal{A}^*$  has a non-negligible advantage of  $\epsilon^*(\kappa) = 2\epsilon(\kappa)/u$  in finding  $y$ .

**2.  $u$  is even.**  $\mathcal{A}^*$  makes a new PIR request  $P'$  by removing the last element  $v_u$  from  $P$ , that is  $P' = \{E_{v_1}, \dots, E_{v_{u-1}}\}$ . Then  $\mathcal{A}^*$  uses the same approach as above for  $P'$ , i.e., splitting  $P'$  into 2 halves, feeding both to  $\mathcal{A}$ . Now,  $\mathcal{A}^*$  outputs  $u - 1$ , if  $\mathcal{A}$  outputs 0, or outputs random  $k \in [1, u - 2]$  otherwise. It can be observed that  $\mathcal{A}^*$  can find  $y$  with non-negligible probability, only if  $y \neq u$ , i.e., the requested element is not the last element discarded from  $P$ . Otherwise,  $\mathcal{A}^*$  cannot find  $y$ . More precisely, the probability of correct guess is  $\Pr[\mathcal{A}^*(P) = y] = \Pr[\mathcal{A}^*(P') = y | y < u] \cdot \Pr[y < u] + \Pr[\mathcal{A}^*(P') = y | y = u] \cdot \Pr[y = u] = \left(\frac{1}{u-1} + \frac{2\epsilon(\kappa)}{u-1}\right) \cdot \frac{u-1}{u} + 0 \cdot \frac{1}{u} = \frac{1}{u} + \frac{2\epsilon(\kappa)}{u}$ . Therefore,  $\mathcal{A}^*$  also has a non-negligible advantage of  $2\epsilon(\kappa)/u$  in finding  $y$ .

Consequently, in both cases,  $\mathcal{A}^*$  has a non-negligible advantage  $\epsilon^*(\kappa) = 2\epsilon(\kappa)/u$  of breaking the cPIR protocol in  $t^*(\kappa) = t(\kappa)$  steps, rendering our reduction tight. □

**Lemma 2** (Counting privacy) *Based on the security of the cPIR scheme by Trostle and Parrish [22], EPiC preserves counting privacy.*

*Proof* We prove our lemma by reduction from cPIR security. Recall the cPIR-security definition as in Lemma 1’s proof. We assume the existence of a PPT  $(t(\kappa), \epsilon(\kappa))$ -EPiC-adversary  $\mathcal{A}$  breaking EPiC’s counting privacy (Definition 4) in  $t(\kappa)$  steps with non-negligible advantage  $\epsilon(\kappa)$ . In the following, we construct a new  $(t^*(\kappa), \epsilon^*(\kappa))$ -PIR-adversary  $\mathcal{A}^*$  that breaks this cPIR security.

$\mathcal{A}^*$  receives as input the PIR request  $P = \{E_{v_1}, \dots, E_{v_u}\}$ , where  $v_y = 1$  and  $v_k = 0, \forall k \neq y$ . The goal of  $\mathcal{A}^*$  is to guess  $y$ . First,  $\mathcal{A}^*$  sets  $\mathcal{E} = \mathcal{E}' = P$  and randomly picks 4 elements  $E_{l_1}, E_{l_2}, E_{l'_1}, E_{l'_2}$  from  $P$  to make two EPiC queries  $Q = \{E_{l_1}, E_{l_2}\}, Q' = \{E_{l'_1}, E_{l'_2}\}$ . Note that  $\mathcal{E}, \mathcal{E}'$  can be viewed as EPiC's two identical single-binary-field data sets, and  $Q, Q'$  are valid queries (corresponding to some patterns  $\chi, \chi'$ ) for  $\mathcal{E}, \mathcal{E}'$ . Then  $\mathcal{A}^*$  runs PROCESSQUERY on  $\mathcal{E}$  with  $Q$  and on  $\mathcal{E}'$  with  $Q'$  to obtain  $E_\Sigma, E'_\Sigma$ . Now,  $\mathcal{A}^*$  forwards  $I = \{\mathcal{E}, \mathcal{E}', Q, Q', E_\Sigma, E'_\Sigma\}$  to  $\mathcal{A}$  and observes  $\mathcal{A}$ 's output.

