UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Van-Hoang TRAN

Range Query Processing over Untrustworthy Clouds

Traitement des Requêtes d'Intervalle sur des Nuages Non Fiables

Thèse présentée et soutenue à Rennes, le 04 Décembre 2020

Unité de recherche : IRISA

Thèse N° :

Rapporteurs avant soutenance :

Benjamin NGUYEN	Professeur, INSA Centre Val de Loire
Ladjel BELLATRECHE	Professeur, École Nationale Supérieure de Mécanique et d'Aérotechnique

Composition du Jury :

	Benjamin NGUYEN	Professeur, INSA Centre Val de Loire
	Caroline FONTAINE	Directrice de Recherche, CNRS
	David GROSS-AMBLARD	Professeur, Université de Rennes 1
	Ladjel BELLATRECHE	Professeur, École Nationale Supérieure de Mécanique et d'Aérotechnique
Dir. de thèse :	Laurent d'ORAZIO	Professeur, Université de Rennes 1
Co-encadrant :	Tristan ALLARD	Maître de Conférences, Université de Rennes 1

Author

Tran-Van Hoang was born in Tra Vinh, Vietnam. He obtained his engineer degree of software engineering from Can Tho University in 2011. Since then, he has worked as a lecturer at Can Tho University. In 2015, he received his master degree in computer science from Université de Bretagne Occidentale. He commenced his PhD in computer science at Université de Rennes 1 in 2017. His main research interests include database, distributed systems, data privacy, and modeling and simulation.

To my family

Acknowledgement

First of all, I would like to express my deepest gratitude and appreciation to my advisors, Prof. Laurent d'Orazio and Assoc. Prof. Tristan Allard, for providing continuous support, guidance, mentoring throughout my PhD. They first gave me a great opportunity to pursue my research career in France. They have trained me how to do research properly and they were so patient with me as I was progressing slowly. My advisors always kept me motivated with their excitement for all works that I made. Needless to say that I could not do well my thesis without my advisors. Also, I would like to give special thanks to Dr. Thuong-Cang Phan who has introduced me to my great advisors. Thank you very much for all!

I would thank Prof. Amr El Abbadi for his collaboration as well as his insightful comments during my PhD. I would also like to thank Caroline Fontaine, Senior Researcher, CNRS, and Julien Lallet, Research Engineer, Nokia, for accepting to be the CSID Committee Members of my thesis. Their insightful suggestions and comments have motivated me to improve and complete my research.

Also, I would like to thank William Guyot-Lénat for kindly supporting me so that I can smoothly run my experiments on the Galactica platform at LIMOS laboratory.

My PhD would not be possible without the financial supports from the Lannion Trégor Communauté and Pôle Excellence Cyber. I would like to acknowledge them for funding my research. Thank you all members of SHAMAN team for giving me a great working environment as well as kindly supports. In addition, thank you Can Tho University, especially College of Information & Communication Technology, my workplace in Vietnam, that facilitates me to complete my research abroad.

Last but not least, I would like to thank my wonderful family for always being there for me. I would like to thank my lovely wife, Thi-Lan-Huong Nguyen, for her great care and understanding. All of them have always believed in me and supported me endlessly. Thank you! You are the main reason I am where I am today.

Declaration

This dissertation has been completed by Tran-Van Hoang under the supervision of Professor Laurent D'Orazio and Assoc. Professor Tristan Allard and has not been submitted for any other degree or professional qualification. I declare that the work presented in this dissertation is entirely my own except where indicated by full references.

Abstract

Over the last years, cloud computing has increasingly become a standard for saving costs and enabling elasticity. While cloud providers expand their services, their clients are still concerned about the security of cloud technologies. This is mainly because security issues often lead to data breaches which demand significant efforts and extreme costs for correction. To address these issues, encryption is usually used to protect confidential data stored and processed on untrustworthy clouds. Encrypting outsourced data however mitigates the functionalities of applications since supporting fundamental functions on encrypted data is still limited, range queries for example.

Many studies have been done on range query processing over encrypted data in recent years. Nevertheless, none of prior schemes exhibits satisfactory performances for modern systems, that require not only low-latency responses, but also high scalability. In particular, most of existing solutions suffer from either inefficient range query processing or security issues. For example, they would take hundreds of seconds to evaluate a range query or disclose the privacy of plaintexts. Even if some can achieve both strong privacy protection and fast query processing, they are unable to provide scalable solutions. In other words, these schemes incur bottlenecks, prohibitive storage overhead, or even limited update operations.

In this dissertation, we aim at providing scalable solutions for secure range query processing while still maintaining high efficiency and strong privacy protection. Our contribution is many-fold: 1) We introduce an approach to support secure range query processing in the context where data arrive at a high speed. 2) We propose an intensive ingestion framework dedicated to secure range query processing on encrypted data. 3) We present a scalable scheme for private range query processing on outsourced data.

The first contribution is to avoid potential bottlenecks in the context where the system confronts a high rate of incoming data. Additionally, our approach preserves fast range query processing and strong privacy protection. To achieve this goal, we extend PINED-

RQ, one of the most efficient state-of-the-art solutions, and adapt this work to the target context. To this purpose, we introduce a notion of index template to PINED-RQ so that the system can consume incoming data on the fly. Besides, a noise management strategy and a complementary data structure are designed to prevent potential privacy leaks caused by such extension. We then develop a parallel version of this extension for improving the ingestion throughput. With practical assumptions, we prove that the privacy protection of our solution is as strong as that of PINED-RQ. The experimental results show that our proposal outperforms PINED-RQ in various metrics. For instance, the publishing time of a dataset of ∼0.5M records is reduced up to $\sim 35\times$ while the maximum data rate of the system experiences a reduction of up to $\sim 2.7\times$.

The second contribution is an intensive ingestion framework dedicated to secure range query processing on encrypted data. The security and efficiency of this framework still remain high. This contribution results from the fact that all existing schemes cannot provide a satisfactory ingestion throughput for real-life needs. To overcome this limitation, we aim at enhancing our first contribution so that it is able to scale the intake ability up as much as possible. In particular, we introduce a new architecture relying on a set of shared-nothing machines and make it fully distributed. By doing it, the ingestion throughput can be scaled up by just adding more nodes into the system. In addition, a new data representation and an asynchronous publishing method are presented and integrated into this architecture to avoid throughput degradation or potential congestion. By precisely coordinating all of them together, our framework can support an intensive consumption throughput, e.g., over 160 thousand record insertions in a second. As compared to our previous contribution, the ingestion throughput is improved by up to $\sim 43\times$. Besides, we adapt our framework to a stronger type of attackers (e.g., online attackers) by introducing and integrating a new noise management into the new architecture. Interestingly, this method also increases the practicality of the framework.

The third contribution is a novel scheme for private range query processing on encrypted data. This scheme addresses the need of a scalable solution in terms of efficiency, high security, practical storage overhead, and numerous updates, which cannot be supported by the former schemes. To achieve it, we develop our solution relying on equal-size chunks (buckets) of data and secure indexes. The former helps to protect privacy of the underlying data from the adversary while the latter enables efficiency. To support lightweight updates, we propose to decouple secure indexes from their buckets by using use equal-size bitmaps. These bitmaps privately maintain links between secure indexes and buckets

of outsourced data. This decoupling approach allows our scheme to efficiently support unlimited lightweight updates without revealing anything about underlying plaintexts. More importantly, our scheme incurs significantly lower storage overhead as compared to its counterparts. With thorough experiments, we show that this novel scheme outperforms the state-of-the-art schemes in various metrics. For example, as compared to PINED-RQ [94], PARADOT is two orders of magnitude faster in terms of query response latency and uses at most $\sim$111$\times$ less space requirement. Moreover, PARADOT is able to efficiently support numerous updates that are often very costly or even not supported in previous works.

Resumé

Au cours des dernières années, l'infonuagique est devenue de plus en plus une norme pour réduire les coûts et offrir de l'élasticité. Alors que les fournisseurs de nuage élargissent leurs services, leurs clients sont toujours préoccupés par la sécurité des technologies des nuages. Cela est principalement dû au fait que les problèmes de sécurité entraînent souvent des violations de données qui nécessitent des efforts considérables et des coûts de correction extrêmes. Pour résoudre ces problèmes, le chiffrement est généralement utilisé pour protéger les données confidentielles stockées et traitées sur des nuages non fiables. Le chiffrement des données externalisées atténue toutefois les fonctionnalités des applications, car la prise en charge des opérations fondamentales sur les données chiffrées est encore limitée, les requêtes d'intervalle par exemple.

Malgré les récents travaux sur le *Fully Homomorphic Encryption (FHE)* [43, 44] qui montrent qu'il est possible de faire des calculs arbitraires sur des données chiffrées, les surcharges sont extrêmement hautes. Une approche plus pratique est d'utiliser des schémas de chiffrement qui permettent à des nuages non fiables d'effectuer des primitives de calcul spécifiques, par exemple des requêtes d'intervalle sur des données chiffrées.

Dans cette thèse, nous nous concentrons sur le problème de la prise en charge des requêtes d'intervalle *non agrégées* sur des données chiffrées stockées dans les nuages. Une requête d'intervalle est une opération fondamentale dans une base de données qui permet d'exprimer une restriction limitée sur les enregistrements récupérés. Par exemple, un hôpital peut externaliser la gestion des dossiers électriques de ses patients, la requête suivante de type SQL peut récupérer les dossiers des patients dont l'âge est compris entre a et b :

$$\texttt{SELECT * FROM patients WHERE age} \geq \texttt{a AND age} \leq \texttt{b}$$

Ces dernières années, de nombreuses études ont été réalisées sur le traitement des requêtes d'intervalle sur des données chiffrées. Néanmoins, aucun des schémas précédents

ne présente des performances satisfaisantes pour les systèmes modernes qui exigent non seulement des réponses à faible latence, mais aussi une haute évolutivité. En particulier, la plupart de ces solutions tentent principalement de trouver un compromis entre efficacité et sécurité. Par exemple, alors que les solutions de chiffrement à vecteur caché (Hidden Vector Ecryption) entraînent des surcharges de calcul prohibitives, les techniques basées sur la préservation de l'ordre (Order Preserving Encryption) sont vulnérables aux attaques statistiques. De même, les systèmes de conteneurisation (Bucketization) sont soit sans garantie formelle, soit confrontés à de hauts taux de faux positifs. Il est donc difficile d'obtenir des solutions évolutives à partir des schémas précédents. Bien que certains systèmes basés sur un index, comme PINED-RQ [94]), permettent d'obtenir à la fois un traitement rapide des requêtes et un haut niveau de sécurité, ils sont encore loin d'être satisfaisants à divers points de vue. Nous nous concentrons en particulier sur les dimensions suivantes.

Premièrement, les systèmes modernes sont souvent confrontés au contexte dans lequel les données sont générées rapidement par diverses sources de données telles que l'Internet des Objets (IoT) ou les applications mobiles. Cette situation exige par la suite que le système supporte un haut débit d'**ingestion** pour éviter les embouteillages potentiels. Notre observation montre que la plupart des protocoles de requête d'intervalle sécurisée existants souffrent de goulots d'étranglement dans un tel contexte. En particulier, les systèmes précédents doivent effectuer des opérations supplémentaires (prétraitement) sur des données avant de transmettre les données traitées au nuage. Ces opérations sont exécutées sur un composant de confiance (par exemple, le collecteur) afin de garantir la confidentialité des données. Malheureusement, ce processus est souvent lourd et prend du temps, comme le chiffrement des données et la construction d'un index sécurisé. Par conséquent, les solutions existantes, en particulier celles basées sur un index, souffrent de hauts niveaux de goulots d'étranglement au niveau du collecteur lorsque les données arrivent à grande vitesse. Plus important encore, toutes les solutions de pointe n'ont pas envisagé un cadre permettant un débit d'ingestion intensif pour des besoins réels, par exemple des centaines d'insertions par seconde.

Deuxièmement, en plus de la nécessité de requêtes à faible latence, les systèmes modernes devraient être efficaces en termes de gestion de l'**espace**. Mais la plupart des systèmes efficaces, qui disposent d'un calcul rapide des requêtes et d'une protection élevée de la confidentialité, doivent faire face à des surcharges de stockage sévères telles que [65, 32, 64, 86]. Bien que le protocole de [94] présente un petit espace de surcharge, il ne

fonctionne que dans des circonstances restreintes, par exemple, un individu possède un petit nombre d'enregistrements. Sinon, il souffre d'une surcharge d'espace prohibitif de données fictives lorsqu'un individu est lié à plusieurs enregistrements dans un ensemble de données. Il est également intéressant de noter que le problème de surcharge de stockage mène non seulement à une gestion inefficace de l'espace, mais peut également augmenter le temps de réponse aux requêtes.

Troisièmement, **la mise à jour** (insertion, modification et suppression) est une opération de base des systèmes de bases de données. En raison de la haute fréquence des mises à jour dans le contexte cible, cette opération devrait être rapide et légère. Cependant, aucune des solutions efficaces précédentes ne peut répondre à ces exigences, et même beaucoup d'entre elles ne permettent pas l'opération de mise à jour. Bien que certaines solutions prennent en charge les mises à jour, elles sont soit très coûteuses, soit limitées. Par exemple, alors que les opérations de mise à jour dans [32, 64] créent d'importantes surcharges de communication, la solution dans [94] ne prend en charge qu'un nombre limité de mises à jour.

Dans cette thèse, nous visons à fournir des solutions évolutives pour le traitement des requêtes d'intervalle sécurisées en préservant une haute efficacité et une forte protection de la confidentialité. Notre contribution est multiple : 1) Nous introduisons une approche pour soutenir le traitement des requêtes d'intervalle sécurisées dans le contexte où les données arrivent à un haut débit. 2) Nous proposons un cadre d'ingestion intensive dédié au traitement des requêtes d'intervalle sécurisées sur des données chiffrées. 3) Nous présentons un schéma évolutif pour le traitement des requêtes d'intervalle privées sur des données externalisées.

La première contribution consiste à éviter les goulots d'étranglement potentiels dans le contexte où le système est confronté à un haut débit de données entrantes. En outre, notre approche préserve la rapidité du traitement des requêtes et une forte protection de la confidentialité. Pour atteindre cet objectif, nous étendons PINED-RQ, l'une des solutions de pointe les plus efficaces, et adaptons ce travail au contexte cible. À cette fin, nous introduisons une notion de modèle d'index dans PINED-RQ afin que le système puisse consommer les données entrantes à la volée. De plus, une stratégie de gestion du bruit et une structure de données complémentaires sont conçues pour prévenir les fuites potentielles de données privées causées par cette extension. Nous développons ensuite une version parallèle de cette extension pour améliorer le débit d'ingestion. Avec des hypothèses pratiques, nous prouvons que la protection de la confidentialité de notre solution

est aussi forte que celle du PINED-RQ. Les résultats expérimentaux montrent que notre proposition est plus performante que PINED-RQ dans divers indicateurs. Par exemple, le délai de publication d'un ensemble de données de 0,5 million d'enregistrements est réduit jusqu'à $\sim 35\times$, tandis que le débit de données maximal du système est réduit jusqu'à $\sim 2,7\times$.

La deuxième contribution est un cadre d'ingestion intensive, `FRESQUE`, dédié au traitement de requêtes d'intervalle sécurisée sur des données chiffrées. La sécurité et l'efficacité de ce cadre restent élevées. Cette contribution résulte du fait que tous les systèmes existants ne peuvent pas fournir un débit d'ingestion satisfaisant à des besoins réels. Pour surmonter cette limitation, nous visons à améliorer notre première contribution afin qu'elle soit en mesure d'augmenter la capacité de réception autant que possible. En particulier, nous introduisons une nouvelle architecture basée sur un ensemble de machines et de manière entrièrement distribuée. Ce faisant, le débit d'ingestion peut être augmenté en simplement des noeuds dans le système. En outre, une représentation des données et une méthode de publication asynchrone sont présentées et intégrées dans cette architecture pour éviter la dégradation du débit ou une congestion potentielle. En coordonnant précisément tous ces éléments, notre cadre peut soutenir un débit de consommation intensif, par exemple plus de 160,000 insertions d'enregistrements en une seconde. Par rapport à notre contribution précédente, le débit d'ingestion est amélioré jusqu'à $\sim 43\times$. De plus, nous adaptons notre cadre à un type des attaques plus fortes (par exemple, les attaques en ligne) en introduisant et en intégrant une nouvelle gestion du bruit dans la nouvelle architecture. Il est intéressant de noter que cette méthode augmente également la praticabilité du cadre.

