CONTENTS

ABSTRACT 1
1. INTRODUCTION 1
2. PROBLEM STATEMENT 2
2.1. THREAT MODEL 2
2.2. SECURITY 2
3. RELATED WORK 3
4. ANALYSIS OF THE PINED-RQ FAMILY 3
4.1. OVERVIEW 3
4.2. LIMITATIONS 4
5. INGESTION FRAMEWORK FOR SECURE 5
5.1. KEY DESIGN FEATURES 5
5.2. UPGRADING PINED-RQ++ FOR COPING WITH 5
5.3. ARCHITECTURE OF FRESQUE 6
6. SECURITY ANALYSIS 8
7. EVALUATION 9
7.1. BENCHMARK ENVIRONMENT 9
7.2. RESULTS 9
8. DISCUSSION 11
9. CONCLUSION 12
REFERENCES 12

FRESQUE: A Scalable Ingestion Framework for Secure Range Query Processing on Clouds

Hoang Van Tran[1,3], Tristan Allard[1], Laurent d'Orazio[1], Amr El Abbadi[2]
[1]Univ Rennes & CNRS & IRISA, [2]UC Santa Barbara, [3]Can Tho University

ABSTRACT

Performing non-aggregate range queries over encrypted data stored on untrusted clouds has been considered by a large body of work over the last years. However, prior schemes mainly concentrate on improving query performance while the scalability dimension still remains challenging. Due to heavily pre-processing incoming data at a trusted component such as encrypting data and building secure indexes, existing solutions cannot provide a satisfactory ingestion throughput. In this paper, we overcome this limitation by introducing a framework for secure range query processing, FRESQUE, that enables a scalable consumption throughput while still maintaining strong privacy protection for outsourced data. Our experiments on real-world datasets show that FRESQUE can support over 160 thousand record insertions in a second, when running on a 12-computing node cluster. It also significantly outperforms one of the most efficient schemes such as PINED-RQ++ by 43 times on ingestion throughput.

1 INTRODUCTION

With the prosperity of online social networks and web-based services, a large amount of personal data is collected every second. To achieve analytical and administrative purposes, it becomes increasingly desirable for modern systems to support not only low-latency queries, but also intensive ingestion throughput over incoming data. For instance, to reduce the impact of seasonal epidemics (e.g., influenza), participatory surveillance systems, to name few, have been deployed in recent years such as Flu Near You [2] in North America, Influenzanet [16] in Europe, and FluTracking [3, 9] in Australia and New Zealand. These systems weekly collect symptoms from their participants to track influenza nation-wide. Such systems are expected to receive a huge number of records every second.

Due to the significant costs necessary for building and maintaining such systems (e.g., computing and scalability requirements, human resources), it may be worth to outsource user data to a cloud service, that can provide lower costs and enable elastic scaling [14]. However, parts of the data may be sensitive, e.g., participants' symptoms, and sociodemographic data (age, gender, etc.). Managing sensitive data on a public cloud increases the risk of unauthorized disclosure since its infrastructure may be compromised by an adversary. According to a recent survey, 52% of companies use cloud services that have experienced a data breach [18].

This paper considers the following cloud computing model (Figure 1): a collector receives data from multiple data sources and stores them on a cloud while a client retrieves the data by using queries. For a simple example, Flu survey participants submit electronic medical records to a Center for Disease Control and Prevention (CDC) that stores the collected data on a cloud. An epidemiologist queries these records from the cloud. In this model, only the cloud is untrusted while the others are trusted.

Figure 1: An architecture of cloud computing

Encryption is a standard technique to ensure the confidentiality of data stored on untrusted environments like clouds [29, 32, 35]. In this study, we focus on non-aggregate range queries over encrypted data since they are fundamental operations. For example, a doctor, Alice, wants to get encrypted electric records of the patients whose ages are between a and b. One example of a range query (expressed with SQL) can be: `SELECT * FROM` database `WHERE` age $\geq$ a `AND` age $\leq$ b.

Although many studies in this line of work have been done over the last years [5–8, 10, 19, 20, 24, 26, 30, 31, 36], none of them satisfies real-life scalability requirements. It is also non-trivial to scale these schemes due to their own limitations. For instance, Hidden Vector Encryption (HVE) methods [8, 36] use bilinear groups equipped with bilinear maps and hide attributes in an encrypted vector. However, they suffer from high latency because it is extremely costly to compute exponentiation and pairing in a composite-order group. Meanwhile, some recent schemes [10, 23, 24] attempt to maintain secure indexes, that rely on *Searchable Symmetric Encryption*, for efficiency. Unfortunately, the secure indexes not only create high space overhead, but also require at least hundreds of second for construction that may lead to bottlenecks. Even though ArxRange [30] does not experience long index building time, it incurs prohibitive storage overhead and only supports a modest ingestion throughput, e.g., about 450 writes per second with caching. Recently, solutions based on differential privacy [11], e.g., the PINED-RQ family [33, 34], have been considered and achieve very good performance in terms of computation and space requirements, however, they do not render a satisfactory ingestion throughput. In particular, PINED-RQ [33] incurs congestion as incoming data rate is high. Meanwhile, PINED-RQ++ [34] experiences modest throughput, ~46K records/s.

Therefore, we aim at developing a scalable ingestion framework dedicated to secure range query processing. To the best of our knowledge, this is the first paper about the architecture of such system. Our solution is developed based on the PINED-RQ family [33, 34]. This choice is motivated by the fact that this family can achieve both fast range queries and provable security guarantees while the secure index requires small space. More specifically, we re-design the architecture of PINED-RQ++ [34] in order to make it fully distributed. That is, we attempt to distributively process all heavy tasks (e.g., parsing and encrypting data) on a

set of *shared-nothing* machines[1]. By relying on such architecture, the system can easily scale. Additionally, we introduce a data representation and an asynchronous publishing method to decrease throughput degradation as much as possible. By precisely coordinating all of them together, our framework can enable intensive ingestion throughput. Experimental results show that as compared to (non-)parallel PINED-RQ++ [34], the throughput is improved by (~43×) ~5.6× respectively (NASA dataset [1]). Furthermore, we also present a new noise management mechanism to cope with strong online attackers having background knowledge about the time distribution of incoming data. This method also improves the practicality of FRESQUE since it does not require a pre-defined timestamp distribution as in PINED-RQ++ (see Section 4.1). In this study, we develop **FRESQUE**, a scalable ingestion **fr**am**e**work for **s**ecure range **que**ry processing on cloud, including the following contributions.

(1) We thoroughly analyze the architecture of the PINED-RQ family [33, 34], and point out obstacles that prevent the existing architecture from achieving a high ingestion throughput. Also, we propose an approach to cope with offline and online attackers while minimizing the required knowledge of the collector.
(2) We design an ingestion architecture that enables to distribute all heavy jobs to multiple workers (computing nodes) of a cluster. Besides, we present and integrate a data representation and an asynchronous publishing method to this architecture, mitigating throughput degradation.
(3) We extensively evaluate FRESQUE on real-world datasets to demonstrate its scalability. Particularly, the throughput of FRESQUE is about 43× higher than that of PINED-RQ++ and being at least one order of magnitude higher than that of other efficient solutions such as [6, 30, 31].
(4) We formally analyze the security of FRESQUE.

The paper is structured as follows. In Section 2, we briefly introduce the problem statement. Section 3 reviews the related work. We analyze the architecture of the PINED-RQ family in Section 4. We then describe our framework in Section 5. Section 6 gives security analyses while Section 7 presents our experimental results. We discuss an application of our solution in Section 8. Section 9 concludes the paper and gives future work.

2 PROBLEM STATEMENT

We assume that data sources produce a set of records where all records have the same number of attributes. These records are immediately sent to the collector. The dataset stored at the collector is a relation $D(A_1, \ldots, A_n)$, where A_i is an attribute. Queries are non-aggregate one-dimensional range queries. A query Q is evaluated over the attribute A_q of D, which contains numerical values. Periodically, the collector pre-processes the dataset, e.g., building a secure index over the dataset and encrypting it, prior to sending it to the cloud.
In this study, we concentrate on the scalability dimension of the system in terms of ingestion throughput. This metric measures how many records a system is able to consume within a time period. The target solution should meet additional requirements, namely formal security guarantees, supporting updates, and incurring practical storage overhead.

[1]We take the shared-nothing architecture into account since it is highly scalable.

2.1 Threat model

We consider the *honest-but-curious* model [15]. In this model, an attacker examines data stored on the cloud to glean sensitive information, but follows the protocol as specified and does not change the datasets or query results. Also, we consider three types of attackers : (1) the *offline* attacker is able to access a copy of the encrypted datasets and secure indices (e.g., by stealing the hard drives), (2) the *online* attacker is able to observe any information available at the cloud or being exchanged between the cloud and the trusted components, and (3) the *informed online* attacker is an online attacker that further has background knowledge about the data distribution of the incoming time of real data.

