# Private and Secure Secret Shared MapReduce

by

**Shlomi Dolev, Yin Li, and Shantanu Sharma**

Technical Report #16-01

May 2016

The Lynne and William Frankel Center for Computer Science Department of Computer Science,
Ben-Gurion University, Beer Sheva, Israel.

# Private and Secure Secret Shared MapReduce

Shlomi Dolev, Yin Li, Shantanu Sharma

**Abstract**—Data outsourcing allows data owners to keep their data in public clouds. However, public clouds do not ensure the privacy of data and computations. One fundamental and useful framework for processing data in a distributed fashion is MapReduce. In this paper, we investigate and present techniques for executing MapReduce computations in the public cloud while preserving privacy. Specifically, we propose a technique to outsource a database using Shamir secret-sharing scheme to the public clouds, and then, provide privacy-preserving algorithms for performing search and fetch, equijoin, and range queries using MapReduce. Consequently, in our proposed algorithms, the public cloud cannot learn the database or the computations. All the proposed algorithms eliminate the role of the database owner, which only creates and distributes secret-shares once, and minimize the role of the user, which only needs to perform a simple operation for reconstructing the result, for query processing. We evaluate the efficiency of all the algorithms by (*i*) the number of communication rounds (between a user and a cloud), (*ii*) the total amount of bit flow (between a user and a cloud), and (*iii*) the computational load at the user-side and the cloud-side.

**Index Terms**—Computation and data privacy, data and computation outsourcing, distributed computing, MapReduce, Shamir's secret-sharing.

## 1 Introduction

Data and computation outsourcing move databases and computations from private and trusted computers (or clouds) to a public cloud, which is not under the control of a single user. Thus, the outsourcing results in less burden on a private cloud in terms of the maintenance of databases, infrastructures, and queries' executions. Unfortunately, the ease in storing data and executing computations in the public clouds implies a risk of violating security and privacy of the databases and the computations.

MapReduce [1] was introduced by Google in 2004. Details about MapReduce can be found in Chapter 2 of [2]. MapReduce provides efficient and fault tolerant parallel processing of large-scale data without dealing with security and privacy of data and computations. While MapReduce is not directly related to the public clouds, many public clouds, *e.g.*, Amazon Elastic MapReduce, Google App Engine, IBM Blue Cloud, and Microsoft Azure, enable users to perform MapReduce cloud computations without considering physical infrastructures and software installation. Thus, the deployment of MapReduce on the public clouds enables users to process large-scale data in a cost-effective manner and establishes a relationship between the two independent entities, *i.e.*, the public clouds and MapReduce.

### 1.1 Motivating Examples

We present two examples (search and equijoin) to show the need for security and privacy of data and query execution using MapReduce in the public cloud.

*Shlomi Dolev is with Ben-Gurion University of the Negev, Beer-Sheva, Israel (e-mail: dolev@cs.bgu.ac.il).*
*Yin Li is with Xinyang Normal University, China. (e-mail: yunfeiyangli@gmail.com).*
*Shantanu Sharma is with Ben-Gurion University of the Negev, Beer-Sheva, Israel (e-mail: sharmas@cs.bgu.ac.il).*
*We thank Jeffrey Ullman for valuable comments. An extended abstract of this work is accepted in the Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec), 2016. This work is supported by the Rita Altura Trust Chair in Computer Sciences, Lynne and William Frankel Center for Computer Sciences, Israel Science Foundation (grant 428/11), the Israeli Internet Association, and the Ministry of Science and Technology, Infrastructure Research in the Field of Advanced Computing and Cyber Security.*

**Secure and privacy-preserving search.** *Problem statement*: Consider a hospital database that can have different users, *e.g.*, doctors, nurses, insurance companies, and database administrators. As there are different users that may search in the database, on one hand, it is required that only an authenticated and authorized user will find the desired result. On the other hand, maintaining a database in the hospital is not a trivial and cheap task. Hence, it is beneficial to outsource the database to the public clouds.

The public clouds, however, do not ensure the privacy of data and computations; any user or the cloud can breach the privacy of data and computations. Therefore, it is necessary to keep a database in the cloud in a privacy-preserving manner so that only authenticated and authorized users can access and know the database.

**Secure and privacy-preserving equijoin of two relations $X(A,B)$ and $Y(B,C)$.** *Problem statement*: The join of relations $X(A,B)$ and $Y(B,C)$, where the joining attribute is $B$, provides output tuples $\langle a,b,c\rangle$, where $(a,b)$ is in $X$ and $(b,c)$ is in $Y$. In the equijoin of $X(A,B)$ and $Y(B,C)$, all tuples of both the relations with an identical value of the attribute $B$ should appear together for providing the final output tuples.

Consider that the relations $X$ and $Y$ belong to two organizations, *e.g.*, a company and a hospital, while a third user wants to perform the equijoin. However, both the two organizations want to provide results while maintaining the privacy of their databases, *i.e.*, without revealing the whole database to the other organization or the user. Hence, it is required to perform the equijoin in a secure and privacy-preserving manner. *Join using MapReduce*: MapReduce provides an easy way for performing join operations on large-scale databases without ensuring security and privacy of data and computations. In the context of MapReduce, a mapper (*i.e.*, an application of a map function to a single input) takes a single tuple from $X(A,B)$ or $Y(B,C)$ and provides $\langle B, X(A)\rangle$ or $\langle B, Y(C)\rangle$ as key-value pairs. A reducer (*i.e.*, an application of a reduce function to a single $key$ and its associated list of $values$) joins the assigned tuples who have an identical key.

### 1.2 Formal Problem Statement

The main obstacle for providing privacy-preserving framework for MapReduce in the adversarial (public) clouds is computational and storage efficiency. An adversarial cloud may breach the privacy of data and computations. Hence, we are interested in making a secure and privacy-preserving computation execution and storage-efficient technique for MapReduce computations in the clouds. We are looking at information-theoretically secure data and computation outsourcing and query execution using MapReduce. Specifically, our focus is on four types of privacy-preserving queries, as follows: `count`, `search` and `fetch`, `equijoin`, and fetch tuples with a value belonging in a `range`. By developing privacy-preserving data and computation outsourcing techniques, a user receives only the desired result without knowing the whole database; moreover, the clouds are also unable to know the database or the query.

### 1.3 Our Contributions

In this paper, we provide the following:

**Information-theoretically secure data outsourcing.** We provide an information-theoretically secure data and computation outsourcing technique that prevents a malicious cloud provider to know the database or the query. Specifically, we use Shamir secret-sharing (SSS) [3] for making secret-shares of each tuple of a relation and send them to the clouds. A user can execute queries using accumulating-automata (AA) [4] on these secret-shares without revealing queries/data to the cloud.

**Privacy-preserving query execution by third-parties.** We can perform the following operations in a privacy-preserving manner, as: `count`, `equijoin`, `search`, and `fetch`. The main idea is that if we can perform privacy-preserving string matching operations on a database, then using the string matching operations we can perform all the above mentioned operations in a privacy-preserving manner.

*`Count` operation.* `Count` operation provides the number of occurrences of a pattern, and in the proposed technique, this operation requires the minimal workload at the user-side; see Section 5.1.

*`Search` and `fetch` operations.* We present two algorithms for searching and fetching all the tuples containing a pattern; see Section 5.2. All these algorithms result in different amounts of workload at the user-side, and we find a tradeoff between the workload at the user-side and the number of communication rounds between a user and a cloud; see Section 5.2.

`Equijoin`. We provide a privacy-preserving approach for performing the equijoin, where the whole job is executed on secret-shares in the cloud and the user only performs an interpolation that provides the desired result.

`Range` query. A range query provides all the tuples where a value belongs in a range. We provide a privacy-preserving range query based on privacy-preserving count, search, and fetch operations; see Section 5.4.

**Advantages of the proposed approach.** The proposed approach has the following main advantages: (*i*) the approach is well suited to the (public) cloud environment; (*ii*) eliminates the need of a database owner in terms of the database maintenance and query processing, except creating and distributing secret-shares to the clouds; (*iii*) the query response time is also smaller than the response time of a query over an encrypted database; and (*iv*) the proposed technique can also be applied for outsourcing a distributed database and perform non-MapReduce computations in a privacy-preserving manner.

**Analysis of the algorithms.** We analyze our algorithm on four parameters, as: the total amount of bits flow between a user and a cloud, the number of interaction rounds between a user and a cloud, and the computational workload on the user-side and the cloud-side.

### 1.4 Related Work

MapReduce was introduced by Dean and Ghemawat in 2004 [1]. PRISM [5], PIRMAP [6], EPiC [7], MrCrypt [8], and Crypsis [9] provide privacy-preserving MapReduce execution in the cloud on encrypted data. However, all these protocols increase computation time due to dependency on encryption and decryption of data, and provide a limited operations (as a tradeoff between preserving data privacy and utilization). Details of security and privacy concerns in MapReduce may be found in [10].

The authors [11] provide a privacy-preserving join operation using secret-sharing. However, the approach [11] requires that two different data owners share some information for constructing *an identical share for identical values* in their relations. However, sharing information among data owners is not trivial when they are governed by different organizations, and moreover, by following this approach, a malicious data owner may be able to obtain the database.

The authors [12] provide a technique for data outsourcing using a variation of SSS. However, the approach [12] suffers from two major disadvantages, as follows: (*i*) in order to produce an answer to a query, the data owner has to work on all the shares, hence, the data owner not the cloud performs a lot of work; and (*ii*) a third party cannot directly issue any query on secret-shares, and it has to contact with the data owner. In [12], the authors provide a way for constructing polynomials that can maintain the orders of the secrets. However, this kind of polynomial is based on an integer ring (no modular reduction) rather than a finite field; thus, it has potential security risk.

There are some other works [13], [14], [15] that provide searching operations on secret-shares. In [13], a data owner builds a Merkle hash tree [16] according to a query. In [14], a user knows the addresses of the desired tuples, so they can fetch all those tuples obliviously from the clouds without performing a search operation in the cloud. Similar ideas can also be found in [15].

To the best of our knowledge, there is no algorithm that (*i*) eliminates the need of a database owner except one time creation and distribution of secret-shares, (*ii*) minimizes the overhead at the user-side, and (*ii*) provides information-theoretically secure MapReduce computations in the cloud. In this paper, we build a technique for data and computation outsourcing based on SSS and accumulating-automata [4]. Essentially, our MapReduce-based `count` operation (Section 5.1) adapts the basic working of AA; hence, we provide the basic working of AA in that section. The remaining advanced operations of AA are detailed in [4].

By using AA, our algorithms can perform a string matching operation on secret-shares in the cloud, without downloading the whole database of the form of secret-shares. However, most of the existing secret-sharing based privacy-preserving algorithms are unable to do string matching operations in the cloud; see Table 1.

The proposed technique overcomes all the disadvantages of the existing encryption-based techniques [5], [6], [7], [8], [9] and

secret-sharing based data outsourcing techniques [11], [12], [13], [14], [15]. Thus, there is no need for (*i*) sharing information among different data owners, (*ii*) working at the database owners, except creation and distribution of secret-shares, (*iii*) having an identical share for multiple occurrences of a value, and (*iv*) a third party can directly execute queries in the clouds without revealing queries to the clouds.