La troisième contribution est un nouveau système de traitement des requêtes d'intervalle privées sur des données chiffrées, `PARADOT`. Ce système répond au besoin d'une solution évolutive en termes d'efficacité, de haute sécurité, de frais de stockage pratiques et de nombreuses mises à jour, qui ne peuvent être pris en charge par les anciens systèmes. Pour y parvenir, nous développons notre solution en nous basant sur des conteneurs de données de taille égale et des index sécurisés. Le premier permet de protéger la confidentialité des données contre l'adversaire, tandis que le second permet l'efficacité. Pour permettre des mises à jour légères, nous proposons de découpler les index sécurisés de leurs conteneurs en utilisant des bitmaps de taille égale. Ces bitmaps maintiennent en privé les liens entre les index sécurisés et les conteneurs de données externalisés. Cette approche de découplage permet à notre système de prendre en charge efficacement un nombre illimité

de mises à jour légères sans révéler quoi que ce soit sur les données en clair. Plus important encore, notre système implique des frais de stockage nettement inférieurs à ceux de ses homologues. Grâce à des expériences approfondies, nous montrons que ce nouveau système surpasse les systèmes de pointe dans divers indicateurs. Par exemple, par rapport à PINED-RQ [94], `PARADOT` est deux ordres de grandeur plus rapide en termes de latence de réponse aux requêtes et utilise au maximum $\sim$111$\times$ moins d'espace requis. De plus, `PARADOT` est capable de prendre en charge efficacement de nombreuses mises à jour qui sont souvent très coûteuses ou même non prises en charge dans les travaux précédents.

Table of Contents

Chapter 1

Introduction

1.1 Context and Motivation

With the prosperity of online social network, web-based services, and IoT, an unprecedented 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 query processing, but also intensive ingestion throughput over incoming data. This puts a tremendous strain on data management activities.

A dynamic and elastic IT infrastructure like cloud computing is a good alternative to handle such massive and unpredictable data [68, 55]. Cloud computing has increasingly gained huge attention over the last years due to continuously increasing needs. Many organizations of all sizes adopt cloud computing technologies to benefit from resource and cost elasticity. According to a recent survey [56], 81% of organizations confirmed already using computing infrastructure or having applications in the cloud.

Despite huge benefits, cloud computing encounters security threats. These threats primarily come from the distributed architecture of cloud computing [21, 90]. Indeed, as several virtual machines share the same physical server, a malicious virtual machine can infer some information about the other virtual machines through shared memory or other shared resources [2, 51, 118]. Another severe threat that cloud computing also faces is insider threats, which could account for more than 43% of all data breaches according to recent statistics [57]. Generally, the insider threat can be considered as the one who has access rights to a system and wrongly uses her privileges [102, 9]. These threats can be malicious such as staff members going rogue, e.g., in 2010, a Google employee broke

into the Gmail and Google Voice accounts of several children [48], but they can also be due to negligence or simple human errors, e.g., in 2019, thousands of customer records of Capital One are leaked by a firewall misconfiguration [17] when they are stored on Amazon Web Services (AWS). Such data breaches not only require significant efforts for corrections, but also may destroy the reputation of related organizations. These security threats consequently make cloud computing become an untrusted environment, remaining one of the top concerns hindering the widespread adoption of such technology [21, 96, 90]. In order to urge the broader adoption of cloud computing, such security concerns must be addressed.

Over the last years, encryption has become one of the powerful techniques to protect confidential data stored on untrusted environments like clouds [93, 85, 88, 105]. With such an approach, sensitive data is first encrypted at a trusted component before the corresponding ciphertext is sent to the cloud for managing. Nevertheless, this straightforward solution also poses several challenges to data management activities. One of them is that the encrypted data may not be decrypted at the cloud due to the requirement of "total privacy". This means that the client may not fully trust the cloud server for complete access to its sensitive data. A naive solution is to transmit the encrypted data from the cloud to a trusted component, decrypt the data, and execute queries over the corresponding plaintexts. But this approach is not scalable and diminishes almost the advantages of cloud computing since it is now only considered as a storage cloud. Therefore, there is a need of performing computations over encrypted data without decrypting the data at untrustworthy clouds.

Although recent work on *Fully Homomorphic Encryption (FHE)* [43, 44] shows it is able to compute arbitrary computations on encrypted data, its performance overheads are extremely high. A more practical approach is to use encryption schemes that allow untrusted cloud to perform specific computation primitives, e.g., range queries on encrypted data.

In this dissertation, we focus on the problem of supporting *non-aggregate* range queries on encrypted data stored on clouds. A range query is a fundamental operation in database that enables to express a bounded restriction over the fetched records. For instance, a hospital may outsource the management of the electric records of its patients, the following SQL-like query can retrieve the records of patients with the age between a and b:

$$\texttt{SELECT * FROM patients WHERE age} \geq \texttt{a AND age} \leq \texttt{b}$$

More specifically, we consider solutions that allow efficiently perform range queries over

encrypted data while still preserving a high level of security. Besides, proposed solutions must be scalable enough to cope with the target context.

Addressing the problem of range query processing on encrypted data has increasingly become a hot topic over the last years [50, 63, 53, 1, 14, 12, 11, 54, 112, 66, 32, 64, 94, 86]. Prior schemes can be classified into four main categories according to the techniques they use. First, hidden vector encryption (HVE) methods [14, 112] use bilinear groups equipped with bilinear maps and hide attributes in an encrypted vector. Range predicate is evaluated over such encrypted vector. Second, order preserving encryption (OPE) schemes [1, 12, 11, 74, 87, 86] aim at keeping the order of the ciphertexts the same as the that of the corresponding plaintexts. Such property enables range predicate evaluation over encrypted data. Third, bucketization based solutions [50, 53, 54, 63] propose to partition an attribute domain into a finite number of buckets and a range query retrieves all data of buckets falling within the range. Fourth, several index-based approaches [32, 66, 64, 94, 86] have been proposed with the aim of maintaining secure indexes on encrypted data such that they often achieve very fast range query processing.

Nonetheless, none of existing schemes can cope with scalability requirements of modern systems. More specifically, most of these solutions mainly attempt a trade-off between efficiency and security, instead of achieving both of them. For example, while the hidden vector encryption solutions incur prohibitive computation overhead, the order preserving based techniques are vulnerable to statistical attacks. Similarly, the bucketization schemes either lack formal guarantees or confront high false positives. Therefore, it is hard to achieve scalable solutions from these schemes. Although some of the index-based schemes like PINED-RQ [94] can achieve both fast query processing and a high level of security, they are still far from satisfactory manners in various perspectives. We particularly focus on the following dimensions.

First, modern systems often confront the context where data is generated rapidly by various data sources such as IoT devices or mobile applications. This situation subsequently demands the system to support a high **ingestion** throughput for avoiding potential congestion. Our observation show that most existing secure range query protocols suffer from bottlenecks in such context. In particular, prior schemes have to perform additional operations (pre-processing) on clear data before sending the processed data to the cloud. These operations are executed on a trusted component (e.g., collector) to ensure the confidentiality of the data. Unfortunately, this process is often heavy and takes time, such as data encryption and secure index construction. As a consequence, bottle-

necks highly occur at the collector in the existing solutions, especially the index-based solutions, when data come at a high speed. More importantly, a framework supporting intensive ingestion throughput for real-life needs, e.g., hundreds of insertions/second, has not been considered by all of the state-of-the-art solutions.

Second, in addition to the need of low-latency queries, modern systems are expected to be efficient in terms of space management. But, most of the efficient schemes, that have fast query computation and strong privacy protection, confront severe **storage** overhead such as [65, 32, 64, 86]. Although the protocol in [94] has small space overhead, it only works in restricted circumstances, e.g., an individual has a small number of records. Otherwise, it suffers from prohibitive space overhead of dummy data when an individual is linked to multiple records in a dataset. It is also worth noting that the storage overhead problem not only leads to inefficient space management, but also may increase query response time.

Third, **update** (insertion, modification, and deletion) is a basic operation of database systems. Due to the high frequency of updates in the target context, such operation is expected to be fast and lightweight. However, none of prior efficient solutions can cope with these requirements, even many of them do not enable the update operation. Although some support updates, they are either very costly or limited. For instance, while update operations in [32, 64] create high communication overheads, the solution in [94] only supports a limited number of updates.

1.2 Research contributions

According to the drawbacks of the existing schemes as presented in Section 1.1, this section highlights the research goals of this dissertation. As mentioned earlier, despite the fact that some schemes enable a good level of query processing and security, they do not cope with the context where data arrives at a high rate. It is expected a new approach to avoid the respective problems, such as bottlenecks, while still maintaining efficiency and high security. Besides, we also need a scalable framework that allows the system smoothly ingest a large number of records per second for real-life needs, e.g., over 100K records/s. This framework should ensures flexibility, efficiency, and strong privacy protection. Lastly, existing solutions cannot achieve all requirements of fast query computation, small storage overhead, high security, and lightweight updates. There is the need of designing a novel scheme that meet all these requirements together.

We propose several practical solutions for range query processing on outsourced databases in clouds. In particular, we have made the following contributions.

1) `PINED-RQ++`: An adaptation of PINED-RQ to the context of high rate of incoming data. This extension addresses the bottleneck problem of the existing schemes in case of high speed of incoming data while still ensuring high security and fast range query processing. To this purpose, we extend PINED-RQ, one of the state-of-the-art solutions, achieving both efficiency and high security by using secure indexes. In other words, we introduce and integrate the notion of *index template* to PINED-RQ, that allows the collector to process incoming data *on the fly* while preserving secure indexes for published data. To maintain high security, we introduce a noise management strategy as well as a complementary data structure (*matching table*) to the integration process. In addition, we develop a parallel `PINED-RQ++` with the aim of improving the ingestion throughput of the system. The experimental results exhibit very good performance, e.g., the publishing time of the NASA dataset ($\sim$0.5M records) [78] is reduced up to $\sim$35x while the maximum data rate at the collector experiences a reduction of up to $\sim$2.7x. Finally, we propose a new architecture that enables to achieve the same level of security with PINED-RQ.

2) `FRESQUE`: An intensive ingestion framework dedicated to secure range query processing on clouds. We develop a framework for supporting highly scalable ingestion throughput that is required by real-life applications. To achieve that goal, we re-design the architecture of `PINED-RQ++` and make it fully distributed by using a set of *shared-nothing* machines. We then present an array data representation and an asynchronous publishing strategy to the new architecture. By coordinating all of them together, this framework is able to support an intensive ingestion throughput, e.g., over 160K records/second. Experimental results show that compared to (non-)parallel `PINED-RQ++`, the ingestion throughput is improved by $\sim$43$\times$ and $\sim$5.6$\times$ in NASA [78] dataset, respectively. Additionally, we propose a new noise management method that not only aligns `FRESQUE` with a stronger type of adversaries (e.g., *online attackers*) but also improves its practicality.

3) `PARADOT`: A novel scheme for scalable and private range query processing on outsourced data. We propose a scalable approach for supporting secure range query processing on encrypted data, `PARADOT`, that meets practical requirements, namely small storage overhead and unlimited lightweight updates, while ensuring efficiency and strong

privacy protection. Our scheme relies on equal-size buckets and secure indexes. The former enables to protect the privacy of outsourced data while the latter allows to provide fast range query processing. In addition, we propose to decouple secure indexes from their buckets by using equal-size bitmaps. These bitmaps privately maintain links between secure indexes and buckets of outsourced data. This decoupling approach allows our scheme to efficiently support unlimited lightweight updates without revealing anything about underlying plaintexts. Moreover, the proposal exhibits a practical space overhead of encrypted bitmaps. Experimental results show that `PARADOT` significantly outperforms the state-of-the-art solutions in various metrics when both uniform and skewed distributions are used. For instance, as compared to PINED-RQ, our solution is $\sim 176\times$ faster in terms of query response latency and uses at most $\sim 119.55\times$ less space requirement. Furthermore, with dataset sizes of 0.5M and 5M records, the index size of `PARADOT` is about 42MB and 406MB while PBtree [66] requires 1.598GB and 18.494GB, respectively.

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 gives basic concepts of cloud computing and data privacy. Chapter 3 presents background knowledge for serving the understanding of our contributions. Additionally, the related works on secure range queries are also reviewed in this chapter. Chapter 4 describes an extension of PINED-RQ. Chapter 5 presents an intensive ingestion framework for secure range query processing. We then introduce a novel scheme for scalable and private range query processing over encrypted data in Chapter 6. Lastly, we give conclusion and our future works in Chapter 7.

Chapter 2

Cloud computing and Privacy

2.1 Introduction

This chapter aims at giving key concepts about cloud computing as well as data privacy. We start from cloud computing with regard to its core elements and potential security issues in Section 2.2. We then briefly discuss personal data and privacy in Section 2.3. In addition, we present an overview about the encryption technique that is widely used to protect privacy in Section 2.4. Section 2.5 concludes this chapter.

2.2 Cloud computing

Cloud computing is an IT organization paradigm that aims at offering *on-demand* resources to users and companies [18]. Traditionally, enterprises of any size have to possess IT infrastructures and need technical employees to deploy and manage services, which result in huge costs of ownership. In contrast, cloud computing offers virtual resources, which encompasses both hardware and software, anytime, anywhere following a *pay-as-you-go* model. By using this paradigm, enterprises not only cut down the maintenance costs, but also easily create scalable global applications. These advantages have led to a widespread adoption of cloud computing over the last years. Many organizations migrate their applications from on-premise to cloud infrastructures. In 2020, IDG reported that 81% of organisations have at least one application or portion of their computing infrastructure in the cloud [56].

While companies can benefit a lot from cloud computing, there are still several security

issues when using systems that are not provided in-house. To understand these security issues, first it is important to understand the basic concepts and technologies of cloud computing.

2.2.1 Concepts in cloud computing

Cloud computing offers software, platforms, and infrastructure as services based on pay-as-you go models. Until now, there have been three main service models that are provided by cloud computing: *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)*, and *Software as a Service (SaaS)*. SaaS offers the highest level of abstraction and allows users to access to ready-to-use applications which are already deployed in the cloud. Cloud-based applications such as Google G Suite [72], Microsoft Office 365 [24], Dropbox [33], and Overleaf [83] are instances of SaaS. In PaaS, users are offered with more control over their IT resources. It provides a framework for developers to easily create cloud applications. But users do not have control on the underlying infrastructure. Examples of PaaS include AWS Elastic Beanstalk [6], Google App Engine [70], and Microsoft Azure [23]. Finally, IaaS provides access to computational resources such as Virtual Machines (VMs) and storage space. It is the pillar of cloud computing and allows users to access the IT infrastructure they require in a flexible way. Examples of IaaS include Amazon EC2 [5], Google Compute Engine [71], and Microsoft Azure [23].

Regarding the access scope, clouds can be classified into three categories: *public cloud*, *private cloud*, and *hybrid cloud*. Private clouds refer to being the property and managed by an organization. They are only accessed within the organization or by its partners/customers. Meanwhile, public clouds are often owned by a single organization, but they are used in public. Finally, hybrid clouds are a combination of private and public cloud. Public clouds are the most popular and often owned by a single large company such as Amazon, Google, or Microsoft. Their infrastructure can be scattered around the world. For example, in 2020, Amazon operates in 77 availability zones located in 24 regions in 5 different continents [7]. In this dissertation, we primarily focus on public clouds for designing our solutions.

2.2.2 Virtualization

The main feature that makes cloud computing appealing to businesses is scalability. This property is mainly created by a virtualization mechanism. The virtualization is a

process that converses physical IT resources into a virtual IT resources. The IT resources encompass servers, storage, and network. A Virtual Machine Monitor (VMM) or hypervisor is a software or a firmware component that can virtualize system resources. Hypervisor is considered a core component in virtual computer systems. It resides between the Virtual Machines (VMs) and the real machine hardware and is used to control the virtualized resources [103]. With the hypervisor, multiple isolated virtual machines run on the same physical host. Hypervisors can be divided into two types [91] as depicted in Figure 2.1.