Let  $U = \{1, \dots, u\}, L = \{l_1, l_2, l'_1, l'_2\}$ . If  $\mathcal{A}$  returns 1,  $\mathcal{A}^*$  concludes that the two queries  $Q$  and  $Q'$  are identical, implying that  $E_{v_y} \notin Q \cup Q'$ , i.e.,  $y \notin L$ . Therewith,  $\mathcal{A}^*$  makes a guess for  $y$  by selecting a random  $k \in U \setminus L$  and outputs  $k$ . Otherwise, if  $\mathcal{A}$  returns 0,  $\mathcal{A}^*$  concludes that  $v_y$  is in either  $Q$  or  $Q'$ , thus  $\mathcal{A}^*$  outputs a random  $k \in L$ . The probability of correct guess is  $\Pr[\mathcal{A}^*(P) = y] = \Pr[\mathcal{A}(I) = 1, k = y | y \in U \setminus L] \cdot \Pr[y \in U \setminus L] + \Pr[\mathcal{A}(I) = 0, k = y | y \in L] \cdot \Pr[y \in L] = (\frac{1}{2} + \epsilon(\kappa)) \cdot \frac{1}{u-4} \cdot \frac{u-4}{u} + (\frac{1}{2} + \epsilon(\kappa)) \cdot \frac{1}{4} \cdot \frac{4}{u} = \frac{1}{u} + \frac{2\epsilon(\kappa)}{u}$ . That is  $\mathcal{A}^*$  has a non-negligible advantage  $\epsilon'(\kappa) = 2\epsilon(\kappa)/u$  of breaking the cPIR protocol in  $t^*(\kappa) = t(\kappa)$  steps. □

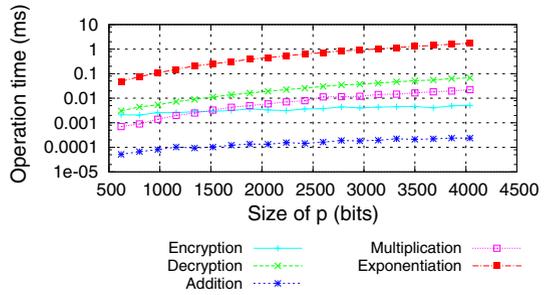
### 4 Evaluation

To show its real-world applicability, we have implemented EPiC in Hadoop's MapReduce framework v1.0.3 [2], and evaluated it on Amazon's public MapReduce cloud [1]. EPiC implementation is written in Java, and all cryptographic operations are *unoptimized*, relying on Java's standard BigInteger data type. Still, exponentiation, e.g.  $C^j$ , with  $j = 15$  and  $|\mathcal{C}| \approx 4000$  takes  $< 2\text{ms}$  on a 1.8GHz Intel Core i7 laptop, a single addition is not measurable with  $< 1\mu\text{s}$ . Figure 1 shows a benchmark of various operations on the ciphertexts using our encryption scheme. In our evaluation, we use security parameters  $s_1 = 400$  bits for good security [22], and  $s_2 = |r| = 160$  bits. We have implemented a data generator program to randomly generate patient records with  $m$  countable fields with size between 4 and 10 bits.