## 2 System and Adversarial Settings

We consider, for the first time, data and MapReduce-based computation outsourcing of the form of secret-shares to $c$ *non-communicating* clouds. The meaning of non-communicating clouds is that they do not exchange data with each other, only exchange data with the user or the database owner.

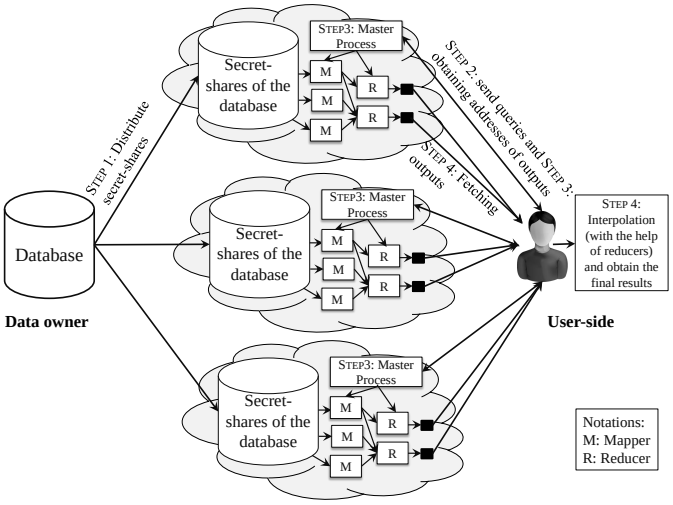


Fig. 1: The system architecture.

### 2.1 The System Architecture

The architecture is simple but powerful and assumes the following:
Step 1. A data owner outsources databases of the form of secret-shares to $c$ (non-communicating) clouds only once; see Step 1 in Figure 1. We use $c$ clouds to provide privacy-preserving computations using SSS. Note that a single *non-trustworthy* cloud cannot provide privacy-preserving computations using secret-sharing.
Step 2. A preliminary step is carried out at the user-side who wants to perform a MapReduce computation. The user sends a query of the form of secret-shares to all $c$ clouds to find the desired result of the form of secret-shares; see Step 2 in Figure 1. The query must be sent to at least $c' < c$ number of clouds, where $c'$ is the threshold of SSS.
Step 3. The clouds deploy a *master process* that executes the computation by assigning the *map tasks* and the *reduce tasks*; see Step 3 in Figure 1. The user interacts only with the master process in the cloud, and the master process provides the addresses of the outputs to the user. It must be noted that the communication between the user and the clouds is presumed to be the same as the communication between the user and the master process.
Step 4. The user fetches the outputs from the clouds and performs a simple operation (especially, polynomial interpolation using Lagrange polynomials [17]) (with the help of reducers) for obtaining the secret-values; see Step 4 in Figure 1.

In this system setting, users wish to execute their MapReduce computations without revealing the computation to the clouds, while the database owner wishes to store its database and perform queries' execution in public clouds without compromising the privacy.

**Note.** Physical machines of a single cloud provider can be compromised as well, possibly leaking information (through the network) they received when participating in the MapReduce; thus, secret sharing will make the leaked information meaningless, as long as the number of leaked machines is less than the threshold or the compromised machines are controlled by different (non-collaborating) adversaries.

### 2.2 Adversarial Settings

We assume, on one hand, that an adversary cannot launch any attack against the data owner, who is trustworthy. Also, the adversary cannot access the secret-sharing algorithm and machines at the database owner side.

On the other hand, an adversary can access public clouds and data stored therein. Hence, the adversary can also access input, intermediate, and output data of a MapReduce job. A user who wants to perform a computation on the data stored in public clouds may also behave as an adversary. Moreover, the cloud itself can behave as an adversary, since it has complete privileges to all the machines and storage. Both the user and the cloud can launch any attack to compromise the privacy of data or computations.

We consider an honest-but-curious adversary, which is considered in the standard settings for security in the public cloud [18], [19], [20]. The honest-but curious adversary performs assigned computations correctly, but tries to breach the privacy of data or MapReduce computations, by analyzing data, computations, or data flow. However, such an adversary does not modify or delete information from the data.

We assume that an adversary can know less than $c' < c$ clouds locations that store databases and execute queries. Recall that $c'$ is the threshold of SSS. In addition, the adversary cannot eavesdrop all the $c'$ or $c$ channels (between the database owner and the clouds, and between the user and the clouds). Hence, we do not impose private communication channels.

Under such an adversarial setting, we provide a guaranteed solution so that an adversary cannot learn the data or computations. It is important to mention that an adversary can break our protocols by colluding $c'$ clouds, which is the threshold for which the secret sharing scheme is designed for.

### 2.3 Parameters for Analysis

We analyze our privacy-preserving algorithms on the following parameters:
*Communication cost*: is the sum of all the bits that are required to transfer between a user and a cloud.
*Computational cost*: is the sum of all the bits over which a cloud or a user works.
*Number of rounds*: shows how many times a user communicates with a cloud for obtaining the results.

Table 1 summarizes all the results of this paper and comparison with the existing algorithms, based on the five criteria, as: (*i*) communication cost, (*ii*) computational cost at the user and the cloud, (*iii*) number of rounds, (*iv*) matching of a pattern online or offline, and (*v*) dependence of secret-sharing. Note that in

TABLE 1: Comparison of different algorithms with our algorithms.

| Algorithms | Communication cost | Computational cost | | # rounds | Matching | Based on |
|---|---|---|---|---|---|---|
| | | User | Cloud | | | |
| **`Count` operation** | | | | | | |
| EPiC [7] | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | 1 | Online | E |
| **Our solution** 5.1 | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ | $nw$ | 1 | Online | SSS |
| **`Search` and single tuple `fetch` operation** | | | | | | |
| Chor et al. [15] | $\mathcal{O}(nmw)$ | $\mathcal{O}(1)$ | $\mathcal{O}(nmw)$ | $log_2 n$ | Online | SSS |
| PRISM [5] | $\mathcal{O}((nm)^{\frac{1}{2}}w)$ | $\mathcal{O}((nm)^{\frac{1}{2}}w)$ | $\mathcal{O}(nmw)$ | $q$ | | E |
| **Our solution** 5.2.1 | $\mathcal{O}(mw)$ | $\mathcal{O}(mw)$ | $\mathcal{O}(mw)$ | 1 | Online | SSS |
| **`Search` and multi-tuples `fetch` operation** | | | | | | |
| rPIR [14] | $\mathcal{O}(nm)$ | $\mathcal{O}(1)$ | $\mathcal{O}(nmw)$ | 1 | No | SSS |
| PIRMAP [6] | $\mathcal{O}(nmw)$ | $\mathcal{O}(mw)$ | $\mathcal{O}(nmw)$ | 1 | No | E |
| Goldberg [13] | $\mathcal{O}(n+m)$ | $\mathcal{O}(m)$ | $\mathcal{O}(nm)$ | 2 | Offline | SSS |
| Emekci et al. [12] | $\mathcal{O}(\ell m)$ | $\mathcal{O}(\ell m)$ | $\mathcal{O}(n)$ | 2 | Offline | vSS |
| **Our solution**: knowing addresses 5.2.2 | $\mathcal{O}\big((log_\ell n + log_2\ell)\ell\big)$ | $\mathcal{O}\big((log_\ell n + log_2\ell)\ell\big)$ | $\mathcal{O}\big((log_\ell n + log_2\ell)\ell nw\big)$ | $\lfloor log_\ell n \rfloor + \lfloor log_2\ell \rfloor + 1$ | Online | SSS |
| **Our solution**: fetching tuples 5.2.2 | $\mathcal{O}((n+m)\ell w)$ | $\mathcal{O}((n+\ell m)w)$ | $\mathcal{O}(\ell nmw)$ | 1 | Online | SSS |
| **Equijoin** | | | | | | |
| **Our solution** 5.3 | $2nwk + 2k\ell^2 mw$ | $2nw + 2k\ell^2 mw$ | $2\ell^2 kmw$ | $2k$ | Online | SSS |

**Notations:** Online: perform string matching in the cloud. Offline: perform string matching at the user-side. E: encryption-decryption based. SSS: Shamir's Secret-sharing. vSS: a variant of SSS. $n$: # tuples, $m$: # attributes, $\ell$: # occurrences of a pattern ($\ell \leq n$), $w$: bit-length of a pattern.

TABLE 2: Notations used in the paper.

| Notations | Meaning |
|---|---|
| $c$ | Number of non-communicating clouds holding search-shares |
| $c'$ | Threshold of Shamir's secret-sharing |
| $\mathcal{R}$ | A relation (or table) |
| $n$ | Number of tuples (or row) in $\mathcal{R}$ |
| $m$ | Number of attributes (or columns) in $\mathcal{R}$ |
| $p$ | A searching pattern |
| $m'$ | An attribute of a relation on which we search $p$ |
| $\ell$ | The number of occurrences of $p$ |
| $w$ | Maximum bit-length |

offline string matching operations, the user needs to download the whole database and then search the pattern. In this way the number of rounds are decreased while the communication and computation cost increase. On the other hand, the meaning of online operations is that the cloud performs the desired operation without sending the whole database to the user. Table 2 shows notations used in this paper.

## 3 Overview of the Proposed Approach

The proposed approach consists of the following four steps, as:

1) Creating secret-shares of a database/relation at the database owner side and then distributing secret-shares to $c$ non-communicating clouds.
2) Sending queries (we consider only four queries, given below) of the form of secret-shares from the user-side to the $c$ clouds.
3) A cloud deploys a MapReduce job, where a mapper reads each word of the relation/an assigned split one-by-one and performs string matching operations over the secret-shares.
4) The user fetches the results of the form of secret-shares from the clouds and interpolates them for obtaining the final output.

In the step (2), the user sends a pattern, $p$, of the form of secret-shares and the length, say $x$, of $p$. In the step (3), a mapper creates an automaton of $x+1$ node[1] for preforming string matching operations. The final value of the last node shows the result of the string matching operation, *i.e.*, 0 or 1 of the form of secret-shares.

We focus on four types of queries, as follows:
`Count`: Here, step (3) is carried out over the whole relation for obtaining the output of the `count` operation.
`Search` and `fetch`: queries are based on the `count` query. We first count the occurrence of $p$ in the relation and then fetch

1. Note that $x+1$ nodes are not machine nodes. These are parts of an automaton.

the desired tuples in a single step, if the occurrence of $p$ is one. Otherwise, we partition the relation according to a tree-based algorithm (given in Section 5.2.2) and then fetch all the tuples.

`Range` query: Here, we check each number, which is of the form of secret-share, whether it is in the range or not, followed by a similar method to the `search` and `fetch` queries.

`Equijon`: Here, a user performs a heavy workload as compared to the previous three queries. The user fetches all the values of a joining attribute and finds identical values in all the relations. Nevertheless, it is less expensive as compared to fetching the whole relations as compared to [12]. After knowing identical joining values, the user performs a similar method to the `search` and `fetch` queries for each value, and the cloud provides equijoin (or cross-product) of identical joining values.

**Aside.** It is challenging in creating secret-shares of a database; but once we did it, the rest of operations are relatively easier. Moreover, creating secret-shares of a database and its storage is less expensive than encrypting a database and its storage [21]. Also, it should be noted that standard techniques based on Berlekamp-Welch algorithm [22], where additional secret shares are used to encode the data can be directly applied here, enabling us to cope with a malicious adversary, with no change in the communication pattern.