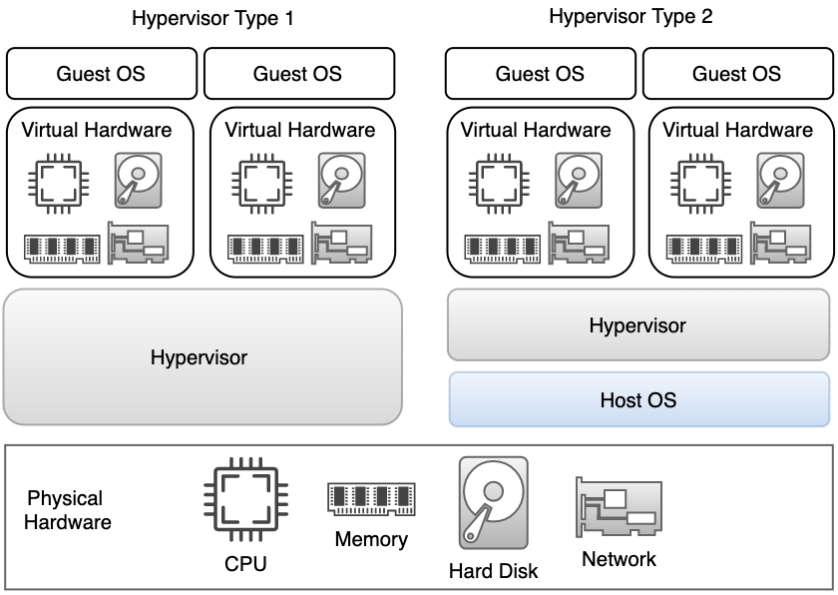


Figure 2.1 – Types of hypervisor in cloud computing [38]

— **Type 1:** Here the hypervisor runs directly on the physical host hardware, interacting directly with its CPU, memory, and physical storage. This type of hypervisor is efficient since it does not depend on any intermediary layers. Another benefit with this approach is that security levels can be increased by isolating the guest VMs. This means that when a virtual machine is compromised, it can only impact itself and does not interfere with the hypervisor or other guest VMs.

— **Type 2:** The second type of hypervisor is installed and run on a pre-existing operating system (host operating system) that provides virtualization services such as I/O device support and memory management. All interactions of virtual machines, namely I/O requests, network operations, and interrupts, are managed by the hypervisor. One of the downsides of this approach is that it may introduce potential

security risks when the host OS is compromised by attackers since they could then manipulate any guest OS running in this hypervisor.

2.2.3 Security threats in cloud computing

With the promise of high scalability and availability, it has become increasingly prevalent to host services as well as data in public clouds. By moving data services to the cloud, data owners can cut down costs in various aspects of data management. However, these benefits often come with a risk of exposing data to more security threats as compared to traditional computer systems. This section thus discusses some major data security threats in cloud computing, including data breaches, malicious insiders, and data loss.

Data breaches. Data breach is defined as the leakage of confidential information of customers or organizations to unauthorized user. A data breach may result from various sources, such as flaws in infrastructure, human errors, and poor security practices [4]. For example, in 2015, BitDefender leaked an undisclosed number of customer usernames and passwords due to a security vulnerability in its public cloud application hosted on Amazon WebServices (AWS) [15]. Moreover, data breaches could also take place as several virtual machines share the same physical server. That is, a malicious virtual machine can infer sensitive information of other virtual machines through shared memory or other shared resources [2, 51, 118]. A cross-VM side channel attack was introduced by Zhang et al. [118], that extracts cryptographic keys of other VMs on the same system and can access their data.

Malicious insiders. Generally, a malicious insider threat to an organization is the one who has access rights to a system, namely a current or former employee, contractor, or other business partners, and wrongly uses their privileges [102, 9]. For example, in 2010, a Google employee broke into the Gmail and Google Voice accounts of several children [48]. A malicious insider threat can also come from negligence or simple human errors. In 2019, thousands of customer records of CapitalOne, being stored on AWS, were disclosed by a firewall misconfiguration [17].

Data loss. Data loss is one of the major issues related to cloud security. Similar to data breach, data loss is a sensitive matter for any organization and can devastatingly affect its business. Data loss can be caused by malicious attackers, data deletion, data

corruption, loss of data encryption key, or natural disasters [4].

2.3 Personal data and privacy

With the convergence of mobile communications, Internet-connected devices, and online social networks, we are witnessing an unprecedented increase in the creation, collection, and consumption of huge amount of personal data. This personal data is precious since it may result in a great potential for applications and business, e.g., seasonal disease tracking, smart grids management, healthcare surveillance, participatory sensing, etc. Mining and analyzing such personal data allows us to well understand the human's behavior and significantly benefit from this knowledge. For example, users often pay unlimited access to their data for using free services that are provided by big companies like Google. This company exploited the collected data for targeted ads and gained $31.2 billion in just the first three months of 2018 [89].

However, the collected data often contains sensitive information, e.g., sociodemographic data, medical data, tweets, photos, videos, and location information, etc. Companies seek to protect data from compromise by external attackers and malicious insiders. This is because data breaches often not only require huge efforts for correction, but also may destroy the reputation of companies. One of the big challenges for companies is how to avoid privacy violation while still making use of the data. In other words, how they still support computation over the data stored on untrusted clouds without violating the data privacy.

2.4 Data confidentiality in Cloud

Encryption is considered as a standard technique to ensure confidentiality of sensitive private data stored on untrusted environment like clouds [85, 88, 105]. However, simple encryption approaches cannot support complex requirements such as queries, thus a variety of techniques have been proposed for query processing on encrypted data over the last years. For example, fully homomorphic encryption [43, 42] is a kind of encryption system that allows performing any operation on ciphertexts without decrypting them. However, it confronts very complicated calculation, and the cost of computing and storage is extremely high. This means that the fully homomorphic encryption is still impractical for modern systems. During the meantime, various encryption schemes for specific tasks have

been studied, namely order preserving encryption [1], encrypted keyword search [31, 19], equality queries [116], and range queries [50, 53, 94]. These methods mainly focus on the dimensions of security and efficiency. Particularly, most of them attempt to reach a trade-off between performance and privacy. Although some can provide fast query computation on encrypted data, they are still far from the performance needs of present systems. Moreover, the scalability dimension has not been thoroughly considered in prior schemes, thus it is non-trivial to integrate them into modern systems. In this dissertation, we focus our attention on range query and encrypted data stored on clouds. More importantly, we take the scalability dimension into account so that our solutions well suit the needs of real-life applications.

2.5 Conclusion

This chapter presents an overview of cloud computing. In particular, we describe key elements of cloud, e.g., cloud services, virtualization mechanism, and its security issues. Finally, the privacy concept and a common technique for protecting data privacy are also discussed in this chapter. In the next chapter, we give background techniques for serving the understanding of our contributions. Additionally, we present related works on secure range query processing.

Chapter 3

Background and Related Works

3.1 Introduction

In this chapter, we present background knowledge and related works for understanding the contributions in this dissertation. We start with basic concepts in database as well as related privacy tools. Then, several studies on querying encrypted data are reviewed in Section 3.4. Especially, we give a focus on the scalability of these schemes with respect to our target context. Research problems are thus identified in this section. We next briefly present PINED-RQ in Section 3.5, a recent scheme, that is closely related to our contributions. Section 3.6 concludes this chapter.

3.2 Basic concepts in database

3.2.1 Range queries

Range query is a fundamental database operation that retrieves all records where some value is between an upper and a lower boundary. For example, a doctor, Alice, wants to get encrypted records of patients whose ages are between a and b. One example of a range query (in form of a SQL query) can be: `SELECT` * `FROM` database `WHERE` age $\geq$ a `AND` age $\leq$ b. In this dissertation, we only focus on one dimension non-aggregate range queries since they are one of the most basic operations.

3.2.2 Indexing

Indexing is one of the most essential database techniques. Indexing is related to creating and maintaining any auxiliary data structure that helps queries access data fast. With the support of indexes, queries can quickly find and retrieve certain records without the need of scanning the entire table. In this section, we mainly focus on the traditional indexes that efficiently support range queries such as B-tree and B+-tree.

B-tree. A B-tree is a tree data structure that keeps data sorted and supports searches, insertions, and deletions in logarithmic time [22]. The B-tree is considered as a generalization of a binary search tree (BST). The main difference is that nodes of a B-tree keep pointers to many children nodes rather than being limited to only two as in BST, as illustrated in Figure 3.1. A B-tree of order m is an m-way search tree has the following properties:

— All leaves are on the same level.

— The nodes in a B-tree of order m can have a maximum of m children.

— Each internal node except the root has at most m children and at least $\lceil m/2 \rceil$ children.

— A non-leaf node with k children has $k - 1$ keys.

— The root has at least two children unless it is a leaf.

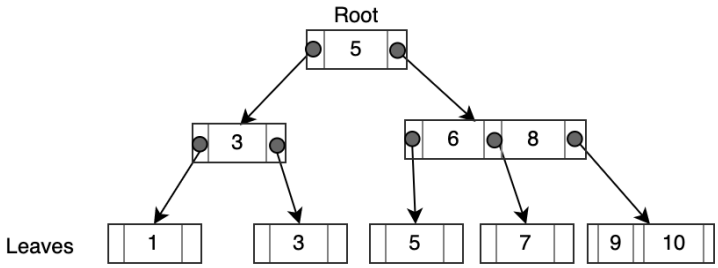


Figure 3.1 – A B-tree of order three

B+-tree. A well-known variant of the B-tree is B+-tree which is extensively used for disk-resident data. The main difference is that in B-tree the keys and pointers to records can be stored as internal as well as leaf nodes whereas in B+-tree, the internal nodes store only the keys (and not the pointers to records) [22]. The leaf nodes in B+-tree store the pointers to the record corresponding to the key. The leaves are linked to each other in a

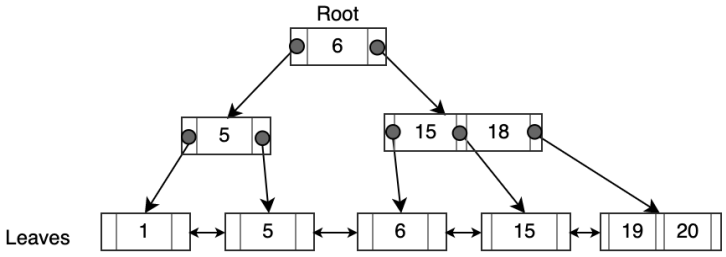


Figure 3.2 – Example of B+-tree

doubly linked list fashion to facilitate their traversal without accessing the parents. For this reason, B+-tree can provide fast and efficient searches. Figure 3.2 presents a simple example of B+-tree with the order of 3.

To maintain such data structure in untrustworthy environments without the leak of privacy of underlying dataset is non-trivial work. Although encryption can be used to conceal node data, the security is not fully guaranteed. For example, index scan operations may reveal the order of underlying value of tree nodes, that might be vulnerable to statistical attacks [79]. In this dissertation, a variety of secure indexes will be discussed. Especially, we seek to use secure indexes for our contributions due to their fast range searches.

Bitmap index. The bitmap index was first introduced by Spiegler and Maayan [98].

Dataset	
RowID	GPA
1	0
2	3
3	4
4	1
5	3
6	0
7	2
8	3
9	2
10	4

Bitmap index				
=0	=1	=2	=3	=4
1	0	0	0	0
0	0	0	1	0
0	0	0	0	1
0	1	0	0	0
0	0	0	1	0
1	0	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	1	0	0
0	0	0	0	1
b_1	b_2	b_3	b_4	b_5

Figure 3.3 – Basic bitmap index for the column GPA that can only take on five distinct values from 0 to 4. RowID represents for "row identifiers"

It is used to boost the performance on various query types including range, aggregation,

and join queries. This technique uses a sequence of bits to indicate the presence or absence of an item in the indexed data. With the bitmap index, queries are evaluated with bitwise logical operations, such as AND, OR, NOT, and XOR. Bitmap index is one of the most efficient indexing methods available for speeding up range queries for read-only or read-mostly data [115] while traditional tree-based indexing structures are designed for databases that change frequently over time.

Basic bitmap index. Given an attribute with n distinct values, the basic bitmap index creates n bitmaps with m bits each, where m indicates the number of records (rows) in the dataset. As a simple example shown in Figure 3.3, the attribute GPA has 5 distinct values (0-4), and there are 10 records. Then, 5 bitmaps are generated and each has 10 bits. If the indexed attribute in the i^{th} record is of a specific value, j, the i^{th} bit of the corresponding bitmap j is set to "1", otherwise the bit is "0". That is, a bit with value 1 indicates that a particular row has the value represented by the bitmap.

Queries can be quickly answered with bitwise logical operations on the bitmaps which are well adapted to the computer hardware. In the example shown in Figure 3.3, a range query "GPA $>$ 0 and GPA $<$ 4" can be answered by performing bitwise OR on b_2, b_3 and b_4 (i.e., $b_2|b_3|b_4$). Such property makes the bitmap index particularly useful for query-intensive applications, namely data warehousing and on-line analytical processing. In Chapter 6, we also take advantage of this property of the bitmap index to build a novel scheme for scalable secure range queries.

3.3 Privacy tools

3.3.1 Differential privacy

Differential privacy is a powerful tool for data privacy proposed by Dwork [34]. This model considers the very strong adversary that has unlimited computational power. Let $\mathcal{D}(A_1, ..., A_n)$ be the dataset and A_i is an attribute of that dataset. Let variable r is a record sampled from a universe $\mathcal{X}$. Two datasets $\mathcal{D}$ and $\mathcal{D}'$ are *neighboring* if they differ in only one record.

A query f is a function that maps dataset $\mathcal{D}$ to a real number: $f : \mathcal{D} \rightarrow \mathbb{R}$. Differential privacy uses a randomization mechanism $\mathcal{M}$ to mask the difference on the outputs between the neighboring datasets. The maximal difference on the outputs of query f is

defined as the *sensitivity* Δf. The sensitivity determines how much perturbation must be added to the outputs. In other words, the adversary learns nothing about an individual, regardless of whether her record is present or absent in $\mathcal{D}$. A formal definition of differential privacy is as follows.

Definition 1 (ϵ-differential privacy [34, 35]): A randomized function $\mathcal{M}$ satisfies ϵ-differential privacy, if for any set of outcomes $O \in Range(\mathcal{M})$, and any pair of neighboring datasets $\mathcal{D}$ and $\mathcal{D}'$,

$$Pr[\mathcal{M}(\mathcal{D}) = O] \leq e^{\epsilon} \cdot Pr[\mathcal{M}(\mathcal{D}') = O] \tag{3.1}$$

where ϵ is defined as the *privacy budget*, that controls the privacy level the mechanism $\mathcal{M}$. A smaller ϵ means stronger privacy level. Apparently, ϵ depends on specific application, and in practice it is often set as less than or equal to 1.

Basically, there are two privacy budget compositions that are widely used in the design of mechanisms [75], the *sequential composition* and the *parallel composition*, being defined as follows.

Definition 2 (Sequential Composition [75]): Let $\mathcal{M} = \{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$ denote a set of functions being sequentially performed on a dataset $\mathcal{D}$, and each $\mathcal{M}_i$ gives ϵ_i-differential privacy. Then $\mathcal{M}$ satisfies $(\sum_{i=1}^{m} \epsilon_i)$-differential privacy.

The sequential composition ensures the privacy guarantee for a sequence of differentially private computations. As a sequence of randomized mechanisms are sequentially performed on a dataset, the privacy budgets will be summed together.

Definition 3 (Parallel Composition [75]): Let $\mathcal{M} = \{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$ denote a set of functions, and each $\mathcal{M}_i$ gives ϵ_i-differential privacy guarantee on a disjoint subset of the entire dataset. Then $\mathcal{M}$ satisfies $(\max\{\epsilon_i, \ldots, \epsilon_m\})$-differential privacy.

The parallel composition considers the case where each $\mathcal{M}_i$ is performed on disjointed subsets of the dataset. The final privacy budget is determined by the largest one.

Definition 4 (Global Sensitivity): For a query $f : \mathcal{D} \rightarrow \mathbb{R}$, the global sensitivity

of f is defined as follows.

$$\Delta f = \max_{\mathcal{D},\mathcal{D}'} \parallel f(\mathcal{D}) - f(\mathcal{D}') \parallel_1 \tag{3.2}$$

where datasets $\mathcal{D}$ and $\mathcal{D}'$ differ in at most one record.

Roughly speaking, the global sensitivity of a function is the largest possible difference that a single record can have on the output of that function, for any dataset. In other words, it indicates how much the difference should be hidden in mechanisms. For example, the counting query normally has $\Delta f = 1$ since adding or removing one record from any dataset will change the count by at most 1.

Laplace Mechanism. The Laplace mechanism is the most common method to obtain ϵ-differential privacy. It adds controlled noise to the query result before returning the noisy result to user. The noise is sampled from the Laplace distribution, which has mean zero and scaling λ. Its respective probability density function is: $pdf(x, \lambda) = \frac{1}{2\lambda} e^{-|x|/\lambda}$.