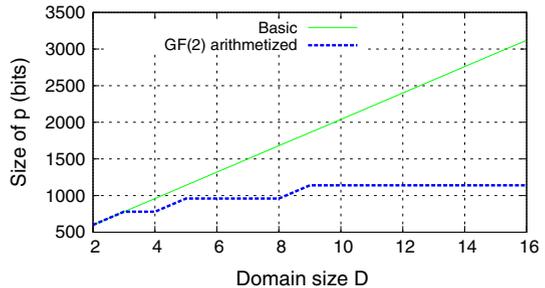
We have evaluated the performance of EPiC by comparing our "Basic" and "GF(2) arithmetized" solutions with a "non-privacy-preserving" solution. Unless otherwise stated, the single/multi-domain size in both "Basic" and "GF(2) arithmetized" solutions is always set to the same value  $|\mathcal{D}|$  for comparison. For brief presentation, we use subscript "B" for Basic, and "G" for GF(2) arithmetized approach, e.g.,  $\|p_B\|, \|p_G\|$  indicate the size in bits of  $p$  in Basic, GF(2) arithmetized approach respectively. We also set  $u = s_1 + \|n\| + \|q\|, v = s_2 + \|q\|$  as fixed parameters.

**Size of prime  $p$**  As discussed in Sect. 3.5, prime  $q$  depends only on the number of records  $n$ , while prime  $p$  also depends on  $|\mathcal{D}|$ . We show the benefit of the GF(2) arithmetized approach ( $m = \|\mathcal{D}\|, |\mathcal{D}_k| = 2$ ) by demonstrating that a conversion to multiple binary fields reduces  $\|p\|$  significantly to  $\|p_G\| = u + \|\mathcal{D}\| \cdot v$ , while the Basic approach ( $m = 1, |\mathcal{D}_1| = |\mathcal{D}|$ ) requires that  $\|p_B\| = u + (|\mathcal{D}| - 1) \cdot v$ . Figure 2 shows  $\|p\|$ 's logarithmic increase with GF(2) arithmetized and linear increase with Basic approach.

**Fig. 1** Computation time of various operations on ciphertexts using our encryption scheme. The exponentiation time is measured for  $C^{15}$



**Fig. 2** Size of  $p$  depends on size of domain  $\mathcal{D}$



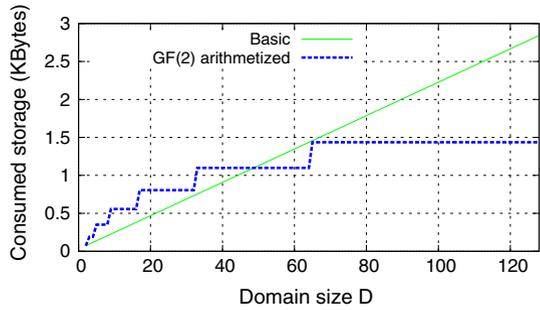
**Storage cost** The storage cost depends on the size of the data stored on the cloud, which is determined by the size of  $p$ . In Basic approach, a generic field of domain  $\mathcal{D}$  requires a storage of  $S_B = \|p_B\| = u + (|\mathcal{D}| - 1) \cdot v$  bits. In GF(2) arithmetized approach, the equivalent multiple binary fields requires a storage of  $S_G = \|\mathcal{D}\| \cdot \|p_G\| = \|\mathcal{D}\| \cdot (u + \|\mathcal{D}\| \cdot v)$  bits. Again, in Fig. 3, we see a linear increase of storage in Basic, and logarithmic increase in GF(2) arithmetized approach.

**User computation cost**  $\mathcal{U}$  prepares the query in plaintext, which incurs very low computation cost compared to ciphertext operations performed on the cloud. Encrypting one coefficient takes about 1 ms (Fig. 1), resulting in roughly  $|\mathcal{D}|$  ms for encrypting all  $|\mathcal{D}|$  coefficients of the query, regardless of using Basic or GF(2) arithmetized.