## 4 Creation and Distribution of Secret-Shares of a Relation

We consider an example of a relation, *Employee*, see Figure 2. A data owner creates secret-shares of this relation and sends to $c$ clouds. In this section, we show how to create secret-shares of a value, following an approach given in [4]. We first provide a simplified and insecure algorithm, and then, a secure algorithm for creating secret-shares based on SSS.

| Employee Id | First name | Last name | Date of birth | Salary | Department |
|---|---|---|---|---|---|
| E101 | Adam | Smith | 12/07/1975 | 1000 | Sale |
| E102 | John | Boro | 10/30/1985 | 2000 | Design |
| E103 | Eve | Smith | 05/07/1985 | 500 | Sale |
| E104 | John | Williams | 04/04/1990 | 5000 | Sale |

Fig. 2: A relation: *Employee*.

**A simple and insecure way for creating *encoded data (non-secret-shares)*.** Assume that a database only contains English words. Since the English alphabet consists of 26 letters, each letter can be represented by a unary vector with 26 bits. Hence, the letter 'A' is represented as $(1_1, 0_2, 0_3, \ldots, 0_{26})$, where the subscript represents the position of the letter; since 'A' is the first letter, the first value in the vector is one and others are zero. Similarly, 'B' is $(0_1, 1_2, 0_3, \ldots, 0_{26})$, 'J' is $(0_1, \ldots, 0_9, 1_{10}, 0_{11}, \ldots, 0_{26})$, and so on. Now, the database owner sends these vectors to a cloud. Note that in this case, the cloud can easily deduce words.

The reason of using unary representation here is that it is very easy for verifying two identical letters. The expression $S = \sum_{i=0}^{r} u_i \times v_i$, compares two letters, where $(u_0, u_1, \cdots u_r)$ and $(v_0, v_1, \cdots, v_r)$ are two unary representations. It is clear that whenever any two letters are identical, $S$ is equal to one; otherwise, $S$ is equal to zero. Binary representation can also be accepted, but the comparison function is different from that used in the unary representation [23].

**A secure way for creating *secret-shares*.** When outsourcing a vector to the clouds, we use SSS and make secret-shares of every bit by selecting different polynomials of an identical degree; see Algorithm 1. For example, we create secret-shares of the vector of 'A' $((1_1, 0_2, 0_3, \ldots, 0_{26}))$ by using 26 polynomials of an identical degree to create secret-shares of each bit, since the length of the vector is 26. Following that, we can create secret-shares for all the other letters and distribute them to different clouds.

**Algorithm 1:** Algorithm for creating secret-shares

**Inputs:** $\mathcal{R}$: a relation having $n$ tuples and $m$ attributes, $c$: the number of non-communicating clouds
**Variables:** $letter$: represents a letter

```
Function create_secret-shares(R) begin
    for (i, j) ∈ (n, m) do
        foreach letter[i, j] do Make_shares(letter[i, j])
Function Make_shares(letter[i, j]) begin
    x ← length of letter[i, j]
    Create x unary-vectors, where the position of the letter
      has value 1 and all the other values are 0
    Use x polynomials of an identical degree for creating
      secret-shares of 0 and 1
    Send secret-shares to c clouds
```

Since we use SSS, a cloud cannot infer a secret. Moreover, it is important to emphasize that we use *different* polynomials for creating secret-shares of each letter; thereby multiple occurrences of a word in a database have different secret-shares. Therefore, a cloud is also unable to know the total number of occurrences of a word in the whole database. Following that, the two occurrences of the word `John` in our example (see Figure 2) have two different secret-shares.

*Secret-shares of numeral values*. We follow the similar approach for creating secret-shares of numeral values as used for alphabets. In particular, we create a unary vector of length 10 and put all the values 0 except only 1 according to the position of a number. For example, '1' becomes $(1_1, 0_2, \ldots, 0_{10})$, '0' becomes $(0_1, 0_2, \ldots, 1_{10})$, and so on. After that, use SSS to make secret-shares of every bit in each vector by selecting different polynomials of an identical degree for each number, and send them to multiple clouds.

## 5 Privacy-Preserving Query Processing on Secret-Shares using MapReduce in the Clouds

In this section, we will present four privacy-preserving algorithms for performing four fundamental operations on a database of the form of secret-share, as: count the occurrences of a pattern, fetch all the tuples containing a pattern, equijoin of two relations, and execution of range queries. All these algorithms are based on string matching of a value of a relation with a pattern, where the value and the pattern are of the form of secret-shares. The string matching operation on secret-shares is done using AA [4]. All these algorithms execute operations obliviously in the cloud so that the cloud can never know which operations are executing on which tuples of a relation, while the user has to perform a simple operation to reconstruct the result. Throughout this section, we denote a pattern by $p$.

### 5.1 Count Query

We present a privacy-preserving algorithm for counting the number of occurrences of $p$ in the cloud; see Algorithm 2. We use our running example to count the number of people who have their first name as John in the relation *Employee*, see Figure 2. The algorithm is divided into two phases, as:
PHASE 1: Privacy-preserving counting in the clouds, Section 5.1.1
PHASE 2: Result reconstruction at the user-side, Section 5.1.2

In short, we apply a string matching algorithm, which is done using AA that compares each value of a relation with $p$. If a value and $p$ match, it will result in 1; otherwise, we have 0. We apply the same algorithm on each value and collect the outputs. The sum of all the outputs provide the number of occurrences of $p$. Note that all the values of a relation, a pattern, and the result, *i.e.*, 0 or 1, are of the form of secret-share.

#### 5.1.1 Counting a pattern

For the purpose of simplicity and understanding, we first show how to perform a string matching operation on the encoded database (non-secret-shares), which we created in Section 4. Then, based on the string matching operation we show how to count the occurrences of a pattern, John, in a relation of the form of secret-shares.

**Counting John in encoded data (non-secret-shares).**

*Working at the user-side*. A user creates unary vectors for each letter of $p$ (as in Section 4) and sends them to the clouds. For example, a user, who wants to search John, creates and sends four unary vectors, corresponding to 'J,' 'o,' 'h,' and 'n' to the clouds, where a MapReduce computation counts the occurrences of John.

*Working in the cloud*. Now, a cloud has three things, as: (*i*) a relation, *Employee* of the form of encoded data, (*ii*) a searching pattern, John, of the form of encoded data, and (*iii*) a code of mappers.

The mapper creates an automaton, which performs a string matching operation, with $x+1$ nodes in an automation, where $x$ is the length of $p$ and initializes values of these nodes. The first node is assigned a value one ($N_1 = 1$, $N_i$ shows the value of node $i$), and all the other nodes are assigned values zero ($N_i = 0, i \neq 1$). The mapper reads each encoded word one-by-one and executes $x + 1$ steps for each word for finding new values of the nodes. At the end of the computation, the value of the node $N_{x+1}$ shows the number of occurrences of $p$ in a relation.

***Example*.** In our running example, since we are searching a pattern of length four, a mapper creates an automaton of five nodes, assigns a node value one to the first node, and zero to the other nodes. The mapper reads the first name of employees one-by-one and executes five steps, given in Table 3 for each word.

***Explanation of the steps for counting John.*** In the first iteration $i = 1$, the mapper reads the word 'Adam,' executes STEPs 1 and 2, and obtains the value of $v_1$ by multiplying the vector of 'A' with the vector of 'J,' which results in $v_1 = 0$ and $N_2^{(1)} = 0$. After that, the mapper executes STEP 3 and obtains the value of $v_2$ by multiplying the vector of 'd' with the vector of 'o,' which results in $v_2 = 0$, and hence, using the value of $N_2^{(1)} = 0$, $N_3^{(1)}$ will be 0. Then, the mapper executes STEPs 4 and 5 and obtains values of $v_3$ and $v_4$ by multiplying the vector of 'a' with the vector of 'h,' which results in $v_3 = 0$ and $N_4^{(1)} = 0$, and respectively, by multiplying the vector of 'm' with the vector of 'n,' which results in $v_4 = 0$ and $N_5^{(1)} = 0$. The value of $N_5^{(1)} = 0$ in the first iteration shows that 'Adam' and John are not identical words.

| |
|---|
| STEP 1: $N_1 = 1$, $N_5^0 = 0$ |
| STEP 2: $N_2^{(i)} = N_1 \times v_1$ |
| STEP 3: $N_3^{(i)} = N_2^{(i)} \times v_2$ |
| STEP 4: $N_4^{(i)} = N_3^{(i)} \times v_3$ |
| STEP 5: $N_5^{(i)} = N_5^{(i-1)} + N_4^{(i)} \times v_4$ |
| The notation $N_j^{(i)}$ shows that the node $j$ is executing a step in iteration $i$. |
| The final value of the node $N_5$, which is sent to the user, is the number of occurrences of the pattern. |
| In our example, there are four tuples so that these five steps will be executed exactly four times. |

TABLE 3: The steps executed by a mapper for counting John.

Next, the mapper reads the word 'John' and executes the second iteration, $i = 2$. In STEP 2, multiplication of the vector of 'J' with the vector of 'J' results in $v_1 = 1$ and $N_2^{(2)} = 1$. Similarly, the mapper executes STEPS 3, 4, and 5, and obtains, the values as: $v_2 = 1$ and $N_3^{(2)} = 1$, $v_3 = 1$ and $N_4^{(2)} = 1$, and $v_4 = 1$ and $N_5^{(2)} = 1$. Next, the mapper reads the word 'Eve,' executes all the STEPs, and results in $v_1 = 0$ and $N_2^{(3)} = 0$, $v_2 = 0$ and $N_3^{(3)} = 0$, $v_3 = 0$ and $N_4^{(3)} = 0$, $v_4 = 0$, and the value of $N_5^{(3)}$ will be 1, which shows that until now only one employee has John as a first name. The mapper reads the word 'John,' executes all the STEPs and eventually results in $N_5^{(4)} = 2$, which shows that two employees have John as their first names.

**Counting a pattern, John, in secret-shares in different clouds.** Now, we explain how to count the occurrences of John in a relation of the form of secret-shares; see Algorithm 2. We use a similar approach presented above to do so.

*Working at the user-side*. Recall that the user creates unary vectors for each letter of $p$. In order to hide the vectors of $p$, the user creates secret-shares of each vector of $p$, as suggested in Section 4, sends them to the clouds. In our running example, a user creates four unary vectors for each letter of John, and then, creates secret-shares of each unary vector. In addition, the user writes a code of mappers for each cloud and also creates node values of the form of secret-shares.

*Working in the cloud*. Now, a cloud has three things, as: (*i*) a relation of the form secret-shares, (*ii*) a searching pattern of the form of secret-shares, and (*iii*) code of mappers with node values of the form of secret-shares.

In order to count the number of occurrences of $p$, the mapper performs five steps, as mentioned above, for comparing John with each first name. At this time, the mapper is unable to know the value of the node $N_5$ in each iteration and sends the final value of $N_5$ to the user of form of a $\langle key, value \rangle$ pair, where a $key$ is an identity of an input split over which the operation has performed, and the corresponding $value$ is the final value of the node $N_5$ of the form of secret-shares. The user collects $\langle key, value \rangle$ pairs from all the clouds or a sufficient number of clouds such that the secret can be generated using those shares.