2.2 Security

2.2.1 Differential privacy. The ϵ-differential privacy model [11] requires that any possible individual record can only have a limited impact on the output distribution of an ϵ-differentially private function. This model considers a very strong adversary that is not computationally-bounded (information-theoretic guarantees). Definition 1 gives a formal definition.
Definition 1 (ϵ-differential privacy [11, 13]): A randomized mechanism $\mathcal{M}$ satisfies ϵ-differential privacy, if for any set $O \in Range(\mathcal{M})$, and any datasets $\mathcal{D}$ and $\mathcal{D}'$ s.t. $\mathcal{D}$ is $\mathcal{D}'$ with one record more or one record less,

$$Pr[\mathcal{M}(\mathcal{D}) = O] \leq e^{\epsilon} \cdot Pr[\mathcal{M}(\mathcal{D}') = O]$$

where ϵ represents the privacy level. A smaller ϵ means stronger privacy level. The Laplace mechanism is the most common method to achieve ϵ-differential privacy.
Laplace Mechanism [12]: Let $\mathcal{D}$ and $\mathcal{D}'$ be two datasets such that $\mathcal{D}$ is $\mathcal{D}'$ with one record more or one record less. Let $Lap(\beta)$ be a random variable that has a Laplace distribution with the probability density function $pdf(x, \beta) = \frac{1}{2\beta} e^{-|x|/\beta}$. Let f be a real-valued function, the Laplace mechanism adds $Lap(max \parallel f(\mathcal{D}) - f(\mathcal{D}') \parallel_1 / \epsilon)$ to the output of f, where $\epsilon > 0$.
Theorem 1 (Sequential Composition [27]): Let $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_r$ denote a set of mechanisms and each $\mathcal{M}_i$ gives ϵ_i-differential privacy. Let $\mathcal{M}$ be another mechanism that executes the sequence of $\mathcal{M}_1(\mathcal{D}), \mathcal{M}_2(\mathcal{D}), \ldots, \mathcal{M}_r(\mathcal{D})$. Then $\mathcal{M}$ satisfies $(\sum_{i=1}^{r} \epsilon_i)$-differential privacy.

2.2.2 Semantic Security. Loosely speaking, a cryptosystem is semantically secure if it is infeasible for a computationally-bounded adversary, i.e., a probabilistic polynomial algorithm, to derive significant information about plaintext from its ciphertext and any auxiliary information, e.g., obtained from external sources. Today, AES (in CBC mode) is the common instance of efficient private key encryption schemes satisfying semantic security.
Definition 2 (Semantic security [15]): A private key encryption algorithm E_χ, where χ is the secret key, is semantically secure if for every probabilistic polynomial time algorithm A there exists a probabilistic polynomial time algorithm A' such that for every input dataset $\mathcal{D}$, every auxiliary background knowledge $\zeta \in \{0,1\}^*$, every polynomially bounded function $g : \{0,1\}^* \rightarrow \{0,1\}^*$, every polynomial $p(\cdot)$, every sufficiently large $n \in N$, it holds that:

$$Pr[A_n(E_\chi(\mathcal{D}), |\mathcal{D}|, \zeta) = g(\mathcal{D})] < Pr[A'_n(|\mathcal{D}|, \zeta) = g(\mathcal{D})] + \frac{1}{p(n)}$$

2.2.3 Unified privacy model. Sahin *et al.* [33] is based on a probabilistic relaxation of a variant of differential privacy that considers computationally bounded adversaries [28] and that considers the cryptographically-negligible leaks due to the use of efficient real-world encryption schemes, i.e., AES in CBC mode. Definition 3 is a simplification of ϵ_n-SIM-CDP, the simulation-based computational differential privacy model proposed in [28].

Definition 3 (ϵ_n-SIM-CDP privacy [28]): The randomized function f_n provides ϵ_n-SIM-CDP if there exists a function F_n that satisfies ϵ_n-differential-privacy and a polynomial $p(\cdot)$, such that for every input dataset $\mathcal{D}$, every probabilistic polynomial time adversary A, every auxiliary background knowledge $\zeta \in \{0,1\}^*$, and every sufficiently large $n \in N$, it holds that:

$$Pr[A_n(f_n(\mathcal{D},\zeta)) = 1] - Pr[A_n(F_n(\mathcal{D},\zeta)) = 1] \leq \frac{1}{p(n)}$$

Definition 4 ((ϵ,δ)-Probabilistic-SIM-CDP [33]): A randomized function f_n satisfies (ϵ,δ)-Probabilistic-SIM-CDP, if it provides ϵ_n-SIM-CDP to each individual with probability greater than or equal to δ, where $\delta \in [0,1]$.

3 RELATED WORK

In this section, we briefly review prior schemes with respect to the target requirements. Table 1 gives the corresponding summary.

Table 1: Prior schemes with respect to target requirements

Scheme	Formal security guarantees	Update support	Low latency	Small storage overhead
HVE [8, 36]	✓	✓		
Bucketization [17, 19, 20]		✓	✓	✓
OPE [5–7, 26, 31]		✓	✓	✓
PBtree [24]	✓		✓	
IBtree [23]	✓	✓	✓	
ArxRange [30]		✓	✓	
Demertzis et al. [10]	✓	✓	✓	
PINED-RQ family [33, 34]	✓	✓	✓	✓

Hidden vector encryption (HVE) methods [8, 36] employ asymmetric cryptography to conceal attributes in an encrypted vector. A range predicate is privately evaluated on such vector. However, these schemes incur prohibitive computation and storage costs.

Bucketization approaches [17, 19, 20] partition an attribute domain into a finite number of buckets. Each bucket is then assigned by a random tag (bucket-id). When the client sends a range query to the server, the buckets that intersect the query are determined by using the index tag stored at the client. All contents of the intersecting buckets are finally returned to the client. These approaches lack formal privacy guarantee.

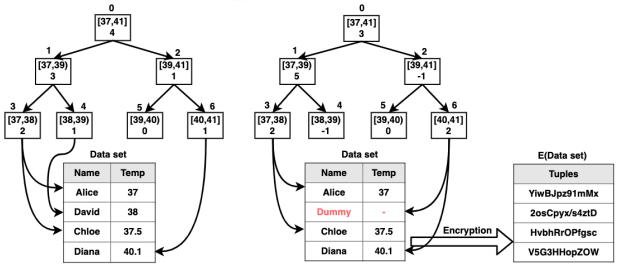
Order-preserving encryption schemes (OPE) [5–7, 26, 31] transform plaintexts into ciphertexts so that the relative order of their plaintexts is preserved. This property enables to efficiently execute range predicate evaluation on encrypted data. Unfortunately, OPE schemes disclose the underlying data distribution, and hence they are vulnerable to statistical attacks.

Recently, a few works have taken indexing solutions into account such as [10, 23, 24, 30, 33]. These schemes seek to maintain a secure index over outsourced data for fast range query processing. However, most of them [10, 23, 24, 30] suffer from prohibitive storage overhead. On the other hand, Sahin et al. [33] introduce an approach constructing *clear* secure indexes, called PINED-RQ indexes, for serving efficient range queries. The efficiency is enabled by the secure index while the security relies on computational differential privacy guarantees. Although PINED-RQ index is small in space, this scheme publishes data in batches, hence suffering from bottlenecks in the system as data arrives at high speed. Tran et al. [34] later introduce an index template-based approach, PINED-RQ++, to handle this limitation. Similar to prior schemes, high ingestion throughput still remains challenging for these solutions.

As shown in Table 1, the PINED-RQ family outperforms its counterparts with regard to the target requirements. To achieve our goal, we thus develop our solution based on this family. Particularly, we address the drawback of modest throughput in PINED-RQ.

4 ANALYSIS OF THE PINED-RQ FAMILY

This section gives thorough analyses of the PINED-RQ family [33, 34] and points out their drawbacks. Based on them, we develop our solutions in the Section 5.



(a) Clear index **(b) Perturbed index**

Figure 2: Sample PINED-RQ index

4.1 Overview

Building index. There are two primary steps to build a secure PINED-RQ index.

Step 1 - Building an index. PINED-RQ is inspired from B+Trees. In PINED-RQ, the set of all nodes is defined as a histogram covering the domain of an indexed attribute (e.g., the participants' body temperature (Temp)), as illustrated in Figure 2a. Each leaf node has a count that represents the number of records falling within its interval. It also keeps pointers to those records. Likewise, the root and any internal node have a range and a count, combining the intervals and the counts of their children, respectively.

Step 2 - Perturbing an index. All counts in the index are independently perturbed by Laplace noise [12]. The noise may be positive or negative, thereby after this step, the count of a node may increase or decrease, respectively. As shown in Figure 2b, the count of node 4 changes from 1 to -1 while the count of node 6 changes from 1 to 2. Such changes consequently lead to inconsistencies between the noisy count of a leaf node and the number of pointers it holds. To address this issue, PINED-RQ adds dummy records to the dataset when a leaf node receives positive noise, otherwise, if a leaf node receives negative noise, real records are removed from the dataset. The records removed are then inserted into the corresponding *overflow array* which is a fixed-size array. This overflow array is filled with dummy records if it has free space. As illustrated in Figure 2b, the record (David) belonging to node 4 is removed from the dataset while one dummy record is added and linked to node 6. Lastly, the perturbed index, the encrypted dataset, and the overflow arrays are published to the cloud.

Query processing on indexed data. A range query will start from the root of an index. It then traverses the child of any node that has a non-negative count and intersects with the query range. This is repeated recursively until the leaves of the index are reached. At the leaves that overlap the query range, their records and overflow arrays are returned.