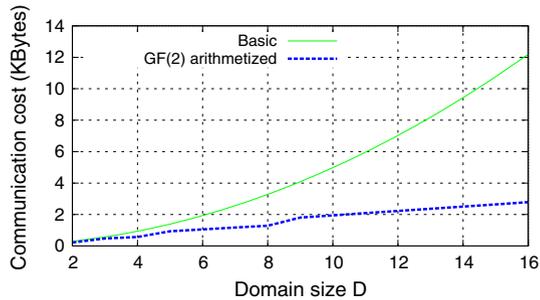
**Communication cost** Due to oblivious counting, user  $\mathcal{U}$  prepares and sends all  $|\mathcal{D}|$  coefficients corresponding to *all* monomials to the cloud. The total size of the encrypted coefficients is  $|\mathcal{D}| \cdot \|p\|$ . In Basic approach, the query size is  $Q_B = |\mathcal{D}| \cdot \|p_B\| = |\mathcal{D}| \cdot (u + (|\mathcal{D}| - 1) \cdot v)$ . In contrast, the GF(2) arithmetized approach reduces to  $Q_G = |\mathcal{D}| \cdot \|p_G\| = |\mathcal{D}| \cdot (u + \|\mathcal{D}\| \cdot v)$ . For example of a data set containing  $n = 10^6$  records with a countable field of domain size  $|\mathcal{D}| = 1024$  (i.e.,  $\|\mathcal{D}\| = 10$ ), the corresponding query size in each approach is  $Q_B = 22.5$  MBytes, and  $Q_G = 280$  KBytes, respectively.

The answer size (size in bits of the received ciphertext as final sum) depends on the maximum size of the multivariate monomial. The monomial size is determined by the ciphertext size (i.e.,  $\|p\|$ ) and the number of performed multiplications, i.e., its multi-degree. Let  $d$  denote the maximum multi-degree of monomials, then,  $d = |\mathcal{D}|$  in the Basic approach, and  $d = \|\mathcal{D}\|$  in the GF(2) arithmetized approach. We have

**Fig. 3** Consumed storage for each field depends on size of domain  $\mathcal{D}$



**Fig. 4** Communication cost depends on size of domain  $\mathcal{D}$



$A_B = |\mathcal{D}| \cdot \|p_B\| = |\mathcal{D}| \cdot (u + (|\mathcal{D}| - 1) \cdot v)$  and  $A_G = \|\mathcal{D}\| \cdot \|p_G\| = \|\mathcal{D}\| \cdot (u + \|\mathcal{D}\| \cdot v)$ . For example of a data set of  $n = 10^6$  records with a countable field of  $|\mathcal{D}| = 1024$ , the answer size in each approach is  $A_B = 22.5$  Mbytes, and  $A_G = 2.7$  KBytes, respectively.

*Total transfer cost:* The total communication cost,  $C = Q + A$ , as shown in Fig. 4, is much less in GF(2) arithmetized approach than in Basic approach:

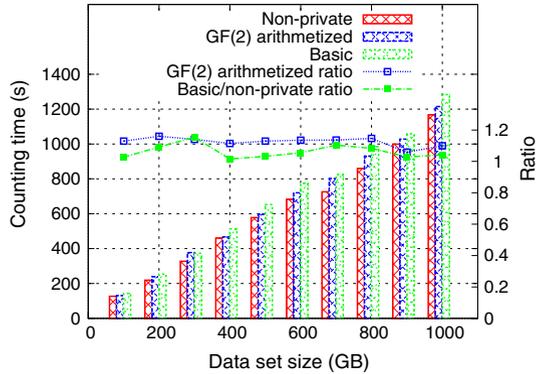
$$C_B = Q_B + A_B = 2 \cdot |\mathcal{D}| \cdot (u + (|\mathcal{D}| - 1) \cdot v),$$

$$C_G = Q_G + A_G = (|\mathcal{D}| + \|\mathcal{D}\|) \cdot (u + \|\mathcal{D}\| \cdot v).$$