Definition 5 (Laplace mechanism [36]). Given a function $f : \mathcal{D} \to \mathbb{R}$ over a dataset $\mathcal{D}$, the mechanism $\mathcal{M}$ gives the ϵ-differential privacy.

$$\mathcal{M}(\mathcal{D}) = f(\mathcal{D}) + Lap(\frac{\Delta f}{\epsilon}) \tag{3.3}$$

where Δf is the sensitivity of the function f.

3.3.2 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) [113] is a common instance of efficient private key encryption schemes satisfying semantic security.

Definition 6 (Semantic security [46]): 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 data set $\mathcal{D}$, 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:

$$Pr[A_n(E_\chi(\mathcal{D}), |\mathcal{D}|, \zeta) = g(\mathcal{D})] < Pr[A'_n(|\mathcal{D}|, \zeta) = g(\mathcal{D})] + \frac{1}{p(n)} \tag{3.4}$$

3.3.3 Computational differential privacy model

A variant of differential privacy that requires privacy guarantees to hold against computationally bounded adversaries [77]. This model considers the cryptographically-negligible leaks due to the use of efficient real-world encryption schemes such as AES (in CBC mode) [113] or Paillier scheme [84]. Definition 3 is a simplification of ϵ_n-SIM-CDP, the simulation-based computational differential privacy model proposed in [77].

Definition 7 (ϵ_n-SIM-CDP privacy [77]): 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:

$$Pr[A_n(f_n(\mathcal{D}, \zeta)) = 1] - Pr[A_n(F_n(\mathcal{D}, \zeta)) = 1] \leq \frac{1}{p(n)} \tag{3.5}$$

3.3.4 Unified privacy model

Sahin et al. [94] have introduced a probabilistic relaxation of a variant of differential privacy that considers computationally bounded adversaries [77] and that considers the cryptographically-negligible leaks due to the use of efficient real-world encryption schemes.

Definition 8 ((ϵ, δ)-Probabilistic-SIM-CDP [94]): 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 $\delta \in [0, 1]$.

3.3.5 Partially homomorphic encryption

Partially homomorphic encryption (PHE) schemes have the capacity of computing over encrypted data without access to the private key. For instance, additively homomorphic schemes, such as Paillier cryptosystem [84], enable to carry out the addition of ciphertexts,

such that the decrypted result is equal to the sum of the plaintexts. Given a message $m \in \mathbb{Z}_n$, where n is a product of two large prime numbers p and q, we denote $\mathtt{E}(m) \in \mathbb{Z}_{n^2}$ to be the encryption of m with the public key. Then, $\forall m_1, m_2 \in \mathbb{Z}_n$, $\mathtt{E}(m_1).\mathtt{E}(m_2) = \mathtt{E}(m_1 + m_2)$.

Any partially homomorphic encryption scheme used in this dissertation must meet two main properties. First, it needs to provide semantic security guarantees. Second, it should be additively-homomorphic. In particular, we consider the use of Paillier cryptosystem [84] to build `PARADOT` in Chapter 6.

3.4 Range queries over encrypted data

We now review prior works on privacy-preserving range queries over the last years. More importantly, we identify their drawbacks with respect to the target context. Particularly, none of efficient and secure schemes adapts well to the context of intensive ingestion and satisfies all target scalability requirements such as small space usage and unlimited lightweight updates. Table 3.1 presents a summary of related works as well as proposed solutions with respect to scalability requirements.

Table 3.1 – Summary of prior schemes as well as proposed solutions with respect to various metrics

Scheme	Formal security guarantees	Unlimited lightweight updates	Low latency	Small storage overhead	High ingestion throughput
FHE [43, 42]	✓				
ORAM [47, 82, 100]	✓				
HVE [14, 112]	✓				
Bucketization [50, 53, 54]		✓	✓	✓	
OPE [1, 12, 11, 74, 87]		✓	✓	✓	
PBtree [66]	✓		✓		
IBtree [64]	✓		✓		
ArxRange [86]		✓	✓		
Demertzis et al. [32]	✓		✓		
PINED-RQ [94]	✓		✓	✓	
`PINED-RQ++` (Chapter 4)	✓		✓	✓	✓
`FRESQUE` (Chapter 5)	✓		✓	✓	✓
`PARADOT` (Chapter 6)	✓	✓	✓	✓	

3.4.1 Fully homomorphic encryption

Fully Homomorphic Encryption (FHE) [43, 42] was invented by Gentry [43, 42] in 2009. It enables to compute any function directly on the ciphertexts without disclosing any information about the underlying plaintexts and results. Unfortunately, FHE is now still impractical due to its prohibitive overheads of space requirement and computational time.

3.4.2 Oblivious RAM

Oblivious RAM (ORAM) [47, 82, 100] was first introduced by Goldreich and Ostrovsky [47] with the aim of protecting software from the adversary who seeks to exploit memory access pattern. To achieve it, ORAM algorithms continuously shuffle and re-encrypt data as they are accessed by a client. By doing it, even though attackers can observe the physical storage locations accessed, they have negligible probability of learning anything about the true access pattern.

Several studies have been carried out to find solutions that are not only theoretically interesting, but also practical. Nevertheless, most of them still suffer from high bandwidth overhead, client storage cost, and many rounds of communications. For instance, Boneh, Mazières, and Popa [13] propose a scheme that achieves $O(1)$ amortized of rounds communication, but using a client cache of size $O(\sqrt{N \log N})$, where N is the total number of data blocks. This client cache is used to store positions of data blocks at remote server. Then, a client can directly accesses the desired blocks, hence reducing the number of client-server interactions. Moreover, the scheme also supports data requests concurrently with shuffling using an additional space at remote server. However, this work suffers from $O(\log N)$ online cost. Recently, Path ORAM has been introduced by Stefanov et al. [100], which organizes server storage as a binary tree of height $L = \lceil \log_2(N) \rceil$. Each node in the tree is a bucket that has Z slots to store logical data blocks. Every slot contains either a real block or a dummy block. Each block is assigned to a path in the binary tree. A query will fetch all blocks on the path which contains the query target, push them into client memory and then put another group of blocks back to the tree. In other words, every read or write request requires the client to download and upload a random path. Although this protocol is simple and efficient, it still incurs relatively high communication overhead, reaching $O(\log^2 N)$.

3.4.3 Hidden vector encryption

To encrypt data, hidden vector encryption (HVE) methods [14, 112] use bilinear groups equipped with bilinear maps and hide attributes in an encrypted vector. Nevertheless, it is extremely costly to compute exponentiation and pairing in a composite-order group. These solutions thus incur prohibitive computation overhead. For example, PaRQ [112] needs almost 500s to answer a range query over a dataset of 10K records. Moreover, they confront high space requirements due to large encrypted vectors. In particular, to support range predicate, every data item is associated with two vectors, each of which requires the space of $O(l \times n)$, where l is the number of attributes and n is the domain of the indexed attribute. HVE approaches cannot cope with the target context which demands low latency responses and being easy to scale up. The latter requires that the system only accepts small storage overhead for scalable protocols. More importantly, high rate of incoming data is not considered by these solutions.

3.4.4 Bucketization

Hacigümüş et al. [50] introduced the bucketization-based approach for range query processing in an untrusted server in 2002. This work simply partitions a query attribute into some continuous buckets. This process is similar to histogram construction. Each bucket is then assigned a random tag (bucket-id), that makes data items in a bucket indistinguishable from another. A clear index may be built over these bucket ids for boosting the query processing. 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. Apparently, the results almost contain false positives, that are eliminated at the post-processing phase by the client. The straightforward solution proposed in [50] not only incurs large false positives, but also lacks an in-depth privacy analysis against various attack scenarios [58]. Intuitively, the larger the bucket, the higher the privacy protection.

Damiani et al. [28] presented an indexing approach that strike a balance between query efficiency and privacy protection. They build a B-tree over plaintexts, then encrypt every record and the B-tree at the node level. This approach does not disclose the content of B-tree to an untrusted server. However, one of the downsides of this approach is that the B-tree scan is now performed by executing a sequence of queries that retrieve tree nodes at progressively deeper level.

A more principled method has been considered by Hore, Mehrotra, and Tsudik [53]. They provide solutions for maximizing privacy protection while keep the number of false positives bounded. Later, a multi-dimensional solution is also presented in [54]. Unfortunately, bucketing approaches lack formal security guarantees. Moreover, the ingestion throughput and scalability dimensions were not considered in these schemes.

3.4.5 Order-preserving encryption

Oder-preserving encryption schemes (OPE) [1, 12, 11, 74, 87, 86] 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. Due to its good performance, order-preserving encryption has been widely used in databases for SQL queries over encrypted data during the last decade such as [41, 59, 106, 88].

One of the first OPE schemes was proposed by Agrawal et al. [1]. It allows the untrusted server to evaluate range queries on the encrypted representation of numeric data. A formal security analysis of such an OPE scheme was later given by Boldyreva et al. [12]. On the other hand, modular order preserving encryption (MOPE) [11] adds a secret offset to the data before it is encrypted. The aim is to shift the ciphertext (in a ring) and to conceal the real location of the encrypted data in their distribution. An improved version of MOPE has been introduced by Mavroforakis et al. [74]. In order to prevent attacks that exploit the max/min values and improve the security of MOPE, this scheme uses fake queries over the gap between the maximum and minimum values.

Popa, Li, and Zeldovich [87] introduced a mutual order-preserving encoding scheme (mOPE) that can provide ideal security guarantee IND-OCPA [12]. This means that attackers learn nothing except for the order of the plaintexts. To achieve such level of security, mOPE accepts a number of interactions between clients and an untrusted server for inserting a new value. In particular, mOPE stores encrypted values on a binary tree at the server. An insert of a new value requires retrieving one node at each level in this tree (starting from the root) to find a right position for the new value. Consequently, the amortized cost of such interaction is $O(\log n)$, where n is the total number of values encrypted.

It is noted that OPE schemes disclose the underlying data distribution, and hence they are vulnerable to statistical attacks. For instance, recently, Naveed, Kamara, and Wright [79] presented a variety of inference attacks on encrypted databases built upon order-preserving encryption schemes. They show that, given just a data dump of an en-

crypted database and public auxiliary information, i.e. publicly available statistics such as census data or hospital statistic, the adversary can successfully recover almost all of the underlying plaintext values from their ciphertexts.

3.4.6 Index-based schemes

Li et al. [66] propose to maintain a secure index, PBtree, on an encrypted dataset for fast range query processing. To achieve that goal, this work creates a binary tree as follows. For each data item x, it builds a set of *prefixes* (or prefix family), $F(x)$, so that $x \in [a, b]$ if and only if $F(x) \cap (F(a) \cup F(b)) \neq \emptyset$. The root contains prefix family of all data items. The algorithm runs recursively, starting from the root and working in a *top-down* fashion. At each node, it splits the prefix families in two subsets, each corresponding to one child. To achieve privacy protection, prefix families at each node are stored in a Bloom Filter [10]. A range query starts from the root of the index. It traverses a child if it has an intersection with the given range. This is repeated recursively until reaching the leaves of the index. Despite the fact that this scheme can achieve fast range query computation, it suffers from high space overhead and false positives. With regards to the security perspective, this scheme only focuses on *non-adaptive adversaries* rather than adaptive ones [25]. This means that it is secure only in applications that allow the users to submit all queries once, and then they shut down. Otherwise, it is not secure against adaptive adversaries that ask some queries first, and then based on the responses returned, they adapt their attacks with more queries later. This limits the applicability of this solution in real-world applications. Moreover, PBtree does not support updates.

To address the downsides of PBtree [66], several sophisticated improvements have been introduced in [64]. To achieve a better security, particularly adaptive *IND-CKA* security model [25], a novel private index IBtree was developed based on a new data structure, Indistinguishable Bloom Filter (IBF) [64]. Similar to PBTree [66], IBtree also requires high space requirements, $O(n \log n)$, where n is the number of indexing data items. IBtree needs at least 10.28GB for a dataset of 5M data items. Furthermore, this approach also needs significant time to build a secure index, reaching to hundreds of minutes. Hence, this might lead to bottlenecks as the system has to ingest high rate of incoming data. Although IBtree supports update operation, it is extremely costly in terms of communication and latency. For example, with a dataset of 5M records, IBtree takes 8.105 minutes for inserting a new record and incurs the communication overhead of 2.239 GB.

On the other hand, Sahin et al. [94] propose a "clear" secure index relying on differential

privacy [34] and semantic security, PINED-RQ, for serving fast range query processing in clouds. The efficiency is enabled by such clear secure index while the security is achieved by using a unification of differential privacy and semantic security. However, since PINED-RQ has to publish data in batches, high rate of incoming data may create potential bottlenecks in the system. Also, PINED-RQ cannot support numerous updates. We delay the discussion of these limitations to Section 3.5.

Poddar, Boelter, and Popa [86] have recently proposed ArxRange that is built on the mOPE protocol [87] and runs on top of MongoDB [20]. It first constructs an index over a set of keywords, and stores at each index node a *garbled circuit* [117, 45] such that the untrusted server is able to secretly compare the query predicate against the keyword hidden at the node. By delegating comparison computation to the server, ArxRange can eliminate interaction overheads in mOPE. However, ArxRange suffers from prohibitive storage cost (e.g., three range indices results in an overhead of 16×). The storage overhead in ArxRange is high since it has to maintain two *garbled circuits* at every tree node. Furthermore, this work only achieves a modest ingestion throughput, e.g., roughly 450 insertions/second with caching.

3.4.7 Secure hardware

Another direction is to use secure hardware for processing encrypted data, such as Cipherbase [8]. However, with a range index, Cipherbase discloses the full ordering information of index keys. Thus, similar to order preserving encryption (OPE), range indexes in Cipherbase provide “similar confidentiality guarantees” [8], and being vulnerable to attacks based on statistical analysis on encrypted data.

3.5 PINED-RQ

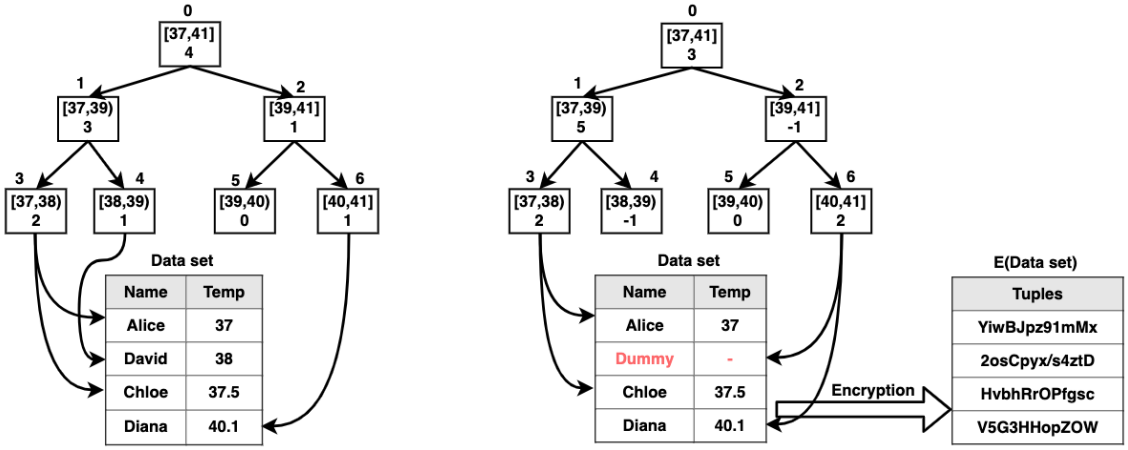
PINED-RQ [94] is one of the solutions that can achieve at the same time efficient range query processing and high security guarantee. The efficiency is enabled by using clear secure index, that relies on differential privacy, while strong privacy protection results from applying semantic security encryption to outsourced data. Moreover, the index in PINED-RQ is "clear" and has a form of B+Tree, it can thus leverage several optimizing techniques that have been developed for B+Tree. Since our study is closely related to PINED-RQ [94], we briefly introduce it in this section. Given a dataset at a trusted

component such as data owner, PINED-RQ builds an index (hierarchy of histograms) over this dataset, then it perturbs the index by using Laplace noise to get a differentially private index. Finally, this secure index is published to the cloud along with the encrypted dataset for serving range queries.

3.5.1 Index construction

Typically, there are two primary steps to build a private index over an attribute A_q, denoted $\mathcal{I}(A_q)$, in PINED-RQ.