Building index with index template. To adapt PINED-RQ to

the context of high rate of incoming data, Tran et al. [34] have previously developed PINED-RQ++ that builds a secure index based on the notion of *index template*. In particular, the collector initially creates an index template and perturbs it by using Laplace noise. This means that the count of bins at first contains only noise. The real count of bins will be updated during a *publishing time interval*, which is defined as the period from when an index template is created to when it is published. During a publishing time interval, whenever a new record arrives, the index template is updated with the record. Next, the collector encrypts and forwards this record to the cloud. At the end of each publishing time interval, the updated index template is published, and the cloud associates it with earlier published records to produce a secure index.

To manage positive noise, being represented by dummy records, the collector can send dummy records to the cloud according to the actual distribution of the sending time of the real records. On the other hand, for negative noise, if a leaf node initially receives negative noise c, the collector moves the first c records (when they arrive) of that leaf node to the corresponding overflow array.

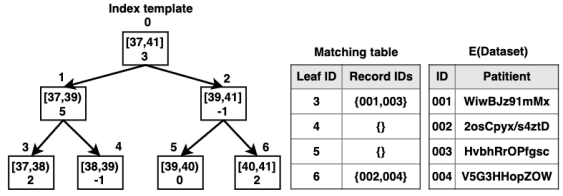


Figure 3: Perturbed index template and matching table (the corresponding PINED-RQ index is presented in Figure 2b)

To privately keep the relationships between published records and index template leaves, PINED-RQ++ utilizes a *matching table* (see Figure 3). In particular, a record is tagged by a random number instead of the real id of a leaf prior to being sent to the cloud. When the matching table is published at the end of each publishing time interval, the cloud uses it to reconstruct pointers between leaves and published data.

Generally, the workflow at the collector starts with initiating a new index template. When new data arrives, it sequentially passes a series of components, as depicted in Figure 4.

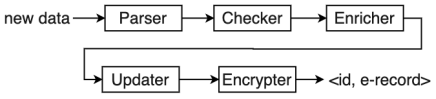


Figure 4: Workflow at the collector of PINED-RQ++

- **Parser** transforms incoming data into a pre-defined format.
- **Checker** buffers the parsed record at the collector as its indexed attribute belongs to a negative leaf. The index template is then updated. Otherwise, that record is forwarded to the next component.
- **Enricher** adds a random number (id) to the record.
- **Updater** updates the index template and matching table based on the record.
- **Encrypter** encrypts the record and gets e-record (encrypted record). The encrypter finally sends a pair of <id, e-record> to the cloud.

Parallel architecture of PINED-RQ++. Since the heavy workflow greatly degrades the throughput at the collector, Tran et al. [34] have introduced a parallel version of PINED-RQ++. Its overview architecture is depicted in Figure 5. Parallel PINED-RQ++ distributively processes heavy tasks on a set of independent

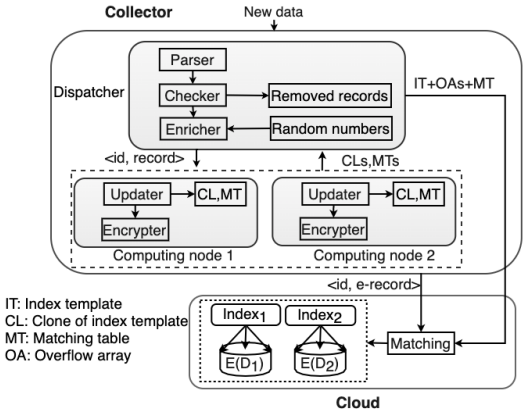


Figure 5: Architecture of parallel PINED-RQ++

machines (e.g., computing nodes), namely updater and encrypter. As a result, ingestion throughput exhibits a significant improvement. Nonetheless, several challenges still remain in this architecture. We now identify the obstacles that hinder PINED-RQ++ from achieving a scalable solution.

4.2 Limitations

Partial parallelism. In PINED-RQ++, the index template is proposed to temporarily store information that is necessary to build the secure index later. By doing it, the count variables of the index template are updated and referenced by the updater and the checker, respectively, during a publishing time interval. The index template is thus considered as a *shared* data structure, that does not run simultaneously in parallel PINED-RQ++. Furthermore, the checker depends on the parser for the checking operation. Hence, both parser and checker are organized to run in sequence at the collector (see Figure 5). Unfortunately, the parsing task usually takes time, especially in case of large record size. Thus, the parser mainly makes the ingestion throughput of the parallel PINED-RQ++ incredibly degraded. For instance, our experiments shows that the parsing task reduces the throughput of the collector by over 50% when we use NASA dataset [1].

Heavily updating index template. Since PINED-RQ++ uses the whole index template for updating and checking incoming data, there are some unnecessary overheads at the collector in terms of memory usage and computation. For example, an update always requires traversing from the root to leaves of the index template, having a complexity of $O(\log_k n)$, where n is the number of leaves and k is the branching factor. Likely, the checker faces the same complexity for checking a record whether it belongs to negative leaf or not. These tasks will take time to process records and diminish the ingestion ability of the system, especially when the domain of index template is huge. The situation is even worse when all tasks which reference the index template have to be processed sequentially.

Synchronous publication. Both PINED-RQ++ and its parallel version are designed to *synchronously* publish datasets to the cloud. In other words, they will start a new publication only if the current publication is sent to the cloud. This mechanism may create congestion in some circumstances. For instance, at the end of each publishing time interval, the collector needs to generate overflow arrays whose size primarily relies upon several configurable parameters, namely domain size, security level (e.g., ϵ), and bin interval. These parameters vary for different applications, thereby the size of overflow arrays will also change accordingly. As the size of overflow arrays is large, the collector

spends long time for generating overflow arrays, giving a heavy burden on the ingestion performance or even bottlenecks at this component.

5 INGESTION FRAMEWORK FOR SECURE RANGE QUERY (FRESQUE)

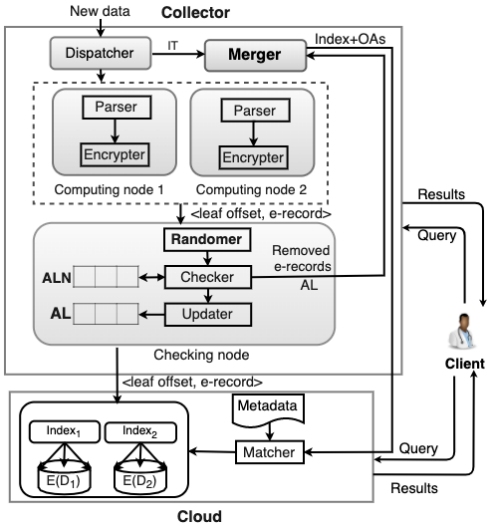


Figure 6: Architecture of FRESQUE

We first present below the key design features of FRESQUE that tackle the main limitations of PINED-RQ++ [34] (see Section 4.2). Then we describe how FRESQUE copes with the informed online attacker (see Section 2.1). Finally we present the complete architecture of FRESQUE.

5.1 Key Design Features

(a) A fully parallel architecture. As stated earlier, **partial parallelism** mainly causes low ingestion throughput in PINED-RQ++. To deal with that, we aim at making the collector fully distributed by parallelizing all heavy jobs (e.g., parser and encrypter) on a cluster of computing nodes.

The difficulty is that the checker, that resides between the parser and the encrypter in the workflow, cannot be parallelized since it relies not only on the parser but also on a shared data structure (e.g., index template). This means that the checker should be positioned after the parser and cannot be run in parallel. In fact, we can run the parser and the encrypter on multiple computing nodes while the checker runs sequentially at another node[2]. After incoming records are parsed at the instances of the parser, they are sent to the checker. These records are then checked by the checker before being sent back to the instances of the encrypter. Nevertheless, this approach would increase unnecessary communication overheads among components at the collector. Instead, we position the checker after the parser and the encrypter in the workflow, as illustrated in Figure 6. This approach allows to scale the intake ability of the collector without creating unnecessary transmission overheads. We then add additional information (e.g., leaf offset) to the ciphertext of the record so that the checker can know which leaf the record belongs to.

(b) Array representation of Leaves (AL). To address the problem of **heavily updating index template**, we replace the index template by an *array representation* of its leaves for the updating and checking operations in our new architecture. Such array representation is small to keeps in memory and accessing array elements requires constant time, $O(1)$. Particularly, the collector maintains two arrays of integers, one with noise (ALN) and the other without noise (AL). The former is used to check whether a record falls within a negative leaf or not while the latter is mainly used to count the number of real records passing the collector. Each element of AL/ALN represents the true count/noise of a leaf, respectively. The size of the two arrays is equal to the number of leaves of the index template. Note that the AL contains the true count of leaves while the IT only contains noise, thereby the two components are sufficient to compute the secure index.