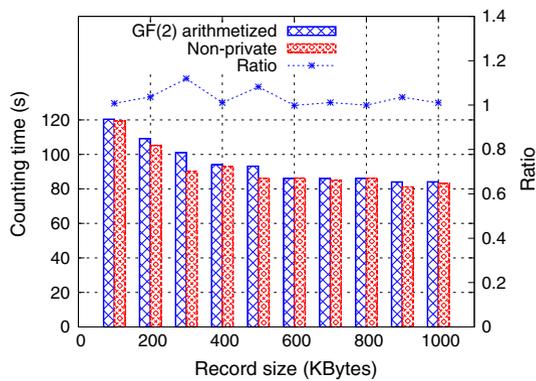
**Cloud computation** We have evaluated the cloud computation cost for large-scale data sets on Amazon’s public cloud. As Amazon imposes an (initial) limit of 20 instances per job, we restrict ourselves to 20 Standard Large On-Demand instances [1]. Each instance comprises 4 2.27GHz Intel Xeon CPUs and a total of 7.5 GB RAM.

*Variable data set size* First, we fix the size of each record to 1 MB. The data set size (x-axis) is varied from 100 GB to 1 TB. We query a countable field of size  $|\mathcal{D}| = 16$ . Figure 5 shows the average counting time for a MapReduce job on the whole data set of different sizes. The y-axis shows the total time for MapReduce to evaluate the user’s query. This is the time that a user has to pay for to Amazon. To put our results into perspective, we not only show the time for both Basic and GF(2) arithmetized approaches, but as well the time a “non-privacy-preserving” counting would take, i.e., the countable field is not encrypted and directly counted. Moreover, we also show

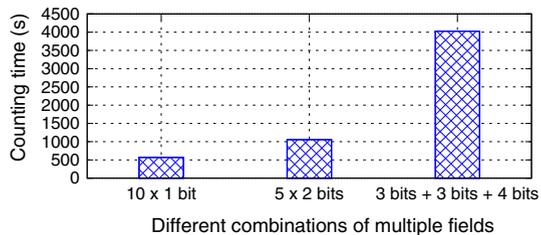
**Fig. 5** Counting time versus data set size ( $|\mathcal{D}| = 16$ )



**Fig. 6** 50 GB data sets, varying record size ( $|\mathcal{D}| = 16$ )



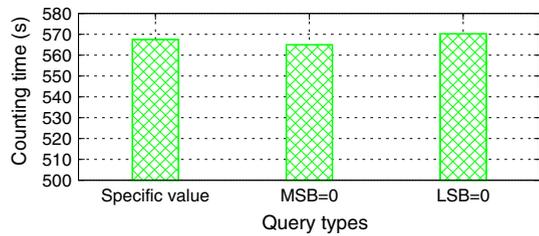
**Fig. 7** Effect of different field combinations



the overhead ratio between EPiC’s two approaches and non-private counting. The additional overhead introduced by EPiC over non-private counting is less than 20%. We conjecture that only 20% overhead/additional cost over non-privacy-preserving counting is acceptable in many real-world situations, rendering EPiC practical.

*Variable record size* To also evaluate the effect of the size of the records on the general performance, we run the system with a fixed data set size of 50 GB. The record size is changed from 100 KB to 1 MB. Figure 6 shows that, while IO time remains unchanged, a higher number of records increases counting time in EPiC. However, the overhead of EPiC is still under 20% even for small record sizes such as 100 KB compared to non-private counting. That is, EPiC is efficient even for small patient records.

**Fig. 8** Different query types on the same data set



*Effect of multiple fields* To study the efficiency of transforming a single countable field  $\mathcal{D}$  into multiple fields of different size, we conduct an experiment on a data set size of 100 GB. The total domain size is set to  $|\mathcal{D}| = 1024$  (10 bits). We compare three cases: (a) transform  $\mathcal{D}$  into 10 single binary fields; (b) transform  $\mathcal{D}$  into 5 quaternary fields each of 2 bits; (c) transform  $\mathcal{D}$  into 3 fields of 3 bits, 3 bits, and 4 bits, respectively. In Fig. 7, we can see that the GF(2) arithmetized approach yields the best performance.