**Algorithm 2:** Algorithm for privacy-preserving `count` operation in the clouds using MapReduce

**Inputs:** $R$: a relation of the form of secret-shares having $n$ tuples and $m$ attributes, $p$: a searching pattern, $c$: the number of clouds, $N_j^{(i)}$: defined in Table 3
**Output:** $\ell$: the number of occurrences of $p$
**Interfaces:** $length(p)$: finds length of $p$
$attribute(p)$: which attribute of the relation has to be searched for $p$
**Variables:** $int\_result_i$: the output at $i^{th}$ cloud after executing the map function
$SS_k[i,j]$: shows a letter of the form of secret-share at $i^{th}$ position of $k^{th}$ string in $j^{th}$ attribute
$result[]$: at the user-side to store outputs of all the clouds

**User-side:**
1 Compute secret-shares of $p$: $p' \leftarrow Make\_shares(p)$ `// Algorithm 1`
2 Send $p', x \leftarrow length(p), m' \leftarrow attribute(p)$ to $c$ clouds

**Cloud $i$:**
3 $int\_result_i \leftarrow MAP\_count(p', x, m')$
4 Send $int\_result_i$ back to the user

**User-side:**
5 $result[i] \leftarrow int\_result_i, \forall i \in \{1, \cdots, c\}$
6 Compute the final output: $\ell \leftarrow REDUCE(result[])$

7 **Function** $MAP\_count(p', x, m')$ **begin**
8 **for** $i \in (1, n)$ **do**
$temp+ = Automata(SS_i[*, m'], p')$
9 **return**$(\langle key, N_{x+1}^n \rangle)$
10 **Function** $Automata(SS_i[*, m'], p')$ **begin**
$N_1 = 1$
$N_2^i = N_1 \times (SS_i[1, m'] \times p'[1])$
$N_3^i = N_2^i \times (SS_i[2, m'] \times p'[2])$
$\vdots$
$N_{x+1}^i = N_{x+1}^i + N_x^i \times (SS_i[x, m'] \times p'[x])$
**return**$(N_{x+1}^i)$
11 **Function** $REDUCE(result[])$ **begin**
**return**(Assign $result[]$ to a reducer that performs interpolation)

#### 5.1.2 Result reconstruction at the user-side

When we count the occurrences of $p$ in encoded data (non-secret-shares), there is no need for result reconstruction at the user-side. The final value of the node $N_{x+1}$ shows the number of occurrences of $p$, where $x$ is the length of $p$. In our example, the final value of the node $N_5$ shows the number of occurrences of `John` in the relation.

In case of secret-shares, however, we need to reconstruct the final value of the node $N_{x+1}$. The user has $\langle key, value \rangle$ pairs from all the clouds. All the values corresponding to a $key$ are assigned to a reducer that performs the interpolation and provides the final value of the node $N_{x+1}$. If there are more than one reducer, then after the interpolation the sum of the final values shows the number of occurrences of $p$.

**Aside.** If a user searches `John` in a database containing names like 'John' and 'Johnson,' then our algorithm will show two occurrences of `John`. However, it is a problem associated with string matching. In order to search a pattern precisely, we may use the terminating symbol for indicating the end of the pattern. In the above example, we can use "`John `", which is the searching pattern ending with a space, for obtaining the correct answer.

**Steps in Counting a Pattern $p$ using Algorithm 2**

1. User creates secret-share of $p$ (see line 1) and sends secret-share of $p$, length ($x$) of $p$, and the attribute ($m'$) where to count $p$, to $c$ clouds; see line 2.
2. The cloud executes a map function, line 3, that
   *a.* Reads each value of the form of secret-share of the $m'$ attribute and executes AA containing $x + 1$ nodes
   *b.* Executes AA, line 10, and computes the final output of the form of $\langle key, value \rangle$, where a $key$ is an identity of the input split over which the map function was executed, and the value of the form of a secret-share is the final output that shows the number of occurrence of $p$; line 9.
   *c.* The final output after executing AA on each tuple is provided to the user, line 4.
3. User executes a reduce function, for obtaining the final output. The outputs from all the clouds are assigned to reducers based on the keys, and reducers perform the interpolation to provide the final output $\ell$, line 11.

**Theorem 1** *The communication cost, the computational cost at a cloud, and the computational cost at the user-side for counting the occurrences of a pattern is at most $\mathcal{O}(1)$, at most $nw$, and at most $\mathcal{O}(1)$, respectively, where $n$ is the number of tuples in a relation and $w$ is the maximum bit length.*

**Proof.** Since a user sends a patterns of bit length $w$ and receives $c$ values from the clouds, the communication cost is almost constant that is $\mathcal{O}(1)$. The cloud works on a specific attribute containing $n$ values, each of bit length at most $w$; hence, the computational cost at a cloud is at most $nw$. The user only performs the interpolation on the $c$ values; hence, the computational cost at the user-side is also constant, $\mathcal{O}(1)$. ■

### 5.2 Search and Fetch Queries

In this section, we provide a privacy-preserving algorithm for fetching all the tuples containing $p$. The proposed algorithms first execute Algorithm 2 for counting the number of tuples containing $p$, and then, fetch all the tuples after obtaining their addresses. Specifically, we provide 2-phased algorithms, where:
PHASE 1: Finding addresses of tuples containing $p$
PHASE 2: Fetching all the tuples containing $p$

We will present Algorithm 3 for fetching a tuple when a relation has only one tuple containing $p$, in Section 5.2.1, and Algorithm 4 for fetching $\ell > 1$ tuples when a relation has $\ell$ tuples containing $p$, in Section 5.2.2. In both the algorithms, the user follows the similar approach for creating secret-shares and counting the occurrences of $p$, as described in Sections 4 and 5.1, respectively. If readers are still not familiar with the creation of secret-shares and counting the occurrences of $p$, we recommend to find details in Sections 4 and 5.1 before going into details of the search operation.

#### 5.2.1 Unary Occurrence of a Pattern

When only one tuple contains $p$, there is no need to obtain the address of the tuple, and Algorithm 3 fetches the whole tuple in a

**Algorithm 3:** Algorithm for privacy-preserving `search` operation and **fetching a single tuple** from the clouds

**Inputs:** $R$, $n$, $m$, $p$, and $c$ are defined in Algorithm 2
**Output:** A tuple $t$ containing $p$
**Variables:** $\ell$: the number of occurrences of $p$
$int\_result\_search_i$: the output at a cloud $i$ after executing the fetching a single tuple in a privacy-preserving manner
$result\_search[]$: an array to store outputs of all the clouds
$SS_k[i,j]$: defined in Algorithm 2

**User-side:**
1 Compute secret-shares of $p$: $p' \leftarrow Make\_shares(p)$ and execute Algorithm 2 for obtaining the number of occurrences ($\ell$) of $p$
2 **if** $\ell > 1$ **then** Execute Algorithm 4
3 **else** Send $p', x \leftarrow length(p)$ and $m' \leftarrow attribute(p)$ to $c$ clouds

**Cloud $i$:**
4 $int\_result\_search_i \leftarrow MAP\_single\_tuple\_fetch(p', x, m')$
5 Send $int\_result\_search_i$ back to the user

**User-side:**
6 $result\_search[i] \leftarrow int\_result\_search[i]$, $\forall i \in \{1, \cdots, c\}$
7 Obtain the tuple $t \leftarrow REDUCE(result\_search[])$

8 **Function** $MAP\_single\_tuple\_fetch(p', x, m')$ **begin**
9 **for** $i \in (1, n)$ **do**
$temp+ = Automata(SS_i[*, m'], p')$
// Algorithm 2
10 **for** $j \in (1, m)$ **do** $temp \times SS_i[*, j]$
11 **for** $(j, i) \in (m, n)$ **do** $S_j \leftarrow$ add all the shares of $j^{th}$ attribute
12 **return**$(\langle key, N_{x+1}^n || S_1 || S_2 || \ldots || S_m \rangle)$
13 **Function** $REDUCE(result\_search[])$ **begin**
**return**(Assign $result\_search[]$ to a reducer that performs the interpolation)

privacy-preserving manner. Here, we explain how to fetch a single tuple containing $p$. Algorithm 3 works as follows:

*Fetching the tuple*. The user sends secret-shares of $p$. The cloud executes a map function on a specific attribute, and the map function matches $p$ with $i^{th}$ value of the attribute. Consequently, the map function results in either 0 or 1 of the form of secret-shares, if $p$ matches the $i^{th}$ value of the attribute, then the result is 1. After that the map function multiplies the result (0 or 1) by all the $m$ values of the $i^{th}$ tuple. In this manner, the map function creates a relation of $n$ tuples and $m$ attributes. When the map function finishes over all the $n$ tuples, it adds and sends all the secret-shares of each attribute, as: $S_1||S_2||\ldots||S_m$ to the user, where $S_i$ is the sum of the secret-shares of $i^{th}$ attribute. The user on receiving shares from all the clouds executes a reduce function that performs the interpolation and provides the desired tuple containing $p$.

**Aside.** When we multiply the output of the string matching operation, which is of the form of secret-shares, by all the values in a tuple, it results in all the values of the tuple either 0 or 1 of the form of secret-shares. Thus, the sum of all the secret-shares of an attribute results in only the value of the attribute corresponding to the tuple containing $p$.

By performing identical operations (*i.e.*, string matching and multiplication of the result) on each tuple and finally adding all the secret-shares of each attribute, the cloud is unable to know which tuple is fetched.

**Theorem 2** *The communication cost, the computational cost at a cloud, and the computational cost at the user-side for fetching a single tuple containing a pattern is at most $\mathcal{O}(mw)$, at most $\mathcal{O}(nmw)$, and at most $\mathcal{O}(mw)$, respectively, where a relation has $n$ tuples and $m$ attributes and $w$ is the maximum bit length.*

**Proof.** The user sends a pattern of bit length $w$ and receives $c$ secret-shares and, eventually, a tuple containing $m$ attributes of size at most $mw$. Thus, the communication cost is at most $\mathcal{O}(mw)$ bits. The cloud counts the occurrences of the pattern in a specific attribute containing $n$ values, and then again, performs a similar operation on the $n$ tuples, multiplying the resultant by each $m$ values of bit length at most $w$. Hence, the computational cost at the cloud is at most $\mathcal{O}(nmw)$. The user performs the interpolation on $c$ values to know the occurrences of the pattern, and then, again performs the interpolation on $c$ tuples containing $m$ attributes. Thus, the computational cost at the user-side is at most $\mathcal{O}(mw)$. ■

**Steps in Fetching a Single Tuple containing a Pattern $p$ using Algorithm 3**

1. User executes Algorithm 2 for counting occurrences, say $\ell$, of $p$; line 1.
2. If $\ell$ is one, the user sends secret-shares of $p$, length, $x$, of $p$, and attribute, $m'$, where $p$ occurs, to $c$ clouds; line 3.
3. Each cloud executes a map function that
   *a.* Executes AA on $i^{th}$ value of the $m'$ attribute (line 9), and this provides a value, say $val$, either 0 or 1 of form of secret-shares. Multiply $val$ by all the values of $m$ attributes in the $i^{th}$ tuples; line 10.
   *b.* When the execution of AA is completed on all the $n$ secret-shares of the $m'$ attribute, add all the secret-shares an attribute; line 11.
   *c.* Each cloud sends the final output of AA and sum of each attribute's secret-shares to the user; line 12.
4. User receives $c$ tuples and executes a reduce function that performs the interpolation and provides the desired tuple; lines 7 and 13.