To integrate such data representation into the new architecture, for a given value, the collector needs to know the *leaf offset* of the corresponding element in AL(N). Thanks to the strongly constrained shape of the PINED-RQ index, the leaf offset of a record can be easily obtained based on the configurable parameters of the system. Given parameters, namely domain min (d_{min}), domain max (d_{max}), bin interval (I_b), and an indexed attribute value (v), the leaf offset (O_v) of v can be achieved as follows.

$$O_v \leftarrow min(\lfloor (v - d_{min})/I_b \rfloor, \lfloor (d_{max} - d_{min})/I_b \rfloor - 1)$$

With such an approach, the checker is lightweight enough to avoid performance bottlenecks even if it runs sequentially.

(c) Asynchronously publishing mechanism. To address the issues of the **synchronous publishing** mechanism, we design our new architecture to *asynchronously* publish datasets. To this purpose, we add a new component to our architecture, named merger, that runs independently of the ingestion component (e.g., dispatcher), as depicted in Figure 6. The merger is only responsible for *publishing tasks*, namely building overflow arrays and combining a secure index from the AL and the IT (Index Templates). At the end of each publishing time interval, the publishing tasks are shifted to the merger, and a new publication is immediately initiated at the dispatcher. With this approach, while the dispatcher ingests data for a new publication, the merger performs the publishing tasks for the previous one. This eliminates the burden of the publishing tasks on the ingesting component and prevents potential bottlenecks at the collector. More importantly, the asynchronous publishing method allows the system to continuously consume incoming data with a very small latency for starting a new publication. Such property partially improves the ingestion throughput.

By using the asynchronous publication strategy, all components in FRESQUE, including the dispatcher and the merger, run independently. To ensure data consistency among publications, e.g., how a component determines to which publication a record belongs, we mark each publication with a unique *monotonic* number, named publication number.

5.2 Upgrading PINED-RQ++ for Coping with Informed Online Attackers

Information about the noise injected, e.g. dummy/removed records of a publication, may be disclosed to the informed online attacker if the order of the incoming data at the cloud is the same as the order of the incoming data at the collector. Indeed, since the informed online attacker has the time at which records (dummy or true) arrive at the cloud, his background knowledge on the time distribution of real data can enable him to distinguish (probabilistically) between incoming true records and incoming dummy records (positive noise), or to gain partial knowledge of the number of removed records (negative noise). First, the records removed by the checker at the collector (if they fall in the interval of a negative leaf - see Section 4) do not leave the collector before

[2]The encrypter can further benefit from hardware cryptographic modules.

the end of the publishing time interval. Their absence may reveal to the informed online attacker information about the values of negative noises. Second, the arrival of records at the cloud at unexpected times makes them more likely to be dummy record (information about positive noises). In PINED-RQ++, the collector releases dummy records according to the true distribution of the incoming time of real data for confusing the informed online attacker. This obviously requires to know the distribution in advance, which may be difficult to achieve in real-life applications. We now seek to design a new noise management method that mitigates privacy leaks against the informed online attacker and does not depend on any pre-defined distribution of the incoming time of real data.

To address this issue, we introduce a new component, called *randomer*, to our architecture (see Figure 6). It aims at *perturbing* the distribution of the incoming time of real data at the collector so that the insertion of dummy records or the deletion of real records are hidden from the adversary. The randomer consists of a *fixed-size buffer* and a *trigger* function. The former is used to *mix* incoming real and dummy records together while the latter is used to control the size of the buffer. In particular, all dummy records of a publication are first generated and are *uniformly at random* sent to the buffer of the randomer during a publishing time interval. For example, suppose a publication has 100 dummy records, then 100 time points are chosen uniformly at random over the current publishing time interval, and one dummy record is released at each time point. When a record (real/dummy) arrives at the randomer, it is buffered here. If the randomer buffer is full, then the randomer randomly picks one record in the buffer and releases it to the next component. Note that at any time point when one record is picked and released, it may be a real or dummy record. As a result, if a new record arrives the cloud at an improbable time point, the adversary cannot conclude with certainty whether it is dummy or not. Similarly, when the adversary does not see any record at an expected time point at the cloud, she/he cannot be sure it is removed or not due to the uncertainty caused by the randomer. The leakage caused by dealing naively with positive or negative noise (i.e., dummy records or removed true records) is thus addressed by the randomer. Moreover, FRESQUE does not require knowing in advance any data distribution.

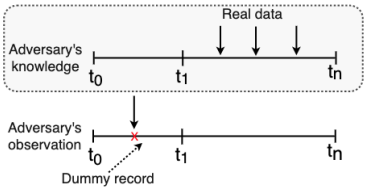


Figure 7: Randomer : Possible Issue of a Tiny Buffer

Challenges of randomer. One of the challenges of using randomer is how to choose a right size for its buffer. Intuitively, a large buffer gives high security. However, if the chosen size is too big, the system may confront bottlenecks at collector, particularly at the checking node. Otherwise, a tiny buffer may result above leak (the extreme case being a buffer of size 1). As an example depicted in Figure 7, we first assume that no real data is present during the period between t_0 and t_1 (the publishing time interval is $[t_0, t_n]$) and the size of the buffer is much smaller than the total number of dummy records. Since all dummy records are randomly released over a publishing time interval, it may happen that the buffer is full of dummy records, e.g., during the period $[t_0, t_1]$. The trigger function is thus activated and dummy records in the buffer are released before t_1. These dummy records will be recognized by the adversary who has prior knowledge about real data distribution. Fortunately, such situation only happens as the randomer buffer is much smaller than the total number of dummy records. Otherwise, if the randomer buffer is large enough, no record will appear at the cloud in that period. Therefore, the buffer size must be chosen to be sufficiently larger than the total number of dummy records of the publication.

A straightforward solution is to determine the buffer size by multiplying the actual number of the dummy records of a publication by several times. However, since we will publish the whole buffer at the end of the publishing time interval, the adversary may infer the size of the buffer, and hence the actual number of dummy records can be leaked. So the method of determining buffer size must (*) not depend on the real number of dummy records and (**) being larger than the number of the dummy records of a publication.

Note that since dummy records are generated due to the Laplace noise, the number of dummy records varies with each publication. It is thus difficult to choose a right capacity for the randomer buffer while meeting both (*) and (**). Fortunately, the noise in FRESQUE is sampled from the Laplace distribution, we can then choose buffer size based on the inverse CDF of the Laplace distribution with a very high probability, δ'. Intuitively, this approach gives an upper bound on the number of dummy records. Given a set of m leaves, denoted $L = \{l_1, ..., l_m\}$, we probabilistically compute the maximum number of dummy records of leaf l_i based on the inverse CDF of the Laplace distribution, considered as s_i. Then, $T = \sum_{i=1}^{m} s_i$ is viewed as the *maximum* number of dummy records of an index. To guarantee the buffer size is larger than T, we multiply it by a configurable coefficient, α. To ensure the buffer size is larger than the total number of dummy records, we suggest to set $\alpha \geq 2$. Then, the buffer size, S, of the randomer is: $S = \sum_{i=1}^{m} s_i \times \alpha$ (or $S = T \times \alpha$), where $\alpha \geq 2$.

Finally, the position of the randomer within the architecture of the collector is important as well: it must be put before the checker and the updater (so that it processes all records, including the removed ones) and after the parser and the encrypter (for obvious latency reasons).

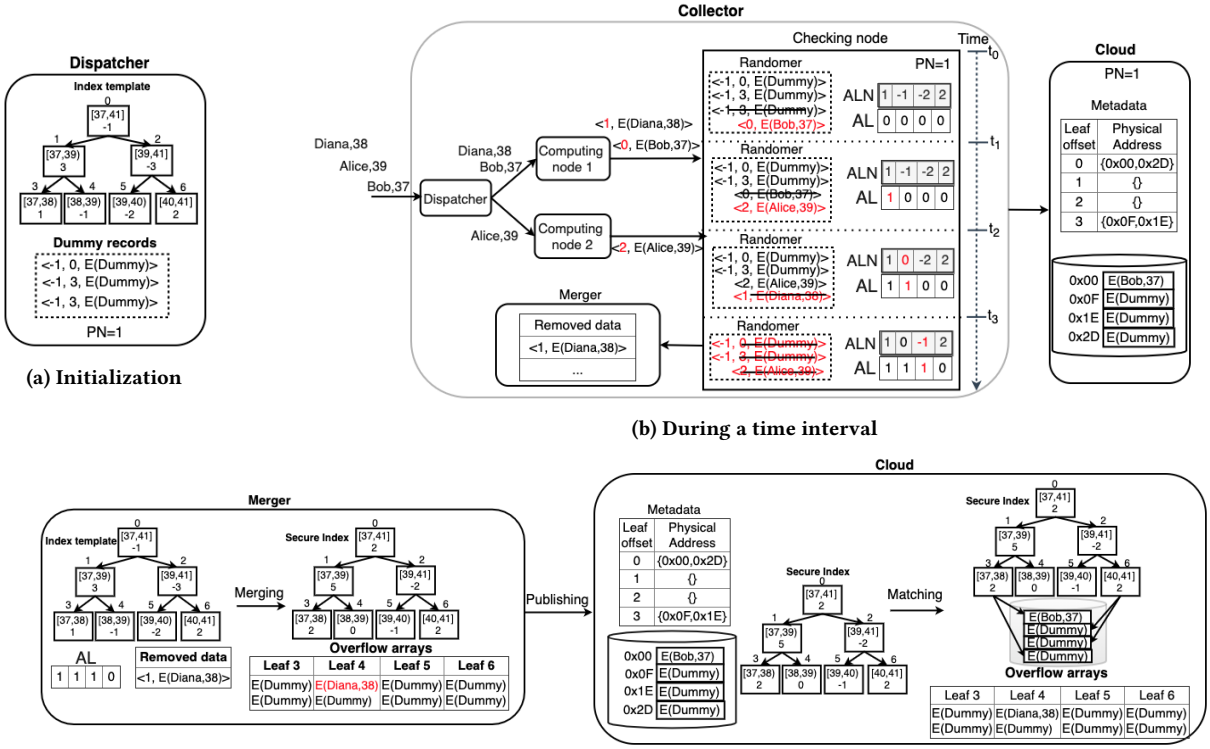
5.3 Architecture of FRESQUE

Following the key design features in Section 5.1, we now detail our ingestion framework to support efficient range query processing over encrypted data. Especially, we describe the orchestration of different components in this architecture.