Step 1 - Building an index. given a data set at the collector, a clear index is



(a) Clear index (b) Perturbed index

Figure 3.4 – Sample PINED-RQ index

constructed based on a B+Tree. In PINED-RQ, the set of all nodes is defined as a *histogram* covering the domain of an indexed attribute. For example, the participants' body temperature (Temp) is used to build histograms as illustrated in Figure 3.4a. Each leaf node has the count that represents the number of records falling within its interval. It also keeps pointers to these 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. To make the index private from the adversary, PINED-RQ takes advantage of differential privacy model. In particular, Laplace mechanism [36] is used to perturb the index. All counts in the index are then independently

perturbed by Laplace noise [36]. Nevertheless, the noise generated from Laplace method may be positive or negative, thereby after this step, the count of a node may increase or decrease, respectively. As shown in Figure 3.4b, the count of node 4 changes from 1 to -1 while the count of node 6 changes from 1 to 2. These 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 as a leaf node receives positive noise, otherwise, if a leaf node receives negative noise, real records are removed from the dataset. To ensure these removed records are to be involved in query processing, Sahin et al. [94] propose to use *overflow arrays*. Each leaf node is allocated an overflow array for containing all removed records of that leaf. These overflow arrays are later filled with dummy records so that they have the same size. Such approach thus allows concealing removed records from the adversary. As illustrated in Figure 3.4b, the record (David) belonging to node 4 is removed from data set while one dummy record is added and linked to node 6. To determine the size for overflow arrays, PINED-RQ uses the inverse of the cumulative distribution function (CDF) of the Laplace distribution with an adjustable probability, δ.

Lastly, PINED-RQ uses a semantic encryption scheme (e.g., AES in CBC mode) to encrypt the dataset and overflow arrays before publishing them to the cloud along with the perturbed index.

3.5.2 Query processing

In PINED-RQ, a client first sends a range query to the collector. It is partially answered according to local un-indexed data. The query is then redirected to the cloud to be evaluated on indexed data.

At the cloud, 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.

3.5.3 Privacy protection

PINED-RQ is designed to satisfy $(\epsilon, \delta)_n$-Probabilistic-SIM-CDP privacy model [94], as presented in Section 3.3, that is a variant of differential privacy [35]. Intuitively, this

model results from the introduction of the encryption and the overflow arrays to the index building process.

3.5.4 Limitations

Although PINED-RQ can provide very good performance in terms of index scanning and high level of security, it still has some downsides.

First, since an update directly to such published indexes would violate differential privacy [34, 35], PINED-RQ cannot support live updates. Additionally, PINED-RQ is also reluctant to publish very small data sets since the aggregation noise would destroy indexes' utility. These properties make PINED-RQ bottlenecked at the collector as data comes at a high speed.

Second, PINED-RQ fail to support numerous updates, especially those of the same individual. That is, this scheme needs to spend a piece of the fixed privacy budget for an update to the same individual. As a result, for applications where an individual has a huge number of updates, e.g., in a patient management system, a patient has multiple records, the system will run out of privacy budget and shut down soon.

3.6 Conclusion

This chapter mainly focuses on background knowledge and related works. More specifically, we present basic concepts in database as well as privacy tools. We then consider the related works on secure range query processing over encrypted and their drawbacks. None of existing schemes can handle the situation of high speed of incoming data and provides a scalable solution for modern systems. These drawbacks mainly lead to our contributions in this dissertation. Lastly, we briefly describe PINED-RQ, especially the main steps necessary for building a secure index over a dataset, and its main drawbacks.

Chapter 4

PINED-RQ++

Abstract: In this chapter we address the problem of privacy-preserving range query processing with regards to the context of the high speed of incoming data. Several solutions have been proposed for secure range query processing, however, they become inefficient or impractical for the target context. In particular, prior schemes often suffer from performance issues such as overload or bottleneck. To address these problems, we develop an extension of PINED-RQ, named `PINED-RQ++`. More precisely, we seek to adapt PINED-RQ to the target context by using a notion of *index template*. Such index template allows the collector to process incoming data *on the fly* while still maintaining private indexes for published data. By using the index template, we can reverse the process of constructing the PINED-RQ index while incoming data can be processed and forwarded to the cloud as soon as possible. Considering the security dimension, we introduce a noise management strategy and a complementary data structure to the reversing process. These modifications allow `PINED-RQ++` to avoid privacy leaks of the reversing process. As a result, `PINED-RQ++` can eliminate potential bottlenecks in the system when data arrives at a high speed while high security is still guaranteed. Furthermore, we also develop a parallel `PINED-RQ++`, with the aim of enhancing the low ingestion throughput caused by the index template building process. We implemented both the non-parallel and the parallel version of `PINED-RQ++`, evaluated, and compared them to PINED-RQ. The experimental results show that our extension outperforms PINED-RQ in various metrics. We also demonstrate experimentally that the parallel extension significantly improves the ingestion throughput of the system in general.

4.1 Introduction

A wide range of applications, such as online survey management and disease tracking, require the capability of performing quick analytics on high-speed, continuously generated data coming from various data sources such as web-based systems and mobile devices. Needless to say, such systems need to efficiently support both fast data consumption and low-latency queries.

For example, to reduce the impact of seasonal epidemics (e.g., H1N1 influenza), early detection of spatial spread of the epidemics could help alleviate severe consequences. It is thus important to track and predict the spread of such diseases in the population. To do that, an individual can interact with a website or mobile application to report personal data (e.g., age, phone, sex, symptoms, travel plan, social network account, etc.), to be utilized for real-time predicting analyses. These systems usually run in very short periods, a few days or weeks after an epidemic emerges. Similarly, a university wants to get their student opinions university services at the end of every semester. To have quick responses for changes in the next semester, such survey can be performed by a web-based system.

Since most of these systems need significant computing capacity in very short periods, purchasing and maintaining local servers would be wasteful. Cloud computing with on-demand capacity and a *pay-as-you-go* model deserves to be used for managing and exploiting collected data. However, the problem is that cloud computing suffers from security issues, e.g., sensitive information can be exploited by cloud's administrators. Encrypting outsourced data is a common solution to handle privacy issues in clouds. In this chapter, we focus on secure range queries over encrypted data with respect to the high rate of incoming data. In particular, such situation creates bottlenecks in prior schemes.

To solve the drawback of existing works, we aim at adapting PINED-RQ [94] to the target context. As presented in Section 3.5, PINED-RQ maintains a "clear" secure index for published data such that range queries can be privately evaluated on this index. Our choice is motivated by the fact that PINED-RQ provides a high level of security while it offers significantly faster range query processing and requires less storage space as compared to its counterparts [32, 66, 64, 86]. Nonetheless, like many prior schemes, PINED-RQ has to publish data in batches and partially processes data at a trusted component (e.g., collector). Consequently, bottlenecks may occur as incoming data and query requests arrive at a high rate. Therefore, to adapt PINED-RQ to the target context, we aim to shift heavy workload from the collector to the cloud, that is able to provide on-

demand capacity. In particular, instead of publishing data in batches, when a new record arrives, the collector immediately sends it to the cloud. The challenge to this approach is how to build the PINED-RQ index for the new records that are previously transferred to the untrusted cloud.

In this chapter, we propose `PINED-RQ++`, an extension of PINED-RQ, to mainly prevent potential bottlenecks at the collector while ensuring a secure index for new data. The key idea behind our prototype is to reverse the process of constructing the PINED-RQ index. This allows the sending of new data to the cloud as soon as possible without sacrificing privacy protection. The experimental results give very good performance, e.g., the publishing time of the NASA dataset ($\sim$0.5M records) [78] is reduced up to $\sim$35x while maximum data rate at the collector experiences a reduction of up to $\sim$2.7x. Additionally, our solution eliminates almost query computation at the collector, making the system more scalable. The contributions of this chapter are as follows.

1. We introduce a notion of index template within PINED-RQ to support high rate of incoming data.
2. We propose a mechanism, `PINED-RQ++`, for updating the index template while still retaining privacy protection for published data comparable to PINED-RQ.
3. We also develop a parallel version of `PINED-RQ++` to improve the ingestion throughput of the system.
4. We provide a thorough privacy analysis to show the security protection of `PINED-RQ++`.
5. We implement the non-parallel and the parallel version of `PINED-RQ++` to show the superiority of them compared with PINED-RQ.

4.2 Problem definition

The dataset stored at the collector is a relation $D(A_1, \ldots, A_d)$, where A_i is an attribute. Queries are non-aggregate one-dimensional range queries. A query $\mathcal{Q}$ is evaluated over the attribute A_q of D.

In this chapter, we mainly focus on the metrics related to the congestion of the system, namely publishing time and network traffic. The former measures how long a publication is sent to clouds while the latter records the total size of data transferred per second at the collector.

4.2.1 Overview architecture of PINED-RQ++

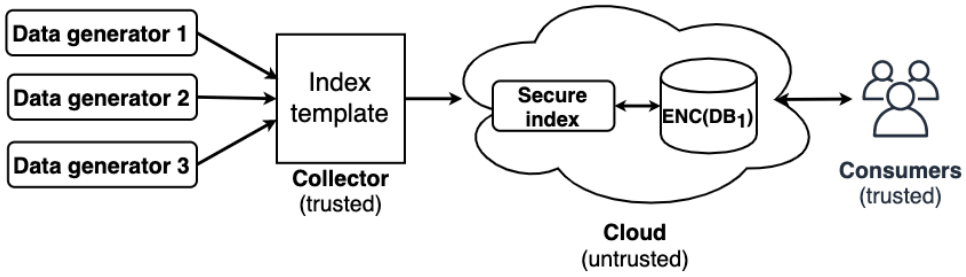


Figure 4.1 – Architecture of PINED-RQ++

We focus on the architecture as depicted in Figure 4.1. Data generators produce raw data and send them to a collector. The data is then pre-processed prior to being sent to a cloud. A consumer poses range queries to the cloud. In this architecture, we assume that the cloud is *honest-but-curious* while the other components are trusted.

At the beginning, an index template is built at the collector. Whenever a new record arrives, the index template is updated with that record. Next, the record is encrypted and forwarded to the cloud. When the index template is published at a later time, the cloud associates it with unindexed data to produce a secure index as described in Section 3.5. The collector then initiates a new index template for future incoming data.

A query is only processed at the cloud which holds both indexed and unindexed data at time. As a result, as a consumer issues a query, it is first evaluated on indexed data (as in PINED-RQ's query processing [94]), the result of this evaluation and all the un-indexed data are returned to the consumer. Finally, the consumer decrypts and filters the returned data for the final results.

4.2.2 Threat model

Our architecture targets attackers to the cloud. Hence, our threat model assumes that attackers do not control or observe data and execution on the trusted components. However, they may access the cloud that consists of encrypted data sets and secure indices.

We consider the *honest-but-curious* model [46]. 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 data sets or query results. In this chapter, we consider two types of attackers, *online* and *offline* attackers. The offline attacker tries to steal a copy of encrypted data sets and secure indices. Meanwhile, the online attacker can record and

observe any information available at the cloud or being exchanged between the cloud and the trusted components to deduce anything in a computationally-feasible way. The observed information includes all changes to the data sets, all in-memory state, and all queries at any time points for any amount of time. Furthermore, we assume that attackers does not have good knowledge about the data distribution of the incoming time of real data.

4.3 PINED-RQ++

This section describes PINED-RQ++, an extension of PINED-RQ, that can cope with the high rate context. The main goal is to process incoming data one by one at the collector instead of publishing data in large batches. More precisely, when a record arrives, the collector pre-processes the record and sends it to the cloud immediately. Such an approach relieves the heavy burden of computation on the collector at publishing time, thereby avoiding potential bottlenecks at this component. However, the difficulty for this approach is how to build the PINED-RQ index over attribute A_q, denoted $\mathcal{I}(A_q)$, on new records that are immediately transferred to the untrusted cloud.

To this end, we aim at reversing the process of constructing the PINED-RQ index at the collector while at the same time keeping privacy protection as strong as possible. To achieve this goal, we introduce a notion of *index template* based on the PINED-RQ index structure. The index template stores on the collector the information necessary for building $\mathcal{I}(A_q)$ at publishing time. Such information includes the Laplace noise and real bin counts being updated according to incoming data. In other words, the collector starts by initiating an index template and independently adds the Laplace noise to its bins. Then, whenever a record arrives at the collector, the index template is updated with that record. The record is then encrypted and forwarded to the cloud. When the updated index template is published, the cloud associates it with un-indexed data to achieve a complete $\mathcal{I}(A_q)$.

In addition to the index template, we introduce a new noise management mechanism as well as a complimentary data structure, named *matching table*. The former secretly manages the addition of dummy records and the deletion of removed records. Meanwhile, the latter privately maintains pointers between the index template and un-indexed records until the index template is published to the cloud. Note that we only focus on the append-only context (i.e. redesigning the *CREATE* and *INSERT* functions of PINED-RQ) in this

extension.

4.3.1 Index template

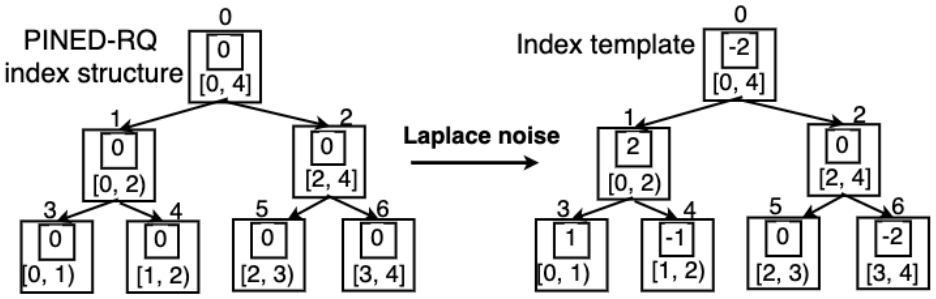


Figure 4.2 – Sample index template

An index template, denoted $\mathcal{IT}(A_q)$, over the queriable attribute A_q of a dataset $\mathcal{D}$, is computed from the domain of $\mathcal{D}$. Basically, the process of building an index template is the same as that of PINED-RQ index (see Section 3.5). The main difference is that the count variables of an index template only contain the Laplace noise and its leaves have no pointers to records (see Figure 4.2). Therefore, to achieve an index $\mathcal{I}(A_q)$ from an index template $\mathcal{IT}(A_q)$, the count variables and pointers of the $\mathcal{IT}(A_q)$ need to be updated and maintained during a *time interval*, which is defined as the period from when an index template is initiated to when it is published.

Recall that due to the injected noise, the process of building a secure index in PINED-RQ may require adding some dummy records to $\mathcal{D}$ and removing real ones from $\mathcal{D}$. `PINED-RQ++` also needs to guarantee such tasks to be performed during a time interval. Nonetheless, this poses several challenges to the proposed approach, for instance, how to publish dummy records to the cloud or how to ensure that pointers between leaves and the new data do not disclose sensitive information to attackers. In Sections 4.3.2 and 4.3.3, we discuss these challenges as well as our solutions.

4.3.2 Matching table

When an index template is published, the cloud should associate it with un-indexed data to achieve a complete PINED-RQ index for those data. To prepare for such association, the collector needs to keep the pointers between un-indexed data and leaves during a time interval. A straightforward solution is to mark the ciphertext of a new record by the

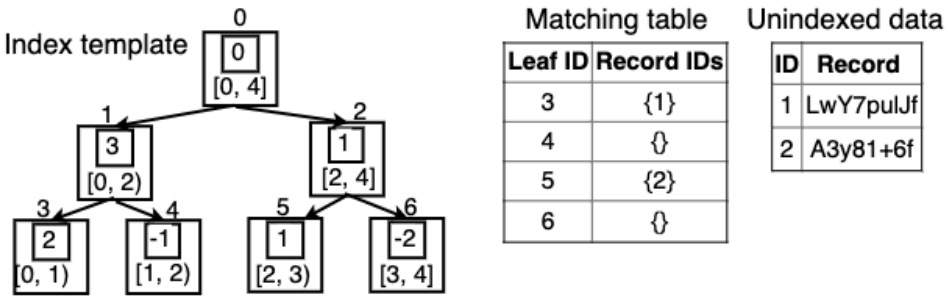


Figure 4.3 – Index template and its matching table

identifier (id) of the leaf node to which the record belongs, and send the marked ciphertext to the cloud. Later, the cloud can rebuild pointers from marked ciphertexts when the index template is published. However, such marked ciphertexts reveal real pointers between un-indexed data and leaves to the cloud during a time interval. `PINED-RQ++` consequently discloses more extra information, e.g., the actual distribution of the incoming time of real data, when compared to PINED-RQ.