### 5.2.2 Multiple Occurrences of a Pattern

When multiple tuples contain $p$, we cannot fetch all those tuples obliviously without obtaining their addresses. Therefore, we first need to perform a pattern search algorithm to obtain the addresses of all the tuples containing $p$, and then, fetch the tuples in a privacy-preserving manner. Throughout this section, we consider that $\ell$ tuples contain $p$. In this section, we provide two algorithms for obtaining the addresses of tuples containing $p$. Both the algorithms have 2-phases, as:
PHASE 1: Finding the addresses of the desired $\ell$ tuples
PHASE 2: Fetching all the $\ell$ tuples

The first algorithm is a naive one and requires two rounds of communication between a user and clouds. The second algorithm is based on the construction of a search tree and requires multiple rounds of communication between a user and clouds, but has lower communication cost. Before going into details of algorithms, we first explore a tradeoff.

**Tradeoff.** When fetching multiple tuples containing $p$, there is a tradeoff between the number of communication rounds and the computational cost at the user-side, and this tradeoff will be clear after the description of the first and the second algorithm. In particular, the user performs a lot of computation when she wants to know the addresses of all the tuples containing $p$ in one round. On the other hand, obtaining the addresses in multiple rounds requires that the cloud has to perform a heavy computation while the user has to perform a simple interpolation.

**Naive algorithm.** A simple and naive algorithm requires only two rounds of communication between a user and the cloud for executing the two-phases, one round for each phase. However, the algorithm requires more workload at the user-side.

*Finding addresses*. The user sends $p$ of the form of secret-shares to the clouds, and the cloud executes a map function that performs a string matching algorithm on secret-shares of each tuple, as we did to count the occurrences of `John` in Section 5.1.1. However, we do not accumulate occurrences, and hence, sends $n$ values corresponding to each tuple. The user implements a reduce function that performs the interpolation and creates a vector, $v$, of length $n$, where $i^{th}$ entity has value either 0 or 1, depending on the occurrence of $p$ in the $i^{th}$ tuple of the relation. As a disadvantage, the user has to work on all the tuples, but the user knows addresses of all the desired tuples in a single round.

*Fetching tuples*. The user creates a $\ell \times n$ matrix, $M$, and creates secret-shares of it, by following the approach suggested in Section 4. All the $n$ columns of a row of the matrix $M$ has 0 but 1 that is dependent on the addresses of the tuples containing $p$. For example, in the vector $v$, if the second position is 1, then we create a row of the matrix $M$ where all the $n$ columns have 0 but the second column has 1. After that, we use $n$ polynomials of identical degree for making secret-shares of all the values and send them to clouds.

Recall that the cloud has a relation of $n$ tuples and $m$ attributes. A mapper in the cloud performs matrix multiplication by multiplying the matrix $M$ with the relation and sends the results to the user. Recall that the matrix $M$ has 0 and 1 of the form of secret-shares, so that the multiplication results in only the desired tuple and all the other tuples are eliminated. The user finally executes a reduce function that performs the interpolation and provides the desired $\ell$ tuples. A similar approach is also presented in [13].

**Theorem 3** *After obtaining the addresses of the desired tuples containing a pattern, $p$, the communication cost, the computational cost at a cloud, and the computational cost at the user-side for fetching the desired tuples is at most $\mathcal{O}((n+m)\ell w)$, $\mathcal{O}(\ell nmw)$, and at most $\mathcal{O}((n + m\ell)w)$, respectively, where a relation has $n$ tuples and $m$ attributes, $w$ is the maximum bit length, and $\ell$ is the number of tuples containing $p$.*

**Proof.** In the first round of the naive algorithm, the user receives $n$ secret-shares, each of bit-length at most $w$, of a particular attribute. In the second round, the user sends a $\ell \times n$ matrix and receives $\ell$ tuples, each of size at most $mw$. Thus, the maximum number of bits flow is $\mathcal{O}((n + m)\ell w)$. A mapper performs string matching operations on $n$ secret-shares of a particular attribute in the first round and then matrix multiplication on all the $n$ tuples and $m$ attributes in the second round. Hence, the computational cost at the cloud is at most $\mathcal{O}(\ell nmw)$. The computational cost at the user-side is at most $\mathcal{O}((n + m\ell)w)$, since the user works on the $n$ secret-shares of a specific attribute, creates a $\ell \times n$ matrix in the first round, and then works on $\ell$ tuples containing $m$ values, each of size at most $w$ bits. ■

**Tree-based algorithm.** In order to decrease the computational load at the user-side, we propose a search-tree-based keyword search algorithm (Algorithm 4) that consists of two phases, as: finding the address of the desired $\ell$ tuples in multiple rounds, and then, fetching all the $\ell$ tuples in one more round.

Taking inspiration form Algorithm 3, we can also obtain the addresses (or line numbers) in a privacy-preserving manner, if only a single tuple contains $p$. Thus, for the case of finding addresses of $\ell$ tuples containing $p$, we divide the whole relation into certain blocks such that each block belongs to one of the following cases:

1) A block contains no occurrence of $p$, and hence, no fetch operation is needed.
2) A block contains one/multiple tuples but only a single tuple contains $p$.
3) A block contains $h$ tuples, and all the $h$ tuples contain $p$.
4) A block contains multiple tuples but fewer tuples contain $p$.

*Finding addresses*. We follow an idea of partitioning the database and counting the occurrences of $p$ in the partitions, until each partition satisfies one of the above mentioned cases. Specifically, we initiate a sequence of Query & Answer (Q&A) rounds. In the first round of Q&A, we count occurrences of $p$ in the whole database (or in an assigned input split to a mapper) and then partition the database into $\ell$ blocks, since we assumed that $\ell$ tuples contain $p$. In the second round, we again count occurrences of $p$ in each block and focus on the blocks satisfying Case 4. There is no need to consider the blocks satisfying Case 2 or 3, since we can apply Algorithm 3 in both the cases. However, if the multiple tuples of a block in the second round contain $p$, *i.e.*, Case 4, we again partition such a block until it satisfies either Case 1, 2 or 3. After that, we can obtain the addresses of the related tuples using the method similar to Algorithm 3.

*Fetching tuples*. We use the approach described in the naive algorithm for fetching multiple tuples after obtaining the addresses of the tuples.

**Aside.** In the above algorithm, we only partition the blocks satisfying Case 4. Consequently, the cloud can deduce that such blocks contain useful information; however, the cloud can never know the information. If we partition all the blocks until each block contains either one or two tuples, the cloud cannot deduce which block is containing useful information. However, the communication and the computational costs increase significantly. Further, we can also fetch the desired tuple during the execution of Q&A rounds, when a block satisfies Case 2 or Case 3. However, the cloud can deduce that some tuples have been fetched from some blocks. In order to completely hide such information, we emphasize that the user should wait until the completion of all the Q&A rounds.

***Example***. Here, we give an example to illustrate the above approach. Let an input split consists of 9 tuples, see Figure 3, and the number of occurrence of $p$ is two. When the user knows the number of occurrences, she starts Q&A rounds. In each Q&A round, a mapper partitions specific parts of the input split into two blocks, performs AA in each blocks, and sends results, which are occurrences of $p$ in each block, of the form of secret-shares back to the user.

In this example, the user initiates the first Q&A round, and a mapper divides the input split into two parts. In each block,

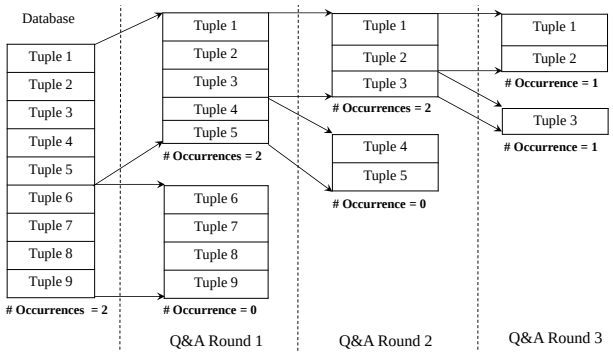


Fig. 3: Example of Q&A rounds.

it counts the occurrences of $p$ and sends the results to the user. The user executes a reducer that performs the interpolation. The user knows that the first and the second blocks contain two and zero tuples having $p$, respectively. The user divides the the first block into two parts again in the second Q&A round. The mapper performs an identical operation as it does in the first round, and after three Q&A rounds, the user have all the two tuples having $p$.

**Algorithm 4's pseudocode description.** A user creates secret-shares of $p$ and obtains the number of occurrence, $\ell$, see line 1. When the occurrences $\ell = 1$, we can perform Algorithm 3 for fetching the only tuple having $p$, see line 2. When the occurrences $\ell > 1$, the user needs to know the addresses of all the $\ell$ tuples contain $p$. Thus, the user requests to partition the input split/relation to $\ell$ blocks, and hence, sends $\ell$ and $p$ of the form of secret-shares to the clouds, see line 3.

The mappers partition the whole relation or input split into $\ell$ blocks, perform privacy-preserving count operation in each blocks, and send all the results back to the user, see lines 4 - 6. The user again executes a reduce function that performs the interpolation and provides the number of occurrences of $p$ in each block, see line 8. Based on the number of occurrences of $p$ in each block, the user decides which block needs further partition, and there are four cases, as follows:

1) The block contains no occurrence of $p$: it is not necessary to handle this block.
2) The block contains only one tuple containing $p$: it is easy to determine its address using function $Address\_fetch()$ that is based on AA; see lines 10 and 14.
3) The block contains $h$ tuples and each $h$ tuple contains $p$: directly know the addresses, *i.e.*, all the $h$ tuples are required to fetch; see line 11.
4) The block contains $h$ tuples and more than one, but less than $h$ tuples contain $p$: we cannot know the addresses of these tuples. Hence, the user recursively requests to partition that block and continues the process until the sub-blocks satisfy the above mentioned Case 2 or Case 3; see lines 9 and 13.

When the user obtains the addresses of all the tuples containing $p$, she fetches all the tuples using a method described for the naive algorithm, see line 12.