(a) Ingestion life cycle. The collector of FRESQUE runs on a small cluster of commodity machines (see Figure 6). At the collector, one (and only one) node runs the Dispatcher (D) and all worker nodes in the cluster run a Computing Node (CN) while the randomer, the checker, and the updater run on the same Checking node (C).

When new records arrive to the dispatcher, they are immediately sent to the computing nodes according to a round robbin approach. This approach is used for the sake of load balancing. The computing node first pre-processes incoming data to get pairs of <leaf offset, e-record>. These pairs are then sent to the checking node. After being randomized and checked at the checking node, such pairs are forwarded to the cloud or the merger. Note that the dispatcher, the computing node, the merger, and the checking node can coexist on the same node.

(b) Instantiation of FRESQUE. In order to demonstrate how data



(a) Initialization

(b) During a time interval

(c) At publishing time

Figure 8: An example demonstrating how FRESQUE processes incoming data by using two computing nodes. Assume that the size of randomer buffer is 4 pairs of e-record.

is processed at the collector and transported to the cloud, Figure 6 shows the composition of FRESQUE running on five nodes at the collector and Figure 8 gives a running example.

Dispatcher (D): At the beginning of a publishing time interval, the dispatcher initiates an Index Template (IT), dummy records, and a Publication Number (PN), as illustrated in Figure 8a. The dispatcher then sends the IT, PN and all dummy records to the checking node. During a publishing time interval, whenever the dispatcher receives new data from data sources, it distributes the data to the computing nodes in a round-robin fashion. As an example shown in Figure 8b, there are three records, (Bob,37), (Alice,39), (Diana,38), arriving in order at the dispatcher. These records are then distributed to the two computing nodes. At the end of each publishing time interval, the dispatcher sends a *publishing* message to all available computing nodes and to the checker. By using the asynchronous publishing method, a new publication is immediately started after a *publishing* message is sent instead of waiting for the publishing tasks to be done. This allows the system to continuously ingest arrivals. By completely removing heavy jobs (e.g., parsing, encrypting, and checking) from the dispatcher and using the asynchronous publishing method, throughput ingestion is maximized at this component.

Computing Node (CN): During a publishing time interval, when new data comes, the computing node parses the raw data into a record, calculates the leaf offset, and encrypts it. Then, a pair of <leaf offset, e-record> is transferred to the checking node. As showed in Figure 8b, after passing the two computing nodes, three records are now parsed, encrypted, and associated with the corresponding leaf offsets, 0, 2, 1, respectively. When the computing node receives a *publishing* message from the dispatcher, it waits for a *done* message from the checking node. Notably, during the meantime, all incoming data will be processed and stored in local in-memory buffers at the computing nodes. By doing it, the delay of performing heavy tasks on buffered data is reduced when a new publication is started.

As mentioned earlier, the parser and the encrypter mainly cause throughput degradation in the system. With the parallel approach, the degradation is reduced significantly and only relies on the number of the computing nodes used. Interestingly, this approach not only allows to easily scale the throughput up, but also shortens the publishing time at the collector. For instance, PINED-RQ++ has to sequentially encrypt removed records and insert them into overflow arrays at the end of each publishing time interval, whereas they are now encrypted in a parallel manner by a set of networked machines during that period. As a result, at the end of each publishing time interval, the collector only randomly inserts removed encrypted records into the corresponding overflow arrays before transferring them to the cloud, reducing the publishing time in FRESQUE.

Checking node (C): At the beginning of a publishing time interval, the checking node receives Index Template (IT) and Publication Number (PN) from the dispatcher. It first initiates the corresponding AL and ALN (see Figure 8b). The checking node then forwards the IT to the merger while the PN is sent to the cloud. During a publishing time interval, when a pair of <leaf offset, e-record> arrives, it stores that pair in the buffer of the randomer. If the buffer is full, one of them is randomly picked and passed to the checker. Next, the checker gets its leaf offset (e.g., i) from the selected pair. If the i^{th} element of ALN is less than zero, the checker increases the value of the i^{th} element of both ALN and AL by one, and then sends that pair to the merger

as removed. Otherwise, that pair is sent to the updater, and only the value of the i^{th} element of AL is increased by one. Finally, that pair is sent to the cloud. As the example presented in Figure 8b, when the pair <0, (Bob,37)> comes to the checking node at timestamp t_0, it is inserted into the randomer's buffer. When this pair is released at timestamp t_1, the 0^{th} element of AL is increased by one since the 0^{th} element of ALN is positive. Otherwise, at timestamp t_2, when the pair <1, (Diana,38)> is considered, since it belongs to a negative element of ALN, the 1^{st} element of AL and ALN are both increased by one and this pair is then sent to the merger.

When the checking node receives *publishing* messages from all available computing nodes, it will send the updated AL to the merger (see Figure 8c). We emphasize that the condition of receiving publishing messages from all computing nodes needs to be guaranteed so that the consistency of publications is achieved. In other words, it makes sure that all (dummy/real) data of the current publication, that are sent by the dispatcher, are received by the checking node. The randomer buffer is then shuffled and published to the cloud. Finally, the checking node sends a *done* message back to the computing nodes.

It is worth noting that the checker and the updater will ignore the dummy records when they pass the checking node. This means that the counts of AL and ALN are independent of such dummy data. To achieve it, the checker and the updater need to perceive which incoming record is dummy in order to ignore it during the updating process. Nonetheless, the difficulty is that they all become ciphertexts after being encrypting by the computing nodes. To address it, we add to dummy records a *special* flag (e.g., -1) to distinguish them from real data. This straightforward technique allows the checker and the updater to know which record is dummy or real. As shown in Figure 8b, at timestamp t_0, a dummy pair is released by the checking node, and does not lead to any update on AL and ALN.

Even if all tasks at the checking node are designed to run sequentially such as the checker and the updater, they do not have much impact on the ingestion throughput at the collector. Moreover, thanks to the array representation, our architecture diminishes the complexity of the updating and checking tasks from $O(\log_k n)$ to $O(1)$, and hence shortening the delay of processing a record and boosting the consumption throughput.

Merger (M): At the beginning of each publishing time interval, the merger receives IT and PN from the checking node, then keeps them in memory. During a publishing time interval, the merger may receive removed records from the checker. Whenever the merger receives the updated AL, it triggers a new merging job that performs publishing tasks, e.g., combining IT and AL to achieve the complete secure index, generating overflow arrays (OAs) to conceal the removed records. Finally, the merger sends them to the cloud with the corresponding PN, as shown in Figure 8c.

Cloud: When the cloud receives a new PN from the checking node, it creates a new file for storing the incoming data. However, when its secure index is published by the merger, the published data will be read from the file on disk for a matching process and finally written back to disk again. Such approach gives rise to high I/O overhead. Instead, we keep small information about the published data, e.g., *metadata*, that is used for the matching process. Specifically, when a pair of <leaf offset, e-record> arrives, the cloud writes the e-record to disk, gets its physical address, and caches a pair of <leaf offset, physical location> in memory. To boost the matching process, we organize *metadata* in the form of <leaf offset, list of physical locations>, as demonstrated in Figure 8b. Such *metadata* is relatively small and independent of the size of e-records. When a publication comes from the merger, the matching process immediately associates the physical address of e-records with leaves based on the cached metadata. The metadata is finally destroyed (see Figure 8c).

(c) Query processing. In FRESQUE, when a query comes at the cloud, it is evaluated on both indexed and unindexed data. With regard to indexed data, the query processing strategy is applied as in Section 4.1. Meanwhile, unindexed data are processed one by one based on the query range. The (removed) records have a range overlapping the query range at the cloud, the randomer, and the merger are returned to the client.

6 SECURITY ANALYSIS

We develop FRESQUE that builds a PINED-RQ index [33] during a publishing time interval. With such an approach, at the end of each publishing time interval, all parts of the index (e.g., IT, AL, and removed e-records) are combined at the merger to get a secure index and overflow arrays. In other words, this process only occurs at the trusted collector, and hence FRESQUE apparently inherits the privacy protection level of the PINED-RQ index and trivially satisfies (ϵ, δ)-Probabilistic-SIM-CDP against offline attackers and (simple) online attackers.