To prevent the leakage, we propose to use *unique random numbers* which are viewed as temporary ids of records and a *matching table*, as depicted in Figure 4.3. A matching table, denoted $\mathcal{MT}$, can be computed directly from $\mathcal{IT}(A_q)$. In the matching table, the first column stores leaves' id while each row of the second column holds the temporary id of records that belong to the corresponding leaf node. For instance, record 1 belongs to node 3 while record 2 belongs to node 5. With this approach, at publishing time, a matching table will be created at the collector. When a record is available, the collector encrypts it, generates a unique random number, and sends the pair of <*random number, ciphertext*> to the cloud. A copy of the generated number is also stored in the corresponding row in the matching table at the collector. The randomness of such number guarantees that no useful information about the published data will be leaked to attackers.

At the end of each time interval, the matching table will be sent to the cloud along with the corresponding index template. Based on these data structures, the cloud can easily reconstruct the pointers between the un-indexed data and the index template, denoted as *matching process*. Specifically, the matching process starts by looping over the published matching table. At each row, it retrieves a leaf id and a set of record ids in the second column, finds encrypted records having the corresponding record ids in the un-indexed dataset. Then, the pointers between the corresponding leaf and the found encrypted records are formed. After matching, a complete index for the new dataset is

achieved, and the matching table and records' id are destroyed.

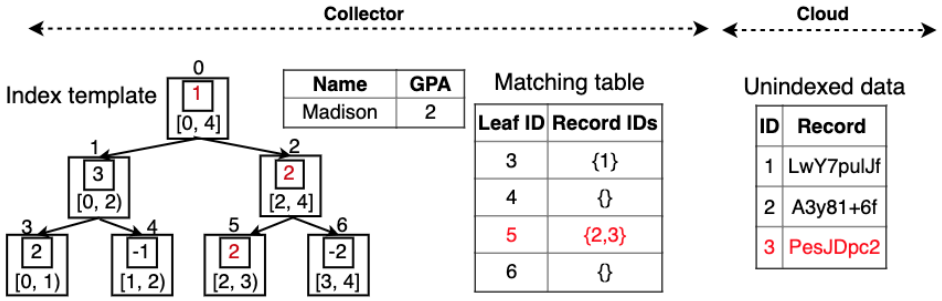
4.3.3 Noise management

As presented in Section 3.5, the Laplace noise primarily helps to protect the privacy of PINED-RQ index. However, the injected noise may result in the insertion and deletion of some dummy and real records, respectively. In other words, PINED-RQ will randomly remove c records from dataset as a leaf node receives negative noise -c. Otherwise, c dummy records will randomly be added to dataset as it receives a positive noise c. One challenge to `PINED-RQ++` is that the collector initiates and perturbs the index template without any existing data. This means no record is available to be deleted in case of negative noise. Also, when leaves receive positive noise, the collector can generate dummy records. However, the question is how to publish these dummy records? To this end, we present two approaches as follows.

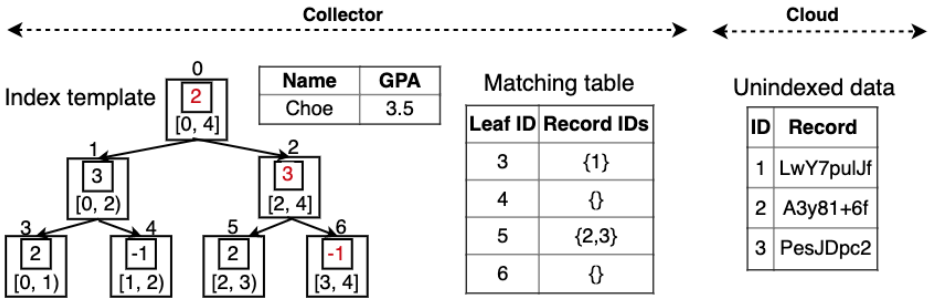
a) Positive noise. Indeed, one may suppose that the collector can immediately generate and send dummy records to the cloud at any random time points within a time interval. Nevertheless, the arrival of all dummy records at the same time could be unsafe since the adversary can exploit the distribution of such arrivals. On the other hand, we can randomly release dummy records over a time interval. It is, however, true that the privacy would be leaked as a dummy record arrives at a chosen time at which real records are unlikely. To avoid such case, we propose to send dummy data according to the actual distribution of the sending time of the real records. This means that the probability of sending a dummy record at each time point is computed from the distribution of real records. With this approach, when a pair of <*random number, ciphertext*> comes to the cloud, the adversary cannot distinguish which pair is dummy or real data.

b) Negative noise. As a leaf node initially receives negative noise c, the collector removes the first c records (when they arrive) of that leaf node. To keep the removals of real records secret from the adversary, we need to hold removed records at the collector. At publishing time, the collector generates overflow arrays and randomly fills all overflow arrays with dummy and removed records. Finally, the collector sends the overflow arrays to the cloud along with the index template and matching table. Notably, the deletion of records only occurs at the collector and the adversary does not know which nodes receive negative noise, hence the privacy of such movements is preserved.

4.3.4 Index template update management



(a) New record belongs to positive leaf



(b) New record belongs to negative leaf

Figure 4.4 – Updating index template and matching table

We now describe how to ensure the counts of `PINED-RQ++` index template are the same as those of PINED-RQ index when the index template is published. In PINED-RQ, the count of a leaf node represents the number of records falling within its interval. The count of internal nodes and the root is a summation of their children's counts. All counts are then perturbed by the Laplace noise. In contrast, an index template only contains noise at first and it will increase its counts as a record arrives at the collector.

Basically, when a record arrives at the collector, the leaf node to which the record belongs is determined. Then, the count of that leaf node and all its ancestors will be increased by 1. As shown in Figure 4.4a, as the new record <Madison, 2> has GPA lying within the interval of node 5, the count of node 5 and all its ancestors (node 2 and node 0) are increased to 2, 2 and 1, respectively. On the other hand, if a new record belongs to a negative leaf, it is kept at the collector as removed. For instance, e.g., since the record <Chloe, 3.5> belongs to node 6 (see Figure 4.4b), it is kept at the collector and the counts in the corresponding path are also increased by one.

4.3.5 Query processing strategy

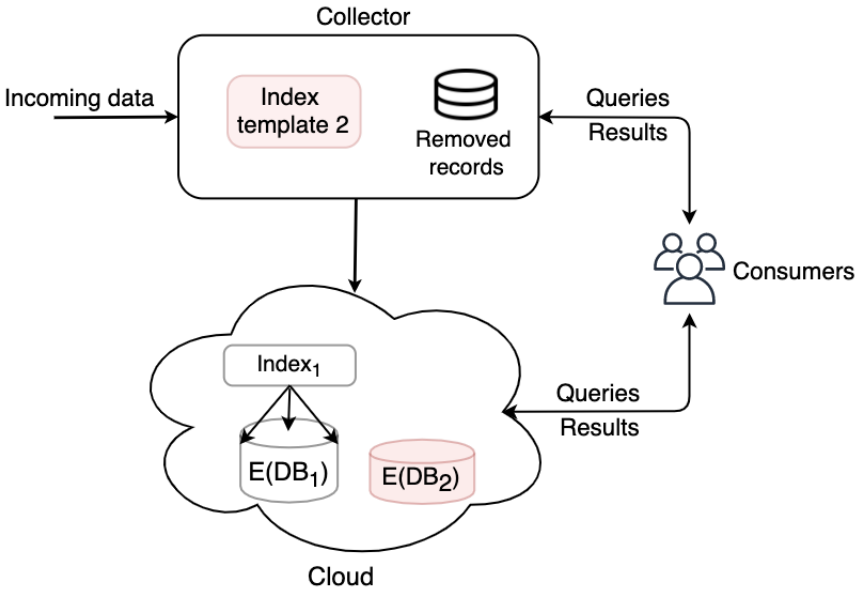


Figure 4.5 – Query processing strategy in PINED-RQ++

In PINED-RQ++, both unindexed and indexed data are stored at the cloud, as shown in Figure 4.5. The collector only keeps removed records of the current publication. Hence, when the consumer issues a query, it is sent to both collector and cloud. At the cloud, the query is first evaluated on indexed data (as in Section 3.5), the result of this evaluation and all the un-indexed data are returned to the consumer. In parallel, at the collector, the removed records overlapping the query range are also returned to the consumer. The consumer finally decrypts and filters the returned data for the final results. It is worth noting that a query in PINED-RQ++ is only executed on removed records at the collector instead of the whole unpublished dataset as in PINED-RQ. Hence, PINED-RQ++ enables to relieve the burden of query processing on the collector, especially when it needs to serve a high frequency of consuming requests.

4.3.6 PINED-RQ++'s correctness

We analyze the correctness of PINED-RQ++ by comparing with its original version. First, we note that the indexes of the two prototypes have the same structure as stated in Section 4.3.1. Second, the main difference between the two prototypes is their noisy counts. In particular, in PINED-RQ, the noisy count of a bin b_i is $p_i = c_i + v$, where c_i

is the number of real records of $\mathcal{D}$ lying into b_i and v is the Laplace noise. PINED-RQ then applies the constrained inference technique [52] on p_i to get $\overline{p}_i$ before an index is published to the cloud. Meanwhile, `PINED-RQ++` only guarantees the count of bin b_i is p_i at publishing time. To achieve that, it initially adds v to a new index template (see Section 4.3.1) and c_i is incrementally updated during a time interval (see Section 4.3.4). Thus, at publishing time, `PINED-RQ++` ensures that the noisy count of a bin b_i is $p_i = c_i + v$ instead of $\overline{p}_i$. Third, PINED-RQ builds pointers between the leaves of an index and the records of $\mathcal{D}$ at the collector while those are constructed at the cloud based on the matching table in `PINED-RQ++`. Fourth, PINED-RQ moves the records that belong to the leaf receiving negative noise to the corresponding overflow array, and adds dummy records to $\mathcal{D}$ as a leaf receives positive noise. These operations are performed during a time interval using the noise management mechanism in `PINED-RQ++`, as presented in Section 4.3.3.

Apparently, the index achieved at the cloud of the two prototypes is almost the same. The minor difference comes from the fact that `PINED-RQ++` does not apply the constrained inference technique on p_i to get $\overline{p}_i$, that may slightly improve the utility of the index. However, we will see in Section 4.7.5 that the quality of the index of the two prototypes is very competitive. We also emphasize that not applying the constrained inference technique on bin counts has no impact on the privacy protection of the `PINED-RQ++` index.

4.4 Parallel `PINED-RQ++`

Recall that `PINED-RQ++` attempts to avoid bottlenecks by processing incoming data one by one. Nonetheless, sequentially processing arrivals greatly diminishes the ingestion throughput at the collector. To support tens of thousands of records per second, we come with a new architecture for `PINED-RQ++`, named parallel `PINED-RQ++`. The main goal is to shift the process of updating index template to a set of *shared-nothing* machines [1] as much as possible. This approach will enable the new architecture to ingest incoming data faster and be more scalable than the non-parallel `PINED-RQ++`. To this purpose, we first decompose the updating process at the collector into components. We then orchestrate them to achieve the parallel `PINED-RQ++`.

1. We use the shared-nothing architecture is because it is highly scalable. This means the system can scale up to multiple nodes easily.

4.4.1 Workflow at collector

After the collector initiate a new index template, new data sequentially passes the following components:

1. **Parser** transforms raw data into a pre-defined format.
2. **Checker** buffers the parsed record at the collector as the indexed attribute of that record lies within the range of a negative leaf. The index template is also updated with that record. Otherwise, the record is forwarded to the next component.
3. **Enricher** adds a unique random id to the record.
4. **Updater** updates the index template and matching table based on the record.
5. **Encrypter** uses an encryption scheme, e.g., AES in CBC mode, to encrypt the record, and obtains an e-record (encrypted record). Finally, the encrypter sends the pair of <id, e-record> to the cloud.

Note that the parser and the checker rely on the index template that is a *shared* data structure. Indeed, the checker repeatedly reads count variables for the checking task while the updater modifies the corresponding count variables due to the arrival of new records. Additionally, to perform the checking task, the checker relies on the parser for obtaining the value of the indexed attribute. Hence, both parser and checker are designed to sequentially run in the parallel `PINED-RQ++`.

4.4.2 Architecture of parallel `PINED-RQ++`

We assume that the collector of the parallel `PINED-RQ++` runs on a small cluster of commodity machines. At the collector, one (and only one) node runs Dispatcher (D) and all worker nodes in the cluster run a Computing Node (CN). The dispatcher plays the role of a coordinator. Only the dispatcher can start a new publication by initiating a new Index Template (IT) and sending the clone (CL) of the index template to all available computing nodes. A clone only contains zero counts. During a time interval, the dispatcher parses, checks, enriches new data, and sends them to computing nodes according to the round robbin approach. Dummy records are also generated at the dispatcher and released based on pre-defined strategy as discussed in Section 4.3.3. Computing nodes take the outputs from the dispatcher and generate a local Matching Table (MT) according to the received clone. When a new record arrives, a computing node updates the clone and the matching table, encrypts the record, and sends the

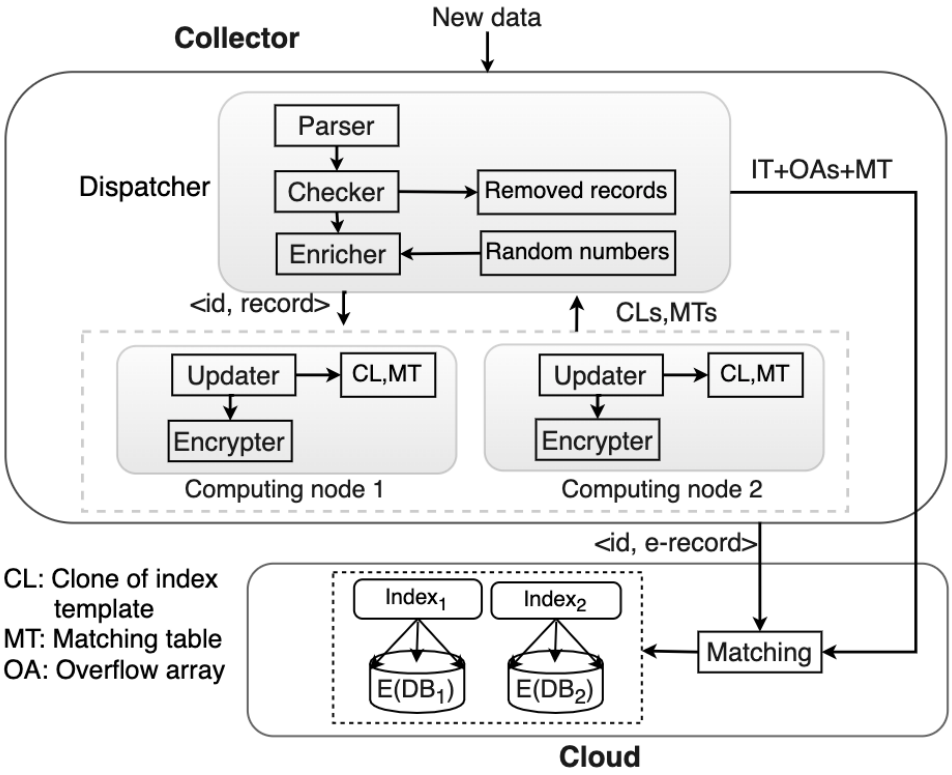
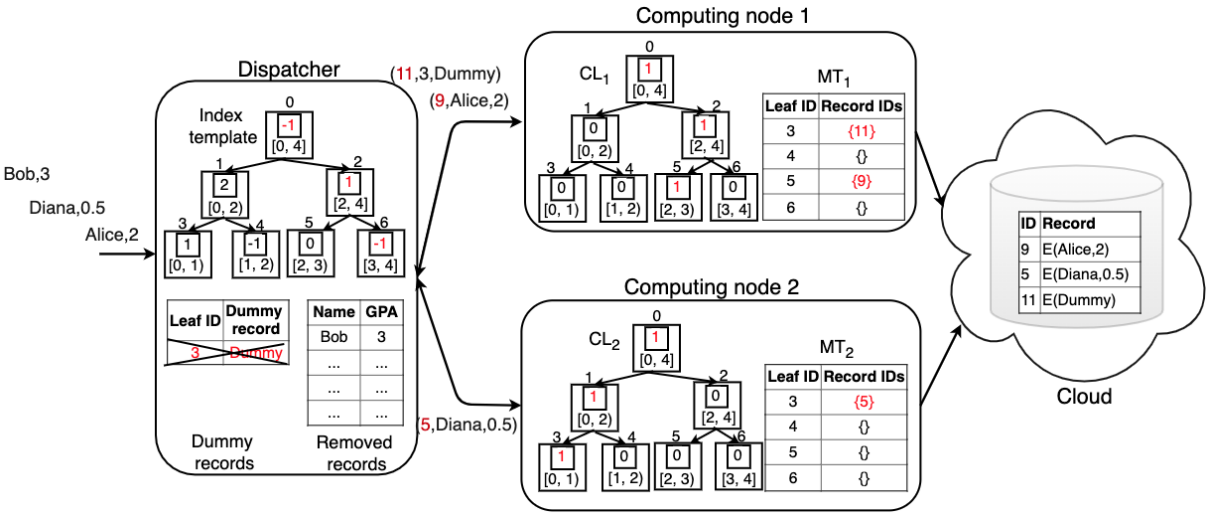