**Algorithm 4:** Algorithm for privacy-preserving `search` operation and fetching multiple tuples from the clouds

**Inputs:** $R$, $n$, $m$, $p$, and $c$ are defined in Algorithm 2
**Outputs:** Tuples containing $p$
**Variables:** $\ell$: the number of occurrences of $p$
$int\_result\_block\_count_i[j]$: at $i^{th}$ cloud to store the number of occurrences of the form of secret-shares in $j^{th}$ block
$result\_block\_count[]$: at the user-side to store the count of occurrences of $p$ of the form of secret-shares in each block at each cloud
$count[]$: at the user-side to store the count of occurrences of $p$ in each block
$Address[]$: stores the addresses of the desired tuples

**User-side:**
1 Compute secret-shares of $p$: $p' \leftarrow Make\_shares(p)$ and execute Algorithm 2 for obtaining the number of occurrences ($\ell$) of $p$
2 **if** $\ell = 1$ **then** Execute Steps 3 to 13 of Algorithm 3
3 **else** Send $p', x \leftarrow length(p)$, $m' \leftarrow attribute(p)$, $\ell$ to $c$ clouds

**Cloud $i$:**
4 Partition $R$ into $\ell$ equal blocks, where each block contains $h = \frac{n}{\ell}$ tuples
5 $int\_result\_block\_count_i[j] \leftarrow$ Execute $MAP\_count(p', x, m')$ $j^{th}$ block, $\forall j \in \{1, \cdots, \ell\}$
6 Send $int\_result\_block\_count_i[j]$ back to the user

**User-side:**
7 $result\_block\_count[i, j] \leftarrow int\_result\_block\_count_i[j]$, $\forall i \in \{1, \cdots, c\}, \forall j \in \{1, \cdots, \ell\}$
8 Compute $count[j] \leftarrow REDUCE(result\_block\_count[i, j])$
9 **if** $count[j] \notin \{0, 1, h\}$ **then**
  Question the clouds about $j^{th}$ block and send $\langle p', count[j], m', j\rangle$ to clouds
10 **else if** $count[j] = 1$ **then**
  $Address \leftarrow Address\_fetch(p', x, j)$
11 **else if** $count[j] = h$ **then**
  $Address \leftarrow (j-1)h+1, (j-1)h+2, \cdots, (j-1)h+h$
12 Fetch the tuples whose addresses are in $Address$ using a method described in the naive algorithm

**Cloud $i$:**
13 **if** *Receive* $\langle p', count[j], m', j\rangle$ **then** Perform Steps 4 to 6 to $j^{th}$ block recursively

14 **Function** $Address\_fetch(p', x, j)$ **begin**
  $line\_number \leftarrow 0$
15  **for** $i \in ((j-1)h+1, (j-1)h+h)$ **do**
16  $line\_number += Automata(SS_i[*, m'], p') \times i$ `// Algorithm 2`
17  **return**$(line\_number)$

**Theorem 4** *The maximum number of rounds for obtaining addresses of tuples containing a pattern, p, using Algorithm 4 is* $\lfloor log_\ell n \rfloor + \lfloor log_2 \ell \rfloor + 1$*, and the communication cost for obtaining such addresses is at most* $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell\big)$*. The computational cost at a cloud and the computational cost at the user-side is at most* $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell n w\big)$ *and at most* $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell\big)$*, respectively, where a relation has* $n$ *tuples and* $m$ *attributes,* $\ell$ *is the number of tuples containing* $p$*, and* $w$ *is the maximum bit length.*

**Proof.** According to the description of Algorithm 4, in the current Q&A round, we partition the blocks that are specified in last round

into $\ell$ blocks equally. Thus, in $i^{th}$ round of Q&A, the number of the items contained in each sub-block is at most $\frac{n}{\ell^i}$.

After $\lfloor \log_\ell n \rfloor$ rounds of Q&A, the number of the items contained in every block is fewer than $\ell$. At this time, note that there may be some blocks still contain more than one tuple containing $p$. Thus, we need at most $\lfloor \log_2 \ell \rfloor$ rounds for determining the addresses of those tuples. When the user finishes partitioning all the blocks that contains more than two tuples containing $p$, it needs at most one more round for obtaining the addresses of related tuples. Thus, the total number of Q&A round is at most $\lfloor log_\ell n \rfloor + \lfloor log_2 \ell \rfloor + 1$.

Notice that for each round, there are at most $\frac{\ell}{2}$ blocks containing more than two tuples containing $p$ that indicates that at most $\frac{\ell}{2}$ blocks need further partitioning. So in every Q&A round (except the first round requires $\ell$ answers), each cloud only needs to perform `count` operation for $\frac{\ell}{2}$ sub-blocks and send the results back to the user. When the cloud finishes partitioning, it has to perform $Address\_fetch()$ operation to determine the addresses. It requires at most $\ell$ words transition between the user and each cloud. Therefore, the communication cost is at most $\mathcal{O}\big((log_\ell n + log_2 \ell) \cdot \ell\big)$.

A cloud performs `count` operation in each round, hence the computational cost at the cloud is at most $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell n w\big)$. In each round, the user performs the interpolation for obtaining the occurrences of the pattern in each block; hence the computational cost at the user-side is at most $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell\big)$. ■

***Example.*** In figure 3, in order to fetch tuples containing $p$, the user needs 3 rounds, which are less than $\lfloor \log_2 9 \rfloor + \lfloor \log_2 2 \rfloor + 1 = 5$.

### 5.3 Equijoin

In this section, we show how to perform the equijoin in a privacy-preserving manner using MapReduce. Throughout this section, we consider two relations $X(A, B)$ and $Y(B, C)$ containing $n$ tuples in each, where the joining attribute is $B$. A trivial way for performing the equijoin in a privacy-preserving manner, as follows: (*i*) fetch all the secret-shares of $B$-values from all the clouds and perform the interpolation, (*ii*) find tuples of both the relations that have an identical $B$-value and fetch all those tuples, (*iii*) perform the interpolation on the tuples, and (*iv*) perform a MapReduce job for joining the tuples at the user-side. However, in this approach the user has to perform the interpolation and MapReduce-based join.

In order to decrease the workload at the user-side, we propose two approaches so that the user has to perform only the interpolation on the output tuples of the join. The first approach assumes that the relations $X$ and $Y$ have at most one occurrence of $B$ values in each, and the second approach does not hold any restriction on the occurrences of $B$-values, *i.e.*, a $B$-value can occur in multiple tuples of the relations.

#### 5.3.1 A unique occurrence of the joining value

We use string matching operations (a variant of Algorithm 3) on secret-shares for performing the equijoin. The following steps are executed for performing the equijoin when a joining value occurs in at most one tuple of a relation, as:

1. In a cloud:
   - *a.* A mapper reads $i^{th}$ tuple $\langle a_i, b_i \rangle$ of the relation $X$ and provides a pair of $\langle key, value \rangle$, where a $key$ is an identity $i$ and a $value$ is secret-shares of $\langle a_i, b_i \rangle$.
   - *b.* A mapper reads $j^{th}$ tuple $\langle b_j, c_j \rangle$ of the relation $Y$ and provides $n$ pairs of $\langle key, value \rangle$, where a $key$ is an identity from 1 to $n$ and a $value$ is secret-shares of $\langle b_j, c_j \rangle$.
   - *c.* A reducer $i$ is assigned $\langle i, [a_i, b_i] \rangle$, where $a_i, b_i \in X$, and all the tuples of the relation $Y$. The reducer performs string matching operations on the $B$ values that result in 0 or 1 of the form of secret-share. Specifically, the reducer matches $b_i \in X$ with each $b_j \in Y$, and the resultant of the string matching operation ($b_i$ and $b_j$) is multiplied by the tuple $\langle b_j, c_j \rangle$. After performing the string matching operation on all the $B$-values of the relation $Y$, the reducer adds all the secret-shares of the attributes $B$ and $C$. The sum of the $B$-values is multiplied by the tuple $\langle a_i, b_i \rangle$ and the sum of the $C$-values is appended to this tuple. Thus, a new tuple is obtained as $\langle a', b', c' \rangle$.
2. The user fetches all the outputs of reducers from all the clouds, performs the interpolation, and obtains the outputs of the equijoin.

| $A$ | $B$ |
|---|---|
| $a_1$ | $b_1$ |
| $a_2$ | $b_2$ |
| $a_3$ | $b_3$ |

| $B$ | $C$ |
|---|---|
| $b_1$ | $c_1$ |
| $b_2$ | $c_2$ |
| $b_4$ | $c_4$ |

Fig. 4: Two relations $X(A, B)$ and $Y(B, C)$.

***Example.*** We consider two relations $X$ and $Y$, see Figure 4. Consider that all values are of the form of secret-shares. Mappers in the cloud read the tuples $\langle a_1, b_1 \rangle$, $\langle a_2, b_2 \rangle$, and $\langle a_3, b_3 \rangle$ and provide $\langle 1, [a_1, b_1] \rangle$, $\langle 2, [a_2, b_2] \rangle$, and $\langle 3, [a_3, b_3] \rangle$, respectively. The mapper reads the tuple $\langle b_1, c_1 \rangle$ and provides $\langle 1, [b_1, c_1] \rangle$, $\langle 2, [b_1, c_1] \rangle$, and $\langle 3, [b_1, c_1] \rangle$. A similar operation is also carried out on the tuples $\langle b_2, c_2 \rangle$ and $\langle b_4, c_4 \rangle$.

A reducer corresponding to key 1 matches $b_1$ of $X$ with $b_1$ of $Y$ that results in 1, then $b_1$ of $X$ with $b_2$ of $Y$ that results in 0, and $b_1$ of $X$ with $b_4$ of $Y$ that results in 0. Remember 0 and 1 are of the form of secret-shares. Now, the reducer multiplies the three values (1,0,0) of the form of secret-shares by the tuples $\langle b_1, c_1 \rangle$, $\langle b_2, c_2 \rangle$, and $\langle b_4, c_4 \rangle$, respectively. After that the reducer adds all the $B$-values and the $C$-values. Note that we will obtain now only the desired tuple, *i.e.*, $\langle b_1, c_1 \rangle$. The reducer multiplies the sum of all the $B$-values by the tuple $\langle a_1, b_1 \rangle$ appended with the sum of all the $C$-values. The same operation is carried out on other $B$-values of the relation $X$. When the user performs the interpolation on the outputs of all the clouds, only the desired output tuples of the equijoin are obtained, and all the other tuples, for example $\langle a_3, b_3 \rangle$, hold value zero. In this manner, the user performs the equijoin in a privacy-preserving manner without knowing undesired tuples.