The main difference between FRESQUE and PINED-RQ with respect to the index creation function is that FRESQUE publishes encrypted records immediately. Informed online attackers may be able to gain information about positive or negative noises based on the expected time distribution of incoming real records. However, thanks to the randomer, this leakage is mitigated. Theorem 2 claims the security of FRESQUE.

Theorem 2 (Security of FRESQUE*):* The index creation function of FRESQUE satisfies (ϵ, δ)-Probabilistic-SIM-CDP [33] against offline attackers and (simple) online attackers, and mitigates the information leak against informed online attackers.

Proof. *(Sketch)* We only consider informed online attackers because the two other attackers are trivial.

Considering dummy records (information leak about positive noises). First, we consider the case where a record arrives at the cloud at the time point at which there is real data. Since real/dummy records are randomly mixed together before being released, the adversary is unable to distinguish dummy records from real ones and does not obtain additional useful information about the values of the positive noises.

Second, we consider the case where a record arrives at the cloud at an unlikely time point at which there is no real data. This situation happens when a dummy record is inserted into a full randomer buffer. This means that with high probability the randomer buffer contains both real and dummy records. When receiving a record recently picked from the buffer, the cloud is thus unable to distinguish dummy records from real ones and does not learn additional information about the values of the positive noises.

Third, we consider the case where the checking node sends the full randomer buffer to the cloud at the end of a publishing time interval. Dummy records are mixed with real ones at the randomer during a publishing time interval. Additionally, the ratio between real and dummy data at any time point is hidden from the adversary (see Section 5.2), hence the adversary does not

obtain additional useful information about the values of the positive noises from this case.

Fourth, we consider the case where the randomer contains only dummy records and no real ones. Note that such situation only occurs if all dummy records are released before real data arrives at the collector. If the buffer of the randomer were allowed to be smaller than or equal to the total number of dummy records, a dummy record would be picked with certainty and released to the cloud. As a result, the adversary would learn with certainty that it is dummy. However, FRESQUE requires that the buffer of the randomer is chosen to be much larger than the total number of dummy records of a publication, with (tunable) high probability (see Section 5.2). This makes this case highly improbable.

Considering removed records (information leak about negative noises). Since dummy records will not be deleted by the checking node, we only focus on real records. Recall that the decision to remove a real record is taken by the checker, after the randomer, and that removed records are buffered by the merger in order to be published within the overflow arrays at the end of the publishing time interval. Without randomer, the artificial removal of records due to negative noise may impact the number of records sent to the cloud and thus, slightly, the time distribution of the records sent to the cloud. However, first, the additional dummy records mitigate the decrease in the number of records overall (recall that the Laplace distribution used for generating the noises is symmetric around 0), and second, with the randomer, the delay introduced by the randomer buffer in releasing both dummy and real records also impacts the time distribution of records to the cloud, similar to the removal of true records, which mitigates the information leak about the values of the negative noises. □

Comparison with PINED-RQ [33]. The highest security of FRESQUE is achieved when the coefficient α is chosen so that the randomer buffer can contain the whole dataset and all dummy records. In that case, at the end of each publishing time interval, the randomer shuffles and sends the buffer to the cloud along with a secure index and overflow arrays. It is easy to see that this is exactly the publishing process of PINED-RQ. Thus, in that case, FRESQUE has the same level of privacy against informed online attackers as PINED-RQ, and thus also satisfies exactly (ϵ, δ)-Probabilistic-SIM-CDP against all attackers.

7 EVALUATION

We evaluate FRESQUE against PINED-RQ++ due to its outperformance compared to other prior schemes (see Table 1). We mainly focus on the metrics contributing to the scalability of the system, namely ingestion throughput and publishing latency at the collector as well as at the cloud.

7.1 Benchmark Environment

Table 2: Experimental environment

Component	CPU (2.4 GHz)	Memory (GB)	Disk (GB)
Dispatcher	4	8	80
Merger	4	8	80
Checking node	4	8	80
Computing node	2	2	20
Data source	4	16	80
Cloud	16	64	160

We implemented FRESQUE in Java 1.8.0[3]. Data was encrypted by the Java package (javax.crypto). We ran our experiments on the Galactica platform [4] and organized FRESQUE as a cluster of 17 nodes, including 12 computing nodes, running on Ubuntu 14.04.4 LTS. Each node was used to run one component of FRESQUE. The configurations of nodes are detailed in Table 2. The TCP socket was used for exchanging data among the components of FRESQUE.

[3]The code is available at https://gitlab.inria.fr/vtran/fresque.git.

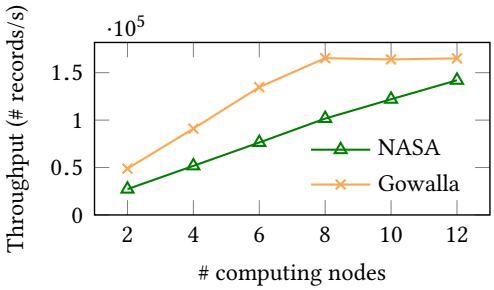


Figure 9: Ingestion throughput of FRESQUE

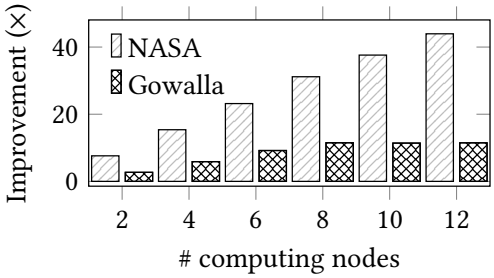


Figure 10: FRESQUE's improvement compared to PINED-RQ++

We evaluate our solution on two real datasets: NASA log [1] (1, 569, 898 records, five attributes) and Gowalla [22] (6, 442, 892 records, three attributes). We use the reply byte and check-in time as indexed attributes, respectively. Based on these datasets, the domain of the reply byte is divided into 3421 bins and each bin interval represents 1 KB. Meanwhile, the domain of the check-in time is 626 bins and each bin interval implies one hour. The fanout is set to 16. We use a publishing time interval of 60 seconds and incoming data rate is 200k records per second. The initial privacy budget and coefficient is set to 1 and 2, respectively, for all experiments unless otherwise stated. Both δ and δ' are set to 99% and every experiment was run over ten minutes. Then, we present the averaged results of ten publications in Section 7.2.

7.2 Results

Ingestion throughput. We first present the ingestion throughput of FRESQUE with a varied number of computing nodes. Then we compare its ingestion throughput to those of the (non-)parallel PINED-RQ++. The results in Figure 9 show that the throughput of FRESQUE significantly increases as the number of computing nodes goes up. Especially, the highest throughput is reached at ~142k records/second (NASA) and at ~165k records/second (Gowalla) with 12 and 8 computing nodes respectively. As compared to ArxRange [30], one of the state-of-the-art solutions, FRESQUE reaches an ingestion throughput that is at least two orders of magnitude higher.

(a) Comparison with non-parallel PINED-RQ++. With the given settings, non-parallel PINED-RQ++ is able to ingest only 3,159 records/s in NASA and 13,223 records/s in Gowalla. Such ingestion throughputs are substantially lower than those of FRESQUE. The results in Figure 10 demonstrate the outperformance of

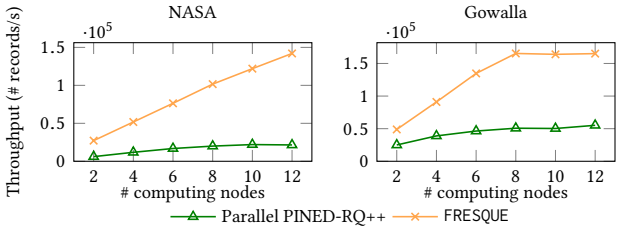


Figure 11: Comparison of ingestion throughput with parallel PINED-RQ++

FRESQUE compared to non-parallel PINED-RQ++. The enhancement goes up as the number of computing nodes grows. The highest improvement can be seen as the collector is configured as a 12-computing node cluster, and the ingestion throughput is improved by ~11× and ~43× in Gowalla and NASA dataset, respectively. Even if only two computing nodes are used, FRESQUE can achieve the improvement of 7.61× (NASA) and 2.69× (Gowalla). Compared to Gowalla, NASA always exhibits higher improvement with the same number of computing nodes. The major source of this gap comes from the fact that the record size and the domain of NASA record are larger than those of Gowalla. Based on such observation, we can conclude that FRESQUE would be more beneficial as datasets have larger size and/or domain.

(b) Comparison with parallel PINED-RQ++. The throughput of FRESQUE is always higher than that of parallel PINED-RQ++ as we vary the number of computing nodes at the collector, as shown in Figure 11. The setting of 12-computing node cluster gives the biggest gap, the throughput of FRESQUE is ~5.6× (NASA) and ~2.2× (Gowalla) better than that of parallel PINED-RQ++. Noted that since the throughput in FRESQUE reaches the peak as we use 8 computing nodes in Gowalla, the use of more computing nodes does not bring more benefit.

Throughput degradation. We measure the throughput degradation at the collector of the three prototypes. Such metric is obtained by comparing their maximum ingestion throughput with the maximum incoming throughput (without any processing on incoming data) at the collector. As shown in Figure 12, FRESQUE experiences the lowest throughput degradation among the three prototypes, with a reduction of at least ~3.9× (compared to parallel PINED-RQ++) in NASA, and at most ~7.9× (compared to PINED-RQ++) in Gowalla.