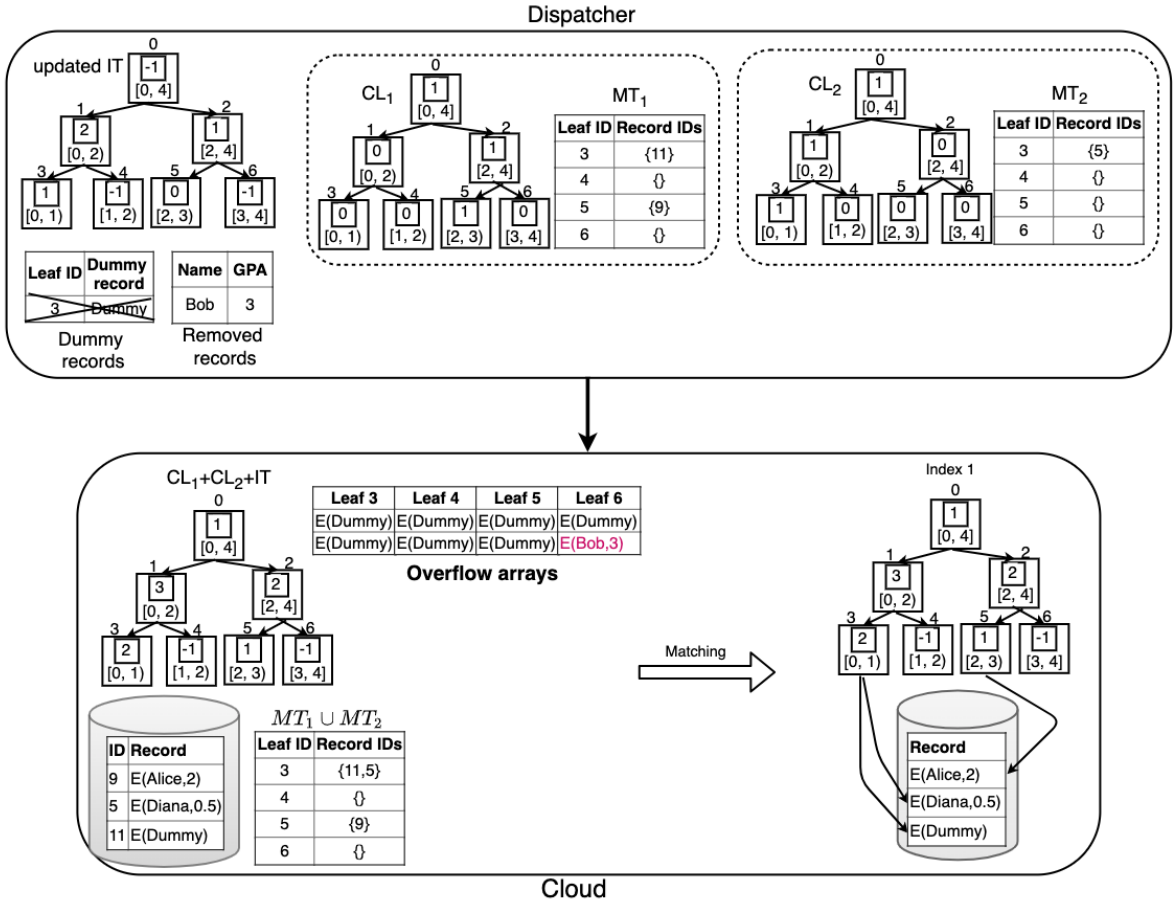
Figure 4.6 – Architecture of parallel PINED-RQ++

encrypted version of that record to the cloud. Notice that the dispatcher and computing nodes can coexist on the same node. In order to demonstrate how data is processed at the collector and transported to the cloud, Figure 4.6 shows the composition of the parallel PINED-RQ++'s collector running on three nodes (one dispatcher and two computing nodes) while Figure 4.7 gives a concrete example with three real records and one dummy record.

Dispatcher (D). At the beginning of a time interval, the dispatcher initiates an IT, unique random numbers, and dummy records. It then sends the clone (CL) of IT to all available computing nodes. The dispatcher schedules the time points at which the dummy records will be released, as stated in Section 4.3.3. During a time interval, new data are passed throughout the parser and the checker. If new records belong to a negative leaf, they will be kept at the dispatcher. The IT is updated based on that record. Otherwise, the records are then sent to computing nodes in a round-robin fashion after receiving unique ids from the enricher. As shown in Figure 4.7a, two records (Alice, 2) and (Diana, 0.5) come first and in turn fall within leaves 5 and 3 which have positive



(a) During a time interval



(b) At publishing time

Figure 4.7 – Example of processing incoming data in parallel `PINED-RQ++`. The original index template is presented in Figure 4.2

noise. They are enriched by unique ids, namely 9 and 5, respectively, before being sent to Computing node 1 and 2. In contrast, record (Bob, 3) belongs to leaf 6 which contains negative noise (-1), and hence it is kept at the dispatcher. On the other hand, assume one dummy record is released after the arrivals of these real records, it is also assigned by an id prior to being sent to Computing node 1.

At the end of each time interval, the dispatcher sends a *publishing* message to all available computing nodes and waits for the CLs (clones) and MTs (matching tables) returned from these nodes. Upon receiving these data structures, the dispatcher merges them with the local IT to achieve a complete IT and MT. The dispatcher generates Overflow Arrays (OAs) and randomly inserts removed records into OAs before sending the complete IT, MT, and OAs to the cloud. As demonstrated in Figure 4.4, assume the overflow arrays' size is 2 records, the dispatcher then combines its data (e.g., updated IT and removed records) with data from all computing nodes (e.g., CLs and MTs) before sending the combination to the cloud. Finally, the dispatcher sends a *done* message to all computing nodes and initiates a new publication. The merging process may take time, especially when datasets and OAs are large. Hence, during the meantime, the dispatcher still ingests incoming data and sends them to computing nodes. Such data will be buffered at computing nodes until a new publication is initiated.

Computing Node (CN). At the beginning of a time interval, or when a computing node receives a *done* message, it receives the CL from the dispatcher, and initiates a new MT. During a time interval, when a new record arrives, the computing node updates the local CL and MT. Then it encrypts the record and sends the encrypted record to the cloud. As depicted in Figure 4.7a, the matching tables and clones at two computing nodes are updated with incoming records from the dispatcher. Note that the CLs are used only for keeping counts on real records, thereby dummy records do not cause any updates to these CLs, but the MTs. For instance, Computing node 1 only inserts the id of the incoming dummy record (11,3,Dummy) to the first row of its MT since this dummy record is generated for leaf 3, but does not update the CL. Finally, they are encrypted and sent to the cloud.

When the computing node receives a *publishing* message from the dispatcher, it sends the updated CL and MT back to the dispatcher. While waiting for a new publication, all incoming records will be buffered at computing nodes.

Cloud. During a time interval, the cloud stores arriving encrypted records on disk due to their high space requirements. When the cloud receives IT, MT, and OAs from the dispatcher, the matching process is triggered, as described in Section 4.3.2. In particular, the cloud reads all records from disk, performs the matching, shuffles the dataset, and writes them back to disk. The output of that process is a secure index over the encrypted dataset, as shown in Figure 4.7b. Although this study considers persisting incoming records on disks, non-volatile memory (NVM) can be used to temporarily store the data. Such approach would greatly shorten the time for the matching process since it avoids heavy I/O accesses to disk drives. Obliviously, such choice mainly relies on specific applications and resource availability.

4.5 Privacy leak and TEX-PINED-RQ++

4.5.1 Privacy leak of real timestamps

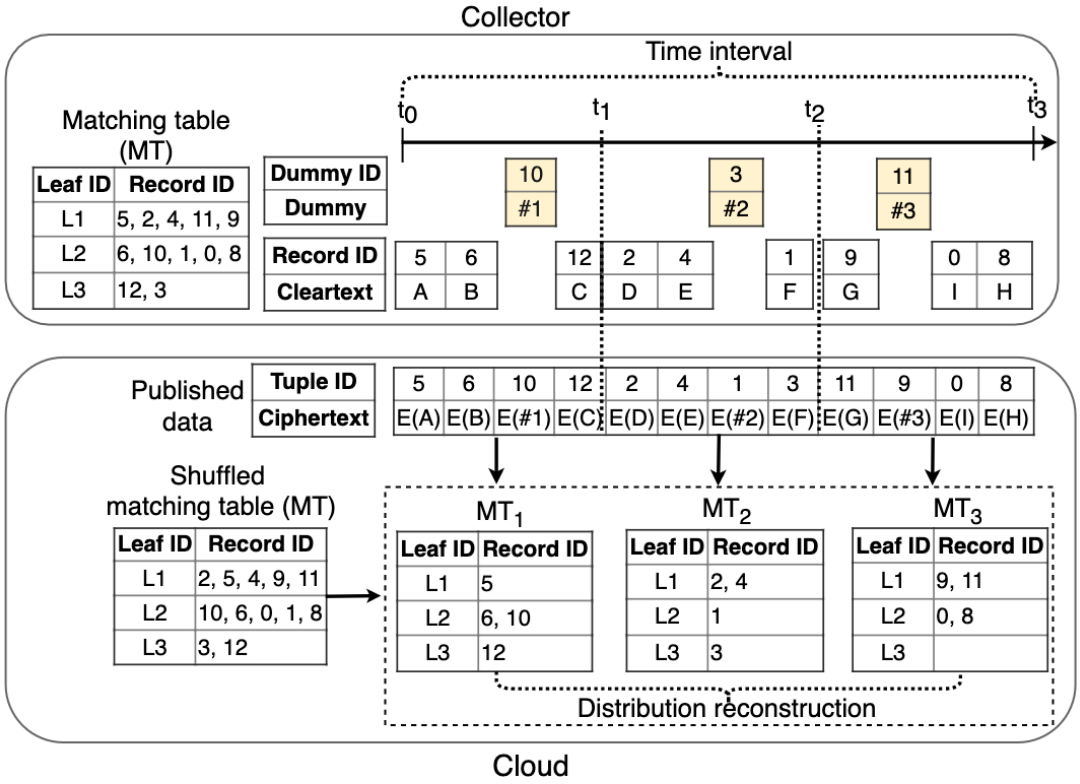


Figure 4.8 – Privacy leak of timestamps

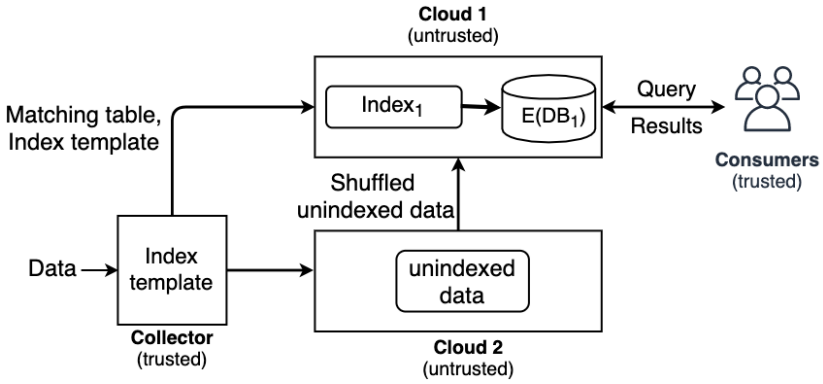


Figure 4.9 – TEX-PINED-RQ++'s architecture for addressing leakages introduced by timestamps of published data

With PINED-RQ++, we find that it incurs extra privacy leaks due to the timestamps of published data. In other words, when a matching table is sent to the cloud at the end of each time interval, based on that matching table and un-indexed data, the adversary can infer the distribution of the incoming time (timestamp) of real data. To demonstrate how attackers can exploit such information, we use an example as in Figure 4.8. Assume that the data arrival to the collector is uniform over a time interval $[t_0, t_3]$. We thus uniformly release dummy records over that period. At the end of the time interval (t_3), when the collector shuffles ids of the matching table and releases the shuffled matching table to the cloud, based on the timestamps of incoming data, attackers can easily derive *sub-matching tables* (MT_1, MT_2, and MT_3) as well as their corresponding data part. This means that MT_1, MT_2, and MT_3 in turn contain ids of data parts belonging to the time interval $[t_0, t_1)$, $[t_1, t_2)$, and $[t_2, t_3]$. Unfortunately, these parts may not contain enough dummy records, and hence attackers can infer sensitive information from their sub-matching tables.

Note that such leakage only occurs during the matching process, from when the matching table arrives at the cloud to when the matching table is destroyed (after the matching process is finished). Thus, this disclosure can only be exploited by the *online* attackers who can observe any information available at the cloud at any time points. PINED-RQ++ provides still strong privacy protection in case of *offline* attackers who try to steal a copy of encrypted data sets and secure indices, but cannot observe data exchanged and computations at the cloud.

4.5.2 TEX-PINED-RQ++

To remedy the security issue caused by real timestamps, we present **TEX-PINED-RQ++**, a new scheme that highly protects data privacy against **T**imestamps **EX**ploitation in **PINED-RQ++**. This solution consists using a complimentary cloud provider (Cloud 2) for *perturbing* real timestamps of un-indexed data, as depicted in Figure 4.9. We assume that Cloud 2 must not collude with the existing one (Cloud 1). The model of using two non-colluding semi-honest cloud providers is not new and has already been widely used in many works such as [37, 76, 112]. To adopt this model, during a time interval, un-indexed data is sent to Cloud 2 instead of Cloud 1. At the end of each time interval, the content of the current matching table is shuffled and sent to Cloud 1 along with the index template. In parallel, Cloud 2 shuffles and sends all un-indexed data to Cloud 1 for a matching process. With this approach, there is no sensitive information to be revealed to both clouds, as analyzed in Section 4.6.

4.6 Privacy analysis

We now analyze the privacy protection of the non-parallel and the parallel version of PINED-RQ++ in comparison with PINED-RQ. It is noted that since PINED-RQ++ is designed to release data during a time interval, they will reveal pairs of <*random number, ciphertext*> to the cloud over that period. In addition, since the collector immediately sends incoming data to the cloud, attackers can glean useful information from the timestamps as discussed in Section 4.5.1.

4.6.1 Analysis of timestamps

We consider the disclosure of timestamps of real data to the cloud. Particularly, we examine the two following models.

Single-cloud model. As discussed in Section 4.5.1, in case of *single-cloud* model, attackers can easily infer sensitive information by using the timestamp of un-indexed data and the corresponding matching table. Such leakage can be exploited by online attackers, but it is unavailable to offline ones.

Two non-colluding clouds model. With such a model, we consider the disclo-

sure of the two clouds. First, during a time interval, Cloud 2 will receive pairs of encrypted data, and obtains the corresponding timestamps. However, Cloud 2 cannot glean any useful information from these timestamps without the matching table. Second, Cloud 1 will receive an index template and matching table from the collector at publishing time. In parallel, Cloud 1 also receives the corresponding dataset from Cloud 2. The matching table (from the collector) and dataset (from Cloud 2) are shuffled before arriving Cloud 1. This means that the timestamps of published data are shuffled before arriving at Cloud 1. Hence, Cloud 1 cannot learn any useful information from the shuffled dataset and the shuffled matching table.