*Query types* Finally, to evaluate the effects of different query types on the performance, we run EPiC with a fixed data set of 100 GB. Total domain size is  $|\mathcal{D}| = 1024$ . We make 3 different queries: (a) query for a specific value; (b) query for the MSB of the field equal to 0; (c) query for the LSB of the field equal to 0. Figure 8 demonstrates that there is no significant difference in counting time between different queries.

## 5 Related work

Protecting privacy of outsourced data and delegated operations in a cloud computing environment is the perfect setting for fully homomorphic encryption. While there is certainly a lot of ongoing research in fully homomorphic encryption (see Vaikuntanathan [23] for an overview), current implementations indicate high storage and computational overhead [9], rendering fully homomorphic encryption impractical for the cloud.

Similar to EPiC, Lauter et al. [15] observe that often weaker “somewhat” homomorphic encryption might be sufficient. Lauter et al. [15]’s scheme is based on a protocol for lattice-based cryptography by Brakerski and Vaikuntanathan [5]. However, for the specific application scenario considered in this paper, EPiC’s somewhat homomorphic encryption scheme allows for much faster exponentiation. Superficially, our work bears similarity with the work of Kamara and Raykova [13] that protect polynomial evaluation by randomized reduction techniques. With  $q$  being the degree of a polynomial, the user splits each data record into  $2 \cdot q + 1$  shares, each of size  $2 \cdot q + 1$ . Shares are then uploaded and evaluated in parallel, and results are aggregated. However, storage expansion, even for modest values of  $q$ , the approach quickly becomes impractical. Also, for different polynomials, the user would need to upload the data multiple times.

Searching on encrypted data has received a lot of attention recently, cf. seminal papers [4, 19]. While closely related, it is far from straightforward to adopt these schemes to perform efficient counting in a highly parallel cloud computing, e.g.,

MapReduce environment. Also notice that, e.g., Boneh et al. [4] rely on the computation of very expensive bilinear pairings for each element of a data set, rendering this approach impractical in a cloud setting. Much research has been done to compute statistics in a privacy-preserving manner using *differential privacy*, see the seminal paper by Dwork [7]. Contrary to the threat model considered in this paper, the adversary in differential privacy research is not the cloud infrastructure, but a curious user querying statistics to learn information about individual entries in a data set. EPiC addresses the opposite problem, where a user does not trust the cloud infrastructure.

## 6 Conclusion

In this paper, we present EPiC to address a fundamental problem of statistics computation on outsourced data: privacy-preserving pattern counting. EPiC's main idea is to count occurrences of patterns in outsourced data through a privacy-preserving summation of the pattern's indicator-polynomial evaluations over the encrypted dataset records. Using a "somewhat homomorphic" encryption mechanism, the cloud neither learns any information about outsourced data nor about the queries performed. Our implementation and evaluation results for MapReduce running on Amazon's cloud with up to 1 TByte of data show only modest overhead compared to non-privacy-preserving counting. This makes EPiC practical in a real-world cloud computing setting today.

**Acknowledgements** This work was partially supported by NSF Grant 1218197.

## References

1. Amazon Elastic MapReduce. <http://aws.amazon.com/elasticmapreduce/>
2. Apache Hadoop (2010) <http://hadoop.apache.org/>
3. Babai L, Fortnow L (1991) Arithmetization: a new method in structural complexity theory. In: Computational complexity, pp 41–66, ISSN: 1016-3328
4. Boneh D, DiCrescenzo G, Ostrovsky R, Persiano G (2004) Public key encryption with keyword search. In: Proceedings of Eurocrypt, pp 506–522. Barcelona, Spain
5. Brakerski Z, Vaikuntanathan V (2011) Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: Rogaway P (eds) Advances in cryptology—CRYPTO 2011, pp 505–524. Springer, Berlin. ISBN: 978-3-642-22792-9
6. Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: Proceedings of symposium on operating system design and implementation, pp 137–150. San Francisco, USA
7. Dwork C (2006) Differential privacy. In: Proceedings of colloquium automata, languages and programming, pp 1–12, Venice, Italy, 2006. ISBN: 3-540-35907-9
8. Gentry C (2009) Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM symposium on theory of computing, STOC '09, pp 169–178. New York, NY, 2009. ACM. ISBN: 978-1-60558-506-2
9. Gentry C, Halevi S (2011) Implementing Gentry's fully-homomorphic encryption scheme. In: Advances in cryptology—EUROCRYPT 2011, pp 129–148. Tallinn, Estonia, 2011. Springer, Berlin. ISBN: 78-3-642-20464-7
10. Google (2010) A new approach to China. <http://googleblog.blogspot.com/2010/01/new-approach-to-china.html>
11. Hadoop. Powered by Hadoop, list of applications using Hadoop MapReduce (2011). <http://wiki.apache.org/hadoop/PoweredBy>