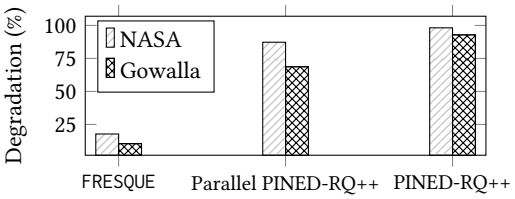


Figure 12: Throughput degradation at the collector

Publishing time. We now turn our attention to the publishing time metric, i.e., the time required to publish a dataset with FRESQUE and with parallel PINED-RQ++. Noted that FRESQUE consists of the three main components, namely the dispatcher, the checking node, and the merger which mainly decide the publishing time at the collector. We thus measure the delay of the three components separately. Additionally, we consider the time needed to perform a matching process at the cloud. This is because a long delay of this process might also lead to bottlenecks.

(a) Publishing time at the dispatcher. As shown in Figure 13, the time is always lower than 520ms with NASA and 200ms with

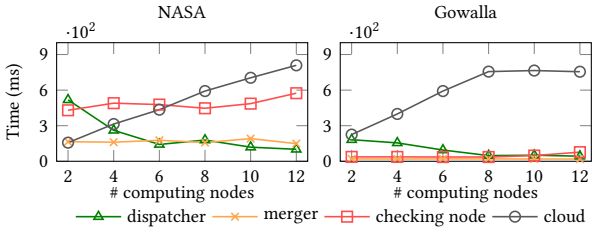


Figure 13: Publishing time in FRESQUE

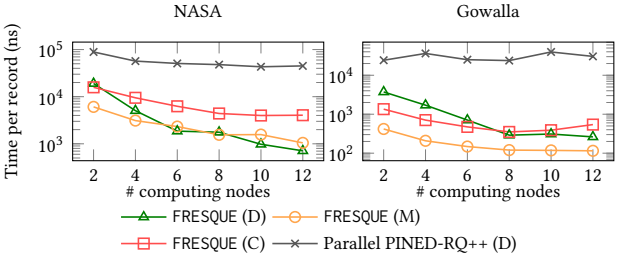


Figure 14: Comparison of publishing time at collector with parallel PINED-RQ++

Gowalla. The delay even gradually decreases as the number of computing nodes increases. In particular, the dispatcher takes only 101ms (NASA) and 19ms (Gowalla) for performing the publishing tasks in a 12-computing node cluster.

(b) Publishing time at the merger. The results in Figure 13 indicates that the time is virtually unchanged in the two datasets as their size changes. Specifically, the time with NASA fluctuates between 149ms and 191ms while that with Gowalla varies between 18ms and 20ms. Since the domain size of NASA (3421 bins) is larger than the one of Gowalla (626 bins), the NASA experiences a higher publishing time than that of the Gowalla dataset.

(c) Publishing time at the checking node. In this study, we attempt to design FRESQUE so that the checking node has a lightweight publishing job and has a reduced impact on the ingestion performance. In particular, the checking node only sends the buffer of the randomer to the cloud and the updated AL to the merger at the end of each publishing time interval. The results in Figure 13 show that the time is under 600ms with NASA and 80ms with Gowalla. It can be understood that the publishing time at the checking node is mainly represented by the time of sending the randomer buffer that varies according to the required level of security. A huge randomer buffer results in long publishing time at this component. Fortunately, since the computing nodes always process and cache incoming data during the meantime, the impact on the ingestion throughput is negligible. We will evaluate the randomer below in the latest part of this section.

(d) Matching time at the cloud. To show the efficiency of FRESQUE at the cloud side, we measure the time required to associate *metadata* (physical locations of records) with published index. As depicted in Figure 13, the time in FRESQUE goes up according to data size. Nonetheless, FRESQUE spends only 877ms and 837ms on matching the large dataset of 8.1M records (NASA) and 9.8M records (Gowalla), respectively. These performances come from the deletion of the matching table from FRESQUE's architecture.

(e) Comparison with parallel PINED-RQ++. We now compare publishing time at the collector between FRESQUE and parallel PINED-RQ++. Since the different numbers of computing nodes used result in different publication sizes, we consider the time is required to publish a record instead of a whole dataset. The results in Figure 14 show that parallel PINED-RQ++ (dispatcher) takes longer delay than FRESQUE (dispatcher, checking node, and merger) for

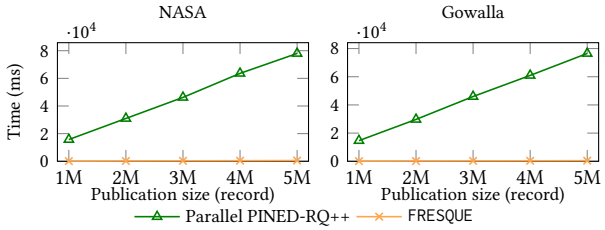


Figure 15: Comparison of matching time at cloud with parallel PINED-RQ++

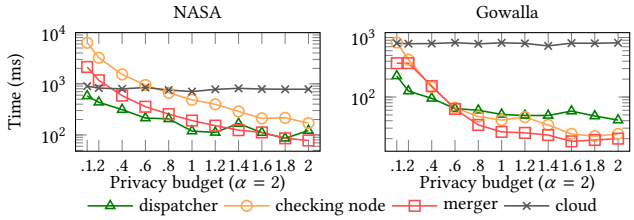


Figure 16: Publishing time with different privacy budgets

the two datasets used. Regarding the dispatcher, the publishing time of FRESQUE is at most ~62× and ~127× lower with NASA and Gowalla, respectively, compared to parallel PINED-RQ++.

Matching time. We also evaluate the matching time needed to process a publication between parallel PINED-RQ++ and FRESQUE. The results in Figure 15 show that the time of parallel PINED-RQ++ increases when publications are larger. For example, when a dataset of 5M records is used, the matching time in parallel PINED-RQ++ reaches ~78s (NASA) and ~76s (Gowalla). In contrast, FRESQUE constantly maintains a short time for processing a publication at the cloud, with a maximum is ~54ms (NASA) and ~43ms (Gowalla). The matching time of FRESQUE is at least two orders of magnitude shorter than that of parallel PINED-RQ++.

Impact of the randomer. For mitigating the information disclosed to the informed online attacker, the randomer maintains a local buffer for perturbing incoming data. A large buffer may introduce a bottleneck at the collector. We thus evaluate the impact of this component here. Indeed, the buffer size is mainly determined by two configurable parameters, namely privacy budget ϵ and coefficient α. Hence, we run various experiments with varied values of the two parameters to evaluate the impact of the randomer. We use a configuration of 10 computing nodes.

(a) Privacy budget ϵ. We now consider the impact of the randomer in terms of publishing time as we use different privacy budgets, ranging from 0.1 to 2.0, for a publication. In these experiments, we record the publishing time at the collector (dispatcher, checking node, and merger), and the matching time at the cloud. The results in Figure 16 show that the privacy budget influences the publishing time at the three components. Indeed, as a smaller privacy budget is used, their publishing time goes up. The highest increase is witnessed at the checking node, approximately 7s (NASA) and about 0.8s (Gowalla) for the budget of 0.1. Similarly, as the privacy budget declines, the size of overflow arrays and the number of dummy/removed records go up, causing a slight increase of the publishing time at the dispatcher and the merger.

(b) Coefficient α. We adjust the value of α to see the impact of randomer on publishing time at the checking node, the merger and the cloud. As expected, when we increase the value of α, the publishing time grows (see Figure 17). However, even if α is set to 20, the checking node only takes about 6s (NASA) and 0.8s (Gowalla). Also, the time does not change much at the dispatcher,

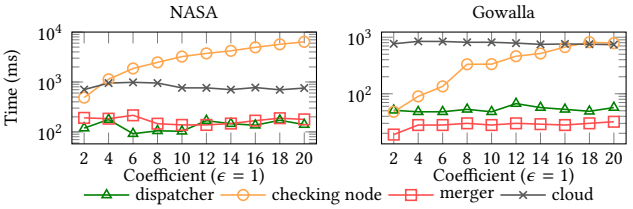
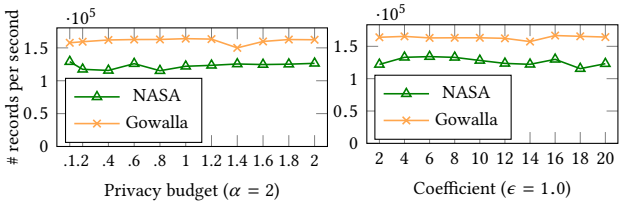


Figure 17: Publishing time with different coefficients



(a) Varying privacy budgets **(b) Varying coefficients**

Figure 18: Ingestion throughput of FRESQUE with randomer

the merger and the cloud.

(c) Impact of the randomer on ingestion throughput. We also consider the ingestion throughput at the collector as we vary the two parameters ϵ and α. Although the publishing time at the checking node goes up as we use smaller privacy budget and/or larger coefficient, the ingestion throughput at the collector is relatively stable. This is because while the checking node prepares publish the current dataset, including the sending of randomer to the cloud, incoming data of the new publication is still processed and buffered at the computing nodes. As it can be seen in Figure 18a and Figure 18b, the results show that the throughput with the NASA dataset fluctuates between ~115k records/s and ~134k records/s while that of Gowalla ranges from ~150k records/s to ~166k records/s.