**Aside.** We assume that the all the $A$, $B$, and $C$ values of the relations do not contain zero.

**Theorem 5** *The communication cost, the computational cost at a cloud, and the computational cost at the user-side for performing the equijoin of two relations $X$ and $Y$, where a joining value can occur at most one time in a relation, is at most $\mathcal{O}(nmw)$, at most $\mathcal{O}(n^2mw)$, and at most $\mathcal{O}(nmw)$, respectively, where a relation has $n$ tuples and $m$ attributes and $w$ is the maximum bit length.*

**Proof.** Since the user receives the whole relation of $n$ tuples and at most $2m - 1$ attributes, the communication cost is at most $\mathcal{O}(nmw)$, and due to the interpolation on the $n$ tuples, the

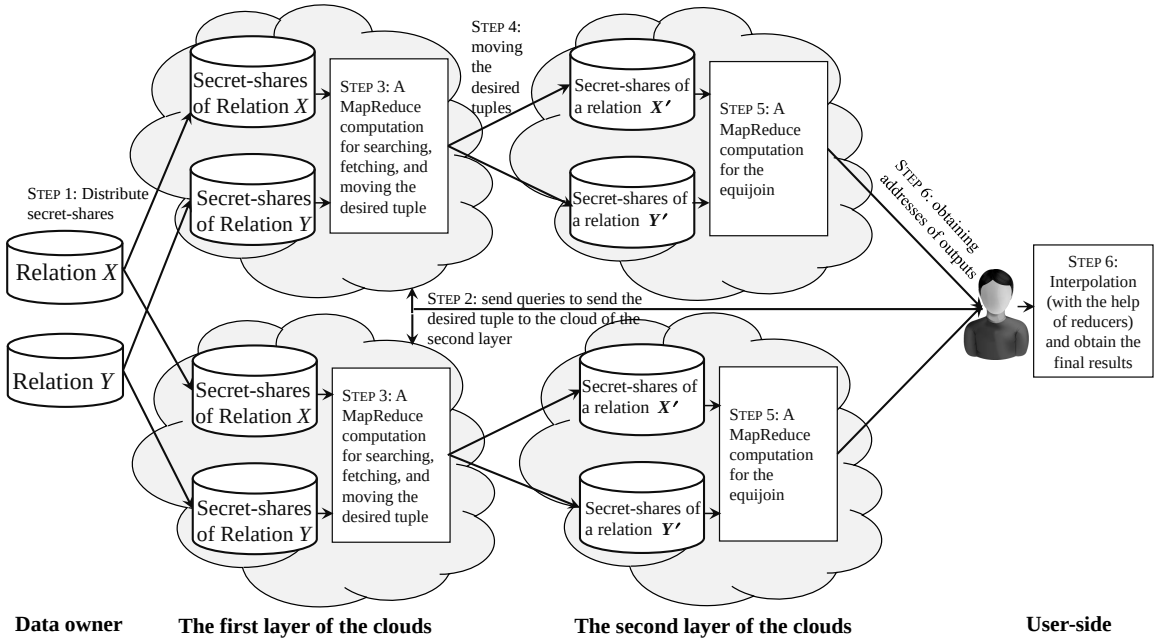


Fig. 5: The new system architecture for the equijoin.

computational cost at the user-side is at most $\mathcal{O}(nmw)$. A mapper compares each value of the joining attribute of the relation $X$ to all $n$ values of the joining attribute of the relation $Y$, and it results in at most $n^2$ comparisons. Further, the output of the comparison is multiplied by $m-1$ attributes of the corresponding tuple of the relation $Y$. Hence, the computational cost at a cloud is at most $\mathcal{O}(n^2mw)$. ■

### 5.3.2 Multiple occurrences of the joining value

We present an algorithm for performing the equijoin when many tuples of a relation have an identical joining value. Consider two relations $X(A, B)$ and $Y(B, C)$, see Figure 6.

| $A$ | $B$ |
|---|---|
| $a_1$ | $b_1$ |
| $a_2$ | $b_1$ |
| $a_3$ | $b_2$ |

| $B$ | $C$ |
|---|---|
| $b_1$ | $c_1$ |
| $b_1$ | $c_2$ |
| $b_3$ | $c_3$ |

Fig. 6: Two relations $X(A, B)$ and $Y(B, C)$ with multi occurrences of a $B$-value.

Note that if we follow the previous approach, Section 5.3.1, then we take $b_1$ of $X$, multiply $b_1$ by all three $B$-values of $Y$, and add all secret-shares. However, after addition, we cannot distinguish two occurrences of $b_1$ in $Y$. Hence, we present a new algorithm and system settings for this type of the equijoin.

**New System Setting.** We need a new system setting only for the equijoin when a joining value occurs many times in a relation, see Figure 5. Recall that in the current system setting, which is mentioned in Section 2, we use $c$ non-communicating clouds to store secret-shares of a relation. Here, we introduce one more layer of the clouds. The clouds within a layer are not allowed to communicate; however, the clouds of the first layer and the second layer can communicate, see STEP 4 of Figure 5.

A data owner outsources databases, *i.e.*, the relations $X$ and $Y$, of the form of secret-shares to $c$ (non-communicating) clouds of the first layer only once; see STEP 1 in Figure 5. The user sends a query of the form of secret-shares to all $c$ clouds of the first layer to find the desired tuples and send them to the clouds of the second layer; see STEP 2 in Figure 5. The clouds of the first layer execute the multi-tuple fetch algorithm (presented in Section 5.2.2) and send the desired tuples to the clouds of the second layer; see STEPs 3 and 4 in Figure 5. The cloud of the second layer creates two relations from the selected tuples of $X$ and $Y$, and performs the join operation on secret-shares; see STEP 5 in Figure 5. Finally, the user fetches the outputs from the clouds of the second layer and performs the interpolation for obtaining secret-values; see STEP 6 in Figure 1. We will explain all these steps with the help of an example shortly.

**The Approach.** The approach consists of the following three steps, where the second step that perform the equijoin is executed in the clouds, as follows:

1. The user fetches all the $B$-values of the relations $X$ and $Y$ and performs the interpolation. After the interpolation, the user knows which $B$-values are identical in both relations and in which tuples they are.
2. For each $B$-value (say, $b_i$) that is in both relations:
   - *a.* The user requests the clouds of the first layer to send all the tuples containing $b_i$ to a cloud of the second layer. This operation is done using the naive algorithm for fetching multiple tuples, refer to Section 5.2.2.
   - *b.* On receiving tuples containing the joining value $b_i$ from the clouds of the first layer, the clouds in the second layer create two new relations corresponding to the tuples of $X$ and $Y$. Then, the clouds in the second layer execute a MapReduce job that concatenates a tuple of the first relation to all the tuples of the second relation and provides the output of the equijoin, since the two new relations have only one identical $B$-value.
3. The user fetches all the output tuples from the second layer of the clouds and performs the interpolation.

***Example.*** For the relations $X$ and $Y$, see Figure 6, the user

fetches all the $B$-values of both the relations and performs the interpolation. After the interpolation, the user knows that the joining value $b_1$ appears in the first tuple and second tuple of both the relations. The user follows the naive algorithm for fetching multiple tuples containing $b_1$ (refer to Section 5.2.2) and asks the clouds of the first layer to send these four tuples to the clouds of the second layer.

The cloud of the second layer creates two new relations $X'$ containing $\langle a_1, b_1\rangle$ and $\langle a_2, b_1\rangle$, and $Y'$ containing $\langle b_1, c_1\rangle$ and $\langle b_1, c_2\rangle$. A mapper reads a tuple and provides $\langle key, value\rangle$ pairs, where a $key$ is an identity and a $value$ is the tuple. A reducer holds all the tuples of both the relations $X'$ and $Y'$ and joins (or concatenates) the first and the second tuples of $X'$ to both the tuples of $Y'$. Finally, the user fetches the output and executes the interpolation.

**Theorem 6** *The number of rounds, the communication cost, the computational cost at a cloud, and the computational cost at the user-side for performing the equijoin of two relations $X$ and $Y$, where a joining value can occur in multiple tuples of a relation, is at most $\mathcal{O}(2k)$, at most $\mathcal{O}(2nwk + 2k\ell^2mw)$, at most $\mathcal{O}(\ell^2kmw)$, and at most $\mathcal{O}(2nw + 2k\ell^2mw)$, respectively, where a relation has $n$ tuples and $m$ attributes, $k$ is the number of identical values of the joining attribute in the relations, $\ell$ is the maximum number of occurrences of a joining value, and $w$ is the maximum bit length.*

**Proof.** Since there are at most $k$ identical values of the joining attribute in both the relations and all the $k$ values can have different number of occurrences in the relations, the user has to send at most $2k$ matrices (following an approach of the naive algorithm for fetching multiple tuples) in $\mathcal{O}(2k)$ rounds.

The user sends at most $2k$ matrices, each of $n$ rows and of size at most $w$; hence, the user sends at most $\mathcal{O}(2knw)$ bits. Since at most $\ell$ tuples have an identical value of the joining attribute in one relation, the equijoin provides at most $\ell^2$ tuples. The user receives at most $\ell^2$ tuples for each $k$ value having at most $2m-1$ attributes; hence, the user receives at most $\mathcal{O}(2k\ell^2mw)$ bits. Therefore, the communication cost is at most $\mathcal{O}(2nwk + 2k\ell^2mw)$ bits.

The cloud of the first layer executes the naive algorithm for fetching multiple tuples for all $k$ values of both relations having $2n$ tuples; hence the clouds of the first layer performs at most $\mathcal{O}(2nkw)$ computation. In the second layer, a cloud performs the equijoin (or concatenation) of at most $\ell$ tuples for each $k$ value; thus, the computational cost at the cloud is at most $\mathcal{O}(\ell^2kmw)$.

The user first interpolates at most $2n$ values of bit length $w$ of the joining attribute, and then, interpolates $k\ell^2$ tuples containing at most $2m-1$ attributes of bit length $w$. Therefore, the computational cost at the user-side is at most $\mathcal{O}(2nw+2k\ell^2mw)$. ■

### 5.4 Range Query

A range query finds, for example, all the employees whose salaries are between \$1000 and \$2000. We propose an approach for performing privacy-preserving range queries based on 2's complement subtraction. A number, say $x$, belongs in a range, say $[a, b]$, if $sign(x - a) = 0$ and $sign(x - b) = 0$, where $sign(x - a)$ and $sign(b - x)$ denote the sign bits of $x - a$ and $x - b$, respectively, after 2's complement based subtraction.

Recall that in Section 4, we proposed an approach for creating secret-shares of a number, $x$, using unary representation that provides a vector, where all the values are 0 except only 1 according to the position of the number. The approach works well to count the occurrences of $x$ and fetch all the tuples having $x$. However, on this vector, we cannot perform subtraction operation. Hence, in order to execute range queries, we present a number using binary-representation, which results in a vector of length, say $l$. After that we use SSS to make secret-shares of every bit in the vector by selecting $l$ different polynomials of an identical degree for each bit position.

**Algorithm 5:** $SS\text{-}SUB(A, B)$: 2's complement based subtraction of secret-sharing

**Inputs:** $A = [a_{t-1}a_{t-2}\cdots a_1a_0], B = [b_{t-1}b_{t-2}\cdots b_1b_0]$ where $a_i, b_i$ are secret-shares of bits of 2's complement represented number, $t$: the length of $A$ and $B$ in binary form

**Outputs:** $rb_{t-1}$: the sign bit of $B - A$

**Variable:** $carry[]$: to store the carry for each bit addition
$rb$: to store the result for each bit addition

```
a_0 ← 1 − a_0                                   // Invert of the LSB of A
carry[0] ← a_0 + b_0 − a_0 · b_0
rb_0 ← a_0 + b_0 − 2 · carry[0]                 // ā_0 + b_0 + 1
for i ∈ (i, t − 1) do
    a_i ← 1 − a_i                               // invert each bit A → Ā
    rb_i ← a_i + b_i − 2a_ib_i
    carry[i] ← a_ib_i + carry[i − 1] · rb_i     // The carry bit
    rb_i+ = carry[i − 1] − 2 · carry[i − 1] · rb_i
return(rb_{t−1})                                // The sign bit of B − A
```

**The approach.** The idea of finding whether a number, $x$, belongs to the range, $[a, b]$, is based on 2's complement subtraction. In [23], the authors provided an algorithm for subtracting secret-shares using 2's complement. However, we will provide a simple 2's complement based subtraction algorithm for secret-shares, see Algorithm 5. A mapper checks the sign bits after subtraction for deciding the number whether it is in the range or not, as follows:

$$\begin{array}{ll} \text{If } x \in [a, b], & sign(x - a) = 0, sign(b - x) = 0 \\ \text{If } x < a, & sign(x - a) = 1, sign(b - x) = 0 \\ \text{If } x > b, & sign(x - a) = 0, sign(b - x) = 1 \end{array} \quad (1)$$

After checking each number, we can use one of the following approaches:

1) *A simple solution*. The mapper sends the sign bit's values of the form of secret-shares to the user for each tuple. The user then implements a reduce function that performs the interpolation and creates an array of length $n$. If the number $x$ in $i^{th}$ tuples belongs in the range $[a, b]$, then the $i^{th}$ position in the array is one. Otherwise, the $i^{th}$ position in the array is zero. Finally, the user fetches all the tuples having value 1 in the array using the naive algorithm for fetching multiple tuples, see Section 5.2.2.
2) It keeps the count of all the numbers that belong in the range and sends the count to the user that interpolates them. After knowing how many numbers are in the range, the user can implement Algorithm 3 or 4 for fetching the desired tuples; see Algorithm 6. However, in this manner, we have to check the numbers whether they are in the range or not at the time of fetching the tuple. In this approach, we use many rounds for fetching the desired tuples; however, at the user-side, the computational cost decreases.