12. Hoffstein J, Pipher J, Silverman JH (1998) NTRU: a ring-based public key cryptosystem. In: *Algorithmic number theory, third international symposium, ANTS-III, Portland, Oregon, USA, June 21–25, 1998, proceedings*, pp. 267–288
13. Kamara S, Raykova M (2013) Parallel homomorphic encryption. In: Adams AA, Brenner M, Smith M (eds) *Financial cryptography and data security*, pp 213–225. Springer, Berlin. ISBN: 978-3-642-41320-9
14. Katz J, Lindell Y (2007) *Introduction to modern cryptography* (Chapman & Hall/CRC Cryptography and network security series). Chapman & Hall/CRC
15. Lauter K, Naehrig N, Vaikuntanathan V (2011) Can homomorphic encryption be practical? In: *Proceedings of ACM workshop on cloud computing security*. Chicago
16. Lepoint T, Tibouchi M (2015) Cryptanalysis of a (somewhat) additively homomorphic encryption scheme used in PIR. In: *Financial cryptography and data security—FC 2015 international workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, pp 184–193
17. Rocha F, Correia M (2011) Lucy in the sky without diamonds: stealing confidential data in the cloud. In: *Proceedings of international workshop on dependability of clouds, data centers and virtual computing y environments*, pp 129–134, Hong Kong, China
18. Shamir A (1992)  $IP = PSPACE$ . *J ACM* 39(4):869–877. ISSN: 0004-5411
19. Song D, Wagner D, Perrig A (2000) Practical techniques for searches on encrypted data. In: *Proceedings of symposium on security and privacy*, pp 44–55, Berkeley, USA
20. Techcrunch. Google confirms that it fired engineer for breaking internal privacy policies (2010). <http://techcrunch.com/2010/09/14/google-engineer-spying-fired/>
21. The Telegraph (2011) Patient records go online in data cloud. <http://bit.ly/1yb28bw>
22. Trostle J, Parrish A (2010) Efficient computationally private information retrieval from anonymity or trapdoor groups. In: *Proceedings of conference on information security*, pp 114–128, Boca Raton, USA, 2010. ISBN: 978-3-642-18177-1
23. Vaikuntanathan V (2011) Computing blindfolded: new developments in fully homomorphic encryption. In: *Proceedings of the 2011 IEEE 52nd annual symposium on foundations of computer science, FOCS'11*, pp 5–16, Washington, DC, USA 2011. ISBN: 978-0-7695-4571-4
24. van Dijk M, Gentry C, Halevi S, Vaikuntanathan V (2010) Fully homomorphic encryption over the integers. In: *Proceedings of international conference on theory and applications of cryptographic techniques, EUROCRYPT'10*, pp 24–43, Monaco, 2010. Springer, Berlin. ISBN: 3-642-13189-1
25. Vo-Huu TD, Blass E-O, Noubir G EPiC Source Code. <http://www.ccs.neu.edu/home/noubir/projects/epic>
26. Whittaker Z (2011) Microsoft admits Patriot Act can access EU-based cloud data. Zdnet <http://www.zdnet.com/>