8 DISCUSSION

We present a possible real-life application of FRESQUE based on the FluTracking use-case [3].

Flutracking is a web-based survey of influenza-like illness. This system weekly sends a link via email to all participants who will then submit required information via a web interface. The submitted data can be managed in a cloud and accessed by authorized users for analysis and prediction.

Although our description of FRESQUE focuses on the insertion of one record per individual, it is simple to extend our approach to the case of multiple records per individual. For example, in Flutracking [3], an individual can submit personal data several times to the database, at most once for a week. For such case, an important question is how to manage privacy budget over multiple insertions of the same individual.

In the targeted use case, it is unlikely to have multiple records of the same individual over a short period (e.g., weekly). Therefore, we can assume that a dataset of each period (e.g., week) is published with a secure index, and this publication consists of at most one record per individual. For each dataset, the system uses a portion of the total privacy budget ϵ_{total} for constructing a secure index. To determine how much budget is spent for a publication, an admin may necessarily determine how long the system needs secure indices for fast range query processing. ϵ_{total} is then divided according to the determined period. For instance, if the system must maintain indices for one year (52 weeks), then an admin can divide the total privacy budget ϵ_{total} into 52 equal

portions, $\epsilon_1, ..., \epsilon_{52}$, so that $\epsilon_{total} = \sum_{i=1}^{52} \epsilon_i$. Each of which is used to publish dataset of one week. Certainly, the system needs to make sure that an individual contributes at most one record per publication. Fortunately, thanks to the existing collecting method of the Flutracking, this work can simply be achieved. In particular, a unique link can be sent to all participants every Monday. The link is set to expired and a dataset is published before the next Monday. This ensures that a participant links to at most one record per publication.

9 CONCLUSION

This paper presents FRESQUE, a scalable ingestion framework for secure range query processing over encrypted data on clouds. We thoroughly analyze and identify the problems of the-state-of-the-art solutions related to the degradation of the ingestion throughput, with a special focus on PINED-RQ++. To address these drawbacks, we design a new architecture that is fully distributed at the collector. Additionally, we introduce a data representation as well as an asynchronous publication mechanism. All of them together allows FRESQUE to achieve intensive consumption throughput, reaching over 160K records/s. Moreover, we introduce and carefully integrate the randomer into our new architecture to improve the practicality and security of FRESQUE as compared to PINED-RQ++. We formally analyse the security guarantees of FRESQUE. Lastly, we discuss a potential application of FRESQUE based on a real-life example. Future works include coping with multiple records per individual and designing alternative indexes based on well-known highly concurrent data structures (e.g., Masstree [25] and ART [21]).

ACKNOWLEDGMENTS

This work was partially funded by LTC and Pôle d'Excellence Cyber.

REFERENCES

[1] NASA Log. http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html (1996).
[2] Flu Near You. https://flunearyou.org/ (2020).
[3] FluTracking.net. http://info.flutracking.net/ (2020).
[4] Galactica. https://galactica.isima.fr/index.html (2020).
[5] Agrawal R., Kiernan J., Srikant R., and Xu Y. Order Preserving Encryption for Numeric Data. In: SIGMOD '04, 563–574 (2004).
[6] Boldyreva A., Chenette N., Lee Y., and O'Neill A. Order-Preserving Symmetric Encryption. In: A. Joux, ed., Advances in Cryptology - EUROCRYPT 2009, 224–241. Springer Berlin Heidelberg, Berlin, Heidelberg (2009).
[7] Boldyreva A., Chenette N., and O'Neill A. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In: P. Rogaway, ed., Advances in Cryptology – CRYPTO 2011, 578–595 (2011).
[8] Boneh D. and Waters B. Conjunctive, Subset, and Range Queries on Encrypted Data. In: TCC'07, 535–554 (2007).
[9] Dalton C., Carlson S., Butler M., Cassano D., Clarke S., Fejsa J., and Durrheim D. Insights From Flutracking: Thirteen Tips to Growing a Web-Based Participatory Surveillance System. JMIR Public Health Surveill, 3(3):e48 (2017).
[10] Demertzis I., Papadopoulos S., Papapetrou O., Deligiannakis A., Garofalakis M., and Papamanthou C. Practical Private Range Search in Depth. ACM Trans. Database Syst., 43(1):2:1–2:52 (2018).
[11] Dwork C. Differential Privacy. In: ICALP'06, 1–12 (2006).
[12] Dwork C., McSherry F., Nissim K., and Smith A. Calibrating Noise to Sensitivity in Private Data Analysis. In: TCC'06, 265–284 (2006).
[13] Dwork C. and Roth A. The Algorithmic Foundations of Differential Privacy. Found. Trends Theor. Comput. Sci., 9(3-4):211–407 (2014).
[14] Elmore A.J., Das S., Agrawal D., and El Abbadi A. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In: SIGMOD '11, 301–312 (2011).
[15] Goldreich O. Foundations of Cryptography: Volume 2, Basic Applications. Cambridge University Press, New York, NY, USA (2004).
[16] Guerrisi C., Ecollan M., Souty C., Rossignol L., Turbelin C., Debin M., Goronflot T., Boëlle P.Y., Hanslik T., Colizza V., and Blanchon T. Factors associated with influenza-like-illness: a crowdsourced cohort study from 2012/13 to 2017/18. BMC Public Health, 19(1):879 (2019).
[17] Hacigümüş H., Iyer B., Li C., and Mehrotra S. Executing SQL over Encrypted Data in the Database-service-provider Model. In: SIGMOD '02, 216–227 (2002).
[18] Help-Net-Security. 52% of companies use cloud services that have experienced a breach. https://www.helpnetsecurity.com/2020/01/28/accessing-cloud-services/ (2020).
[19] Hore B., Mehrotra S., Canim M., and Kantarcioglu M. Secure Multidimensional Range Queries over Outsourced Data. The VLDB Journal, 21(3):333–358 (2012).
[20] Hore B., Mehrotra S., and Tsudik G. A Privacy-preserving Index for Range Queries. In: VLDB '04, 720–731 (2004).
[21] Leis V., Kemper A., and Neumann T. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In: ICDE '13, 38–49. IEEE Computer Society (2013).
[22] Leskovec J. and Krevl A. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data (2014).
[23] Li R. and Liu A.X. Adaptively Secure Conjunctive Query Processing over Encrypted Data for Cloud Computing. In: ICDE'17, 697–708 (2017).
[24] Li R., Liu A.X., Wang A.L., and Bruhadeshwar B. Fast Range Query Processing with Strong Privacy Protection for Cloud Computing. Proc. VLDB Endow., 7(14):1953–1964 (2014).
[25] Mao Y., Kohler E., and Morris R.T. Cache Craftiness for Fast Multicore Key-Value Storage. In: EuroSys '12, 183–196. ACM (2012).
[26] Mavroforakis C., Chenette N., O'Neill A., Kollios G., and Canetti R. Modular Order-Preserving Encryption, Revisited. In: SIGMOD '15, 763–777 (2015).
[27] McSherry F.D. Privacy Integrated Queries: An Extensible Platform for Privacy-preserving Data Analysis. In: SIGMOD '09, 19–30 (2009).
[28] Mironov I., Pandey O., Reingold O., and Vadhan S. Computational Differential Privacy. In: S. Halevi, ed., Advances in Cryptology - CRYPTO 2009, 126–142 (2009).
[29] Papadimitriou A., Bhagwan R., Chandran N., Ramjee R., Haeberlen A., Singh H., Modi A., and Badrinarayanan S. Big Data Analytics over Encrypted Datasets with Seabed. In: OSDI '16, 587–602 (2016).
[30] Poddar R., Boelter T., and Popa R.A. Arx: An Encrypted Database Using Semantically Secure Encryption. Proc. VLDB Endow., 12(11):1664–1678 (2019).
[31] Popa R.A., Li F.H., and Zeldovich N. An Ideal-Security Protocol for Order-Preserving Encoding. In: SP '13, 463–477 (2013).
[32] Popa R.A., Redfield C.M.S., Zeldovich N., and Balakrishnan H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In: SOSP '11, 85–100 (2011).
[33] Sahin C., Allard T., Akbarinia R., El Abbadi A., and Pacitti E. A Differentially Private Index for Range Query Processing in Clouds. In: ICDE '18, 857–868 (2018).
[34] Tran H.V., Allard T., D'Orazio L., and Abbadi A.E. Range Query Processing for Monitoring Applications over Untrustworthy Clouds. In: EDBT'19, 666–669 (2019).
[35] Tu S., Kaashoek M.F., Madden S., and Zeldovich N. Processing Analytical Queries over Encrypted Data. Proc. VLDB Endow., 6(5):289–300 (2013).
[36] Wen M., Lu R., Zhang K., Lei J., Liang X., and Shen X. PaRQ: A Privacy-Preserving Range Query Scheme Over Encrypted Metering Data for Smart Grid. IEEE Transactions on Emerging Topics in Computing, 1(1):178–191 (2013).