**Algorithm 6:** Algorithm for privacy-preserving range query in the clouds using MapReduce

**Inputs:** $R$, $n$, $m$, and $c$: defined in Algorithm 2, $[a, b]$: a searching range
**Output:** $\ell$: the number of occurrence in $[a, b]$
**Variables:** $int\_result_i$: is initialized to 0 and the output at $i^{th}$ cloud after executing the $MAP\_range\_count$ function

**User-side:**
1 Compute secret-shares of $a, b$: $a' \leftarrow Make\_shares(a)$, $b' \leftarrow Make\_shares(b)$
2 Send $a', b', m' \leftarrow attribute(a)$ to $c$ clouds

**Cloud $i$:**
3 **for** $i \in (1, n)$ **do** $int\_result_i \leftarrow MAP\_range\_count(a', b', m')$
4 Send $int\_result_i$ back to the user

**User-side:**
5 $result[i] \leftarrow int\_result_i, \forall i \in \{1, \cdots, c\}$
6 $\ell \leftarrow REDUCE(result[])$
7 Execute Algorithm 3 if $\ell = 1$; otherwise, execute Algorithm 4

8 **Function** $MAP\_range\_count(a', b', m')$ **begin**
9 $sign_{x-a'} \leftarrow SS\text{-}SUB(x, a')$ `// Algorithm 5`
10 $sign_{b'-x} \leftarrow SS\text{-}SUB(b', x)$ `// Algorithm 5`
11 **return**$(\langle key, 1 - sign_{x-a'} - sign_{b'-x} \rangle)$

12 **Function** $REDUCE(result[])$ **begin**
**return**(Assign $result[]$ to a reducer that performs the interpolation)

**Algorithm 5's pseudocode description.** Algorithm 5 provides a way to perform 2's complement based subtraction on secret-shares. We follow the definition of 2's complement subtraction to convert $B - A$ into $B + \bar{A} + 1$, where $\bar{A} + 1$ is 2's complement representation of $-A$. We start at the least significant bit (LSB), invert $a_0$, calculate $\bar{a}_0 + b_0 + 1$ and its carry bit, see lines 1- 3. Then, we go through the rest of the bits, calculate the carry and the result for each bit, see line 4. After finishing all the computations, the most significant bit (MSB) or the sign bit is returned; see line 5.

Algorithm 5 is similar to the algorithm presented in [23], but simpler, as we only need the sign bit of the result. After obtaining secret-shares of sign bits of $x - a$ and $b - x$, we perform an extra calculation:

$$1 - \big(sign(x - a) + sign(b - x)\big). \tag{2}$$

According to Equation 1, if $x \in [a, b]$, the result of Equation 2 is secret-share of 1; otherwise, the result is secret-share of 0. Based on Equation 2, we can obtain the number of occurrences, which are in the required range in the database.

**Algorithm 6's pseudocode description.** Algorithm 6 works in two phases, as: first, it counts the occurrences of numbers that belong in a range, and second, it fetches all those corresponding tuples. A user creates secret-shares of the range numbers $a, b$ and sends them to $c$ clouds, see lines 1 and 2.

The cloud executes a map function that checks each number in an input split by implementing Algorithm 5, see lines 3 and 8. The map function, see line 8, provides 1 (of the form of secret-share) if $x \in [a, b]$; otherwise, 0 (of the form of secret-share) if $x \notin [a, b]$. The cloud provides the number of occurrences (of the form of secret-shares) that belong in the ranges to the user, see line 4. The user receives all the values from $c$ clouds and execute a reduce function that interpolates them to obtain the count, see lines 5 and 6. After obtaining the number of occurrences, say $\ell$, the user can fetch the corresponding tuples by following Algorithm 3 or 4.

*Degree reduction.* Note that in range query, we utilize 2's complement subtraction and each secret-shared bit of the operands. However, during the subtraction procedure, the degree of the polynomial (for secret-sharing) increases. For example, one can check that degree of MSB doubles when one subtraction completed. In [23], the authors add two more players for degree reduction, if we do not have enough clouds to recover the secrets, we can follow the same line as the degree reduction algorithm presented in [23]. For simplicity, we do not give the details of the algorithms for degree reduction, interested readers may refer to [23], [24].

**Theorem 7** *The communication and the computational costs of the range count query have the same order of magnitude as Algorithm 2, and the communication and computational cost of fetching multi-tuples satisfying a range have the same order of magnitude as Algorithm 3 or 4.*

Note that the function $MAP\_range\_count()$, see line 8 of Algorithm 6, works on each value of a specific attribute as we did in `count` queries, Section 5.1. Once we know all the occurrences of tuples satisfying a range, we find their address using Algorithm 3 or 4. Thus, the communication and the computational costs have an identical order of magnitude as Algorithm 3 or 4.

## 6 Conclusion

MapReduce provides efficient large-scale data processing without dealing with the privacy and security of data and computations. However, a database owner can ensure security and privacy of data by allowing users to execute their queries at the side of the database owner so that a heavy computation is required at the side of the database owner for maintaining a database and executing queries. Thus, in order to avoid overheads for maintaining and executing queries at the side of the database owner, a database is outsourced to untrusted public clouds that can reveal the database or computations. In this paper, we proposed a new information-theoretically secure data and computation outsourcing technique. By the proposed techniques, users can execute their computations in the public cloud without the need of the database owner, and the cloud cannot learn the database or the computations. Specifically, we provided MapReduce based privacy-preserving algorithms to execute count, search and fetch, equijoin, and range quires in the public clouds. As compared to the existing algorithms, our algorithms provide perfect privacy protection without introducing computation and communication overheads.

There are two prominent open questions, such as (*i*) how to optimize the workload at the user-side in case of multiple-tuple fetch and equijoin operations, and (*ii*) how to extend the proposed technique for several MapReduce-based operations, *e.g.*, graph processing and different types of operations on databases.

## REFERENCES


[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, 2004, pp. 137–150. [Online]. Available: http://www.usenix.org/events/osdi04/tech/dean.html

[2] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2014.

[3] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979. [Online]. Available: http://doi.acm.org/10.1145/359168.359176

[4] S. Dolev, N. Gilboa, and X. Li, "Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation: Extended abstract," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing, SCC@ASIACCS '15, Singapore, Republic of Singapore, April 14, 2015*, 2015, pp. 21–29. [Online]. Available: http://doi.acm.org/10.1145/2732516.2732526

[5] E. Blass, R. D. Pietro, R. Molva, and M. Önen, "PRISM - privacy-preserving search in MapReduce," in *Privacy Enhancing Technologies - 12th International Symposium, PETS 2012, Vigo, Spain, July 11-13, 2012. Proceedings*, 2012, pp. 180–200. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31680-7_10

[6] T. Mayberry, E. Blass, and A. H. Chan, "PIRMAP: efficient private information retrieval for mapreduce," in *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, 2013, pp. 371–385. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39884-1_32

[7] E. Blass, G. Noubir, and T. V. Huu, "EPiC: efficient privacy-preserving counting for MapReduce," p. 452, 2012. [Online]. Available: http://eprint.iacr.org/2012/452

[8] S. D. Tetali, M. Lesani, R. Majumdar, and T. D. Millstein, "MrCrypt: static analysis for secure cloud computations," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, 2013, pp. 271–286. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509554

[9] J. J. Stephen, S. Savvides, R. Seidel, and P. Eugster, "Practical confidentiality preserving big data analysis," in *6th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud '14, Philadelphia, PA, USA, June 17-18, 2014.*, 2014. [Online]. Available: https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/stephen

[10] P. Derbeko, S. Dolev, E. Gudes, and S. Sharma, "Security and privacy aspects in MapReduce on clouds: A survey," *Computer Science Review*, 2016.

[11] F. Emekçi, D. Agrawal, A. El Abbadi, and A. Gulbeden, "Privacy preserving query processing using third parties," in *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, 2006, p. 27. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2006.116

[12] F. Emekçi, A. Metwally, D. Agrawal, and A. El Abbadi, "Dividing secrets to secure data outsourcing," *Inf. Sci.*, vol. 263, pp. 198–210, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2013.10.006

[13] W. Lueks and I. Goldberg, "Sublinear scaling for multi-client private information retrieval," in *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, 2015, pp. 168–186. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-47854-7_10

[14] L. Li, M. Militzer, and A. Datta, "rPIR: Ramp secret sharing based communication efficient private information retrieval," *IACR Cryptology ePrint Archive*, vol. 2014, p. 44, 2014. [Online]. Available: http://eprint.iacr.org/2014/044

[15] B. Chor, N. Gilboa, and M. Naor, "Private information retrieval by keywords," *IACR Cryptology ePrint Archive*, vol. 1998, p. 3, 1998. [Online]. Available: http://eprint.iacr.org/1998/003

[16] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, 1987, pp. 369–378. [Online]. Available: http://dx.doi.org/10.1007/3-540-48184-2_32

[17] R. M. Corless and N. Fillion, "A graduate introduction to numerical methods," *AMC*, vol. 10, p. 12, 2013.

[18] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *INFOCOM 2010. 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 15-19 March 2010, San Diego, CA, USA*, 2010, pp. 534–542. [Online]. Available: http://dx.doi.org/10.1109/INFCOM.2010.5462174

[19] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 222–233, 2014. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2013.45

[20] F. G. Olumofin and I. Goldberg, "Privacy-preserving queries over relational databases," in *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings*, 2010, pp. 75–92. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14527-8_5

[21] T. B. Pedersen, Y. Saygın, and E. Savaş, "Secret charing vs. encryption-based techniques for privacy preserving data mining," 2007.

[22] L. R. Welch and E. R. Berlekamp, "Error correction for algebraic block codes," Dec. 30 1986, US Patent 4,633,470.

[23] S. Dolev and Y. Li, "Secret shared random access machine," in *Algorithmic Aspects of Cloud Computing - First International Workshop, ALGOCLOUD 2015, Patras, Greece, September 14-15, 2015. Revised Selected Papers*, 2015, pp. 19–34. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-29919-8_2

[24] S. Dolev, J. A. Garay, N. Gilboa, and V. Kolesnikov, "Brief announcement: swarming secrets," in *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, 2010, pp. 231–232. [Online]. Available: http://doi.acm.org/10.1145/1835698.1835750