4.6.2 Analysis of the rest

First, we consider the case where the adversary tries to deduce secret information from the content of the published pairs. Fortunately, since the real id of leaves is replaced with a random number, such pairs do not reveal any sensitive information to attackers. Second, we consider the privacy protection of the noise management mechanism. For dummy records, we release them according to the real distribution of the incoming time of real data (see Section 4.3.3). This means that the probability of sending a dummy record at each time point is computed from the distribution of real records. Hence, when a pair of data arrives at the cloud, attackers cannot distinguish between a dummy and real pair. Therefore, the privacy of dummy records is also protected in `PINED-RQ++`.

On the other hand, removed records are kept at the collector. Note that attackers do not have prior knowledge about the real distribution of the incoming time of real data. Additionally, they do not know which leaf node receives negative noise. Hence, the attackers cannot know which real records are removed by the collector. In addition, such removed records are only released to the cloud after they are concealed into overflow arrays. These properties make the removing process of `PINED-RQ++` private.

4.6.3 Comparison with PINED-RQ

According to the above analyses, we can conclude that the security guarantee of `PINED-RQ++` is as strong as that of PINED-RQ when the two-cloud model is used. This means that `TEX-PINED-RQ++` does not incur any extra privacy leaks as compared to PINED-RQ, and hence, satisfies (ϵ, δ)-Probabilistic-SIM-CDP. In contrast, with the single-cloud model, `PINED-RQ++` only ensures the same level of privacy protection with

PINED-RQ against offline attackers. This is because it reveals real timestamps of data to online attackers.

4.7 Validation

We evaluate our solutions with regards to the two targeted contexts: low and high rate of incoming data. We demonstrate the efficiency and practicality of PINED-RQ++ by examining the effects of varying system configuration parameters. Section 4.7.1 shows the benchmark environment that is used for our experiments. We give results of our solutions in 4.7.5.

4.7.1 Benchmark environment

We implemented PINED-RQ++ in Java. All experiments of PINED-RQ++ are run on the Galactica platform [67]. The the non-parallel and the parallel version of PINED-RQ++ are configured as a set of several nodes running on Ubuntu 14.04.4 LTS. The configuration of nodes is detailed in Table 4.1. The TCP socket is used for exchanging data among the components, and the lfstat package [29] is utilized to monitor network traffic of the collector.

Table 4.1 – Experimental environment of PINED-RQ++

Component	vCPU (2.4 GHz)	Memory (GB)	Disk (GB)
Dispatcher	4	8	80
Computing node	2	2	20
Consumer	4	8	80
Cloud	16	64	160

4.7.2 Datasets

The experiments were performed with real datasets. We chose Gowalla [61], a social networking website where users share their locations by checking-in, and the US Postal Employees [30], USPS, dataset. The Gowalla dataset consists of 6,442,890 check-in records. For query attribute, we use the 32-bit integer representation of check-in times. The USPS

dataset compromises 624,414 employees. We use annual salary as a query attribute and filtered out employee records with an hourly payment rate. After filtering, the dataset consists of 394,763 records. The USPS dataset is highly skewed, whereas Gowalla is relatively uniform.

4.7.3 Settings

The branching factor (bf) is set to 16 and the total privacy budget ϵ_{total} to 1. The domain of A_q is normalized to [0, 100]. The size of overflow arrays is selected using the inverse of the CDF of the Laplace distribution with 99.99% confidence interval.

Query Set. In the experiments, we create various query sets of ranges corresponding to 1%, 5%, 10%, 25%, 50%, and 75% of the entire domain. For each set of query ranges, we sample 1000 queries uniformly over the domain. Unless stated, all other experiments are conducted using a uniform workload.

4.7.4 Metrics

We evaluate `PINED-RQ++` on four main metrics, namely network traffic, the time needed to publish a dataset, response time latency, and ingestion throughput. Additionally, since `PINED-RQ++` does not apply the constrained inference technique to the index, the quality of `PINED-RQ++` index may be different from that of PINED-RQ. We thus take the recall and precision metric into account. We use a time interval of 1 minute. We assume that each publication is about a disjoint set of individuals. The total privacy budget is 1 for each publication. We generate data in uniform fashion, hence dummy records are released uniformly.

4.7.5 Experimental results

We now present the experimental results of non-parallel `PINED-RQ++` in comparison with PINED-RQ. Especially, we point out that the former is able to avoid potential bottlenecks, that the latter suffers from, as incoming data is rapid.

a) Network traffic: Network traffic metric gives an idea of the stability of the overall system, which is crucial for analytical processing. In these experiments, we take

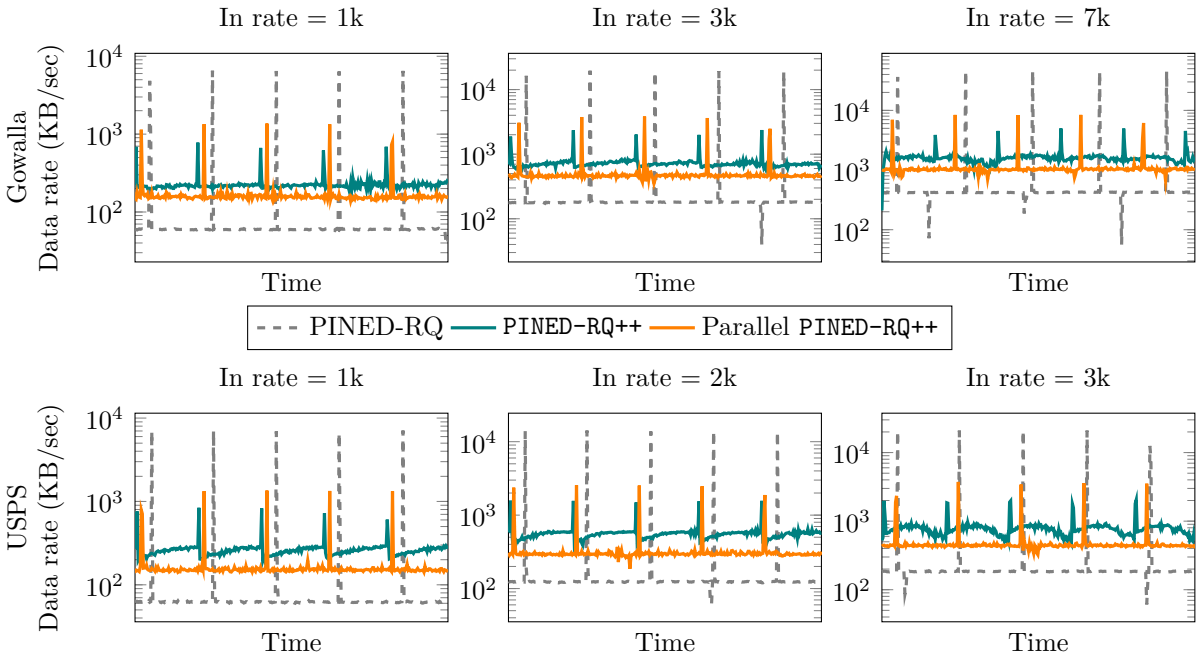


Figure 4.10 – Network traffic of the three prototypes

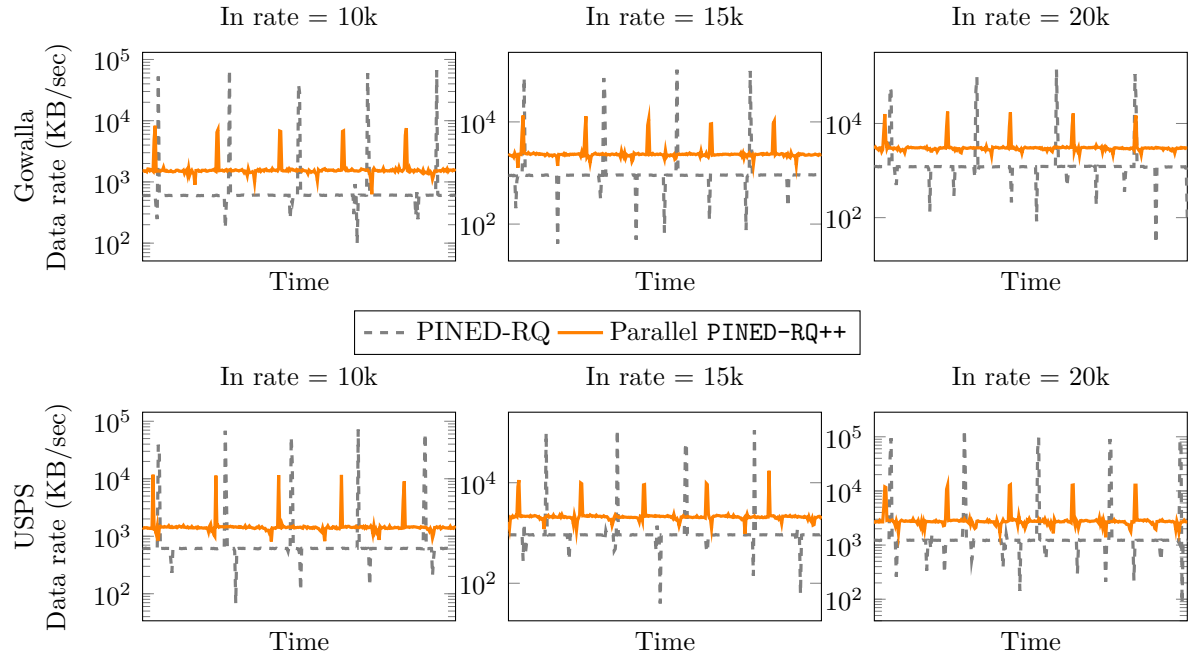


Figure 4.11 – Network traffic of PINED-RQ and parallel PINED-RQ++

successive 1-millisecond intervals and obtain data rate (in and out) at the collector over five minutes. Since the collector in non-parallel PINED-RQ++ performs heavy operations

(e.g., parsing, encrypting) on incoming data, its throughput is modest and around 7k (Gowalla) and 3k (USPS) records/s, as depicted in Figure 4.13. We thus consider two cases: **(1)** We compare network traffic among the three prototypes by choosing medium incoming data rates (*in rates*) varied between 1k and 7k records/s; **(2)** We then use higher in rates, ranging from 10k to 20k records/s, to measure network traffic of PINED-RQ and parallel `PINED-RQ++`.

(1) The results in Figure 4.10 show that the network traffic in (non-)parallel `PINED-RQ++` is significantly more stable than that in PINED-RQ. (Non-)parallel `PINED-RQ++` constantly experience a lower data rate compared to PINED-RQ. With an in rate of 7k records/s in Gowalla, the maximum data rate of (non-)parallel `PINED-RQ++` is (∼4,883 KB/s) ∼8,410 KB/s, and (∼9.2×) ∼5.5× lower than that of PINED-RQ (∼46,192 KB/s). Likewise, by using an in rate of 3k records/s in USPS, the maximum data rate of (non-)parallel `PINED-RQ++` is (∼2,049 KB/s) ∼3,734 KB/s, and (∼10.2×) ∼5.6× smaller than that of PINED-RQ (∼11,777 KB/s). `PINED-RQ++` and its parallel version achieve such gains since they immediately process and send incoming data to the cloud instead of publishing data in batches as in PINED-RQ.

It is worth noting that the maximum data rate of parallel `PINED-RQ++` is slightly higher than that of non-parallel `PINED-RQ++`. This is because parallel `PINED-RQ++` needs to transfer data between computing nodes and the dispatcher at publishing time, such as matching tables and clones of index template. In particular, the maximum data rate of parallel `PINED-RQ++` is ∼1.7× (in rate=7k records/s, Gowalla) and ∼1.8× (in rate=3k records/s, USPS) higher as compared to `PINED-RQ++`.

We also consider the *fluctuation range* of the three prototypes. The fluctuation range here represents the gap between the lowest point and the highest point of a data rate line over the measured period. A wide fluctuation range may highly contribute to the instability of the network. Figure 4.10 exhibits that the fluctuation range of all prototypes rises as the in rate increases. However, PINED-RQ exhibits a larger fluctuation range compared to (non-)parallel `PINED-RQ++`. For example, with an in rate of 7k records/s in Gowalla, the fluctuation range of (non-)parallel `PINED-RQ++` is at most (∼4,773) ∼7,741 KB/s while that of PINED-RQ is ∼46,135 KB/s.

(2) The results in Figure 4.11 give similar patterns as in **(1)**. With the three different in rates (10k, 15k, and 20k records/s), parallel `PINED-RQ++` still has better network

stability than PINED-RQ. The maximum data rate of parallel `PINED-RQ++` is at least ∼6× (in rate=15k, USPS) and at most ∼10.5× (in rate=20k, USPS) lower than that of PINED-RQ. Similarly, as we use an in rate of 20k records/s, Gowalla dataset gives the maximum fluctuation range, with ∼15,834 KB/s of parallel `PINED-RQ++` against 138,370 KB/s of PINED-RQ. In other words, the maximum fluctuation range of parallel `PINED-RQ++` is ∼8.7× smaller than that of PINED-RQ.

b) Publishing time: We compare the time required to publish a dataset to the

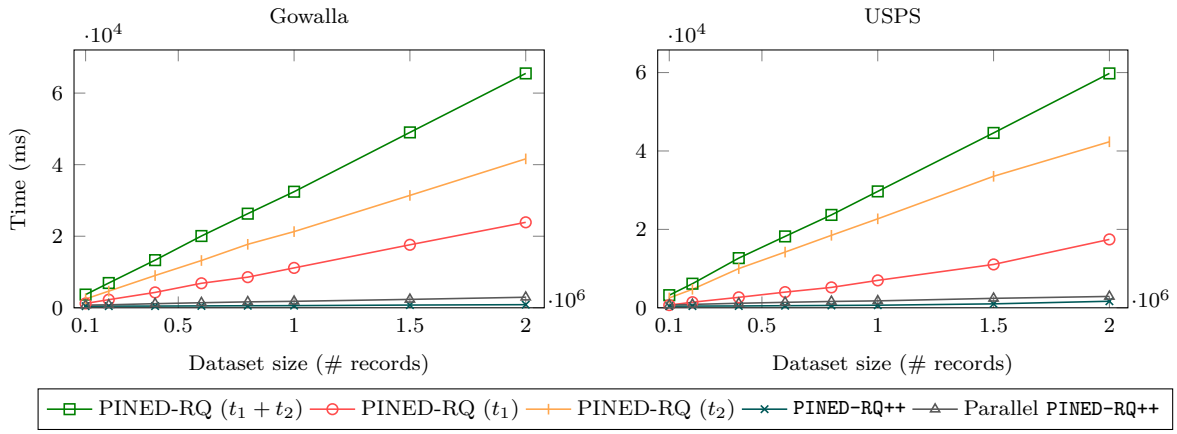


Figure 4.12 – Publishing time

cloud among the three prototypes: PINED-RQ, `PINED-RQ++`, and parallel `PINED-RQ++`. This metric is essential in case of applications having high rate of incoming data since a long delay may lead to bottlenecks in the system. We measure such metric by varying the size of publications. To obtain datasets with different sizes, we simply adjust the time interval parameter. Note that PINED-RQ consists of two main tasks that mainly contribute to the publishing time at the collector: 1) building an index; 2) publishing a dataset, that includes the encryption of the corresponding dataset. Thus, we also evaluate the processing time of these tasks, represented as t_1 and t_2, respectively.

The results in Figure 4.12 show that (non-)parallel `PINED-RQ++` mostly needs a very short time to publish a dataset while the publishing time in PINED-RQ is proportional to the size of datasets. Specifically, as we increase dataset size, the time remarkably goes up in PINED-RQ while the publishing time of (non-)parallel PINED-RQ remains almost unchanged. This is because (non-)parallel PINED-RQ only publishes two small

data structures, e.g., index template and matching table, at publishing time. The size of index template is independent of the size of dataset while that of matching table slightly rises due to the increase of the number of random ids. These properties make the publishing time of (non-)parallel PINED-RQ very short, and hence, they are able to avoid potential bottlenecks at the collector. As shown in Figure 4.12, for a dataset of 2M records, the publishing time of (non-)parallel `PINED-RQ++` is about (75×) 22× and (35×) 20× shorter than that of PINED-RQ for Gowalla and USPS, respectively. Moreover, PINED-RQ witnesses a long delay, ∼65s (Gowalla) and ∼60s (USPS), that is even longer than or equal the set time interval (e.g., 60s). This means that as the system needs to publish a dataset of 2M records within 60 seconds, bottlenecks will occur at the collector in PINED-RQ. In contrast, (non-)parallel PINED-RQ needs at most ∼2.95s to publish a dataset of such size.

It is also worth noting that although the building time (t_1) and the publishing time (t_2) both increase according to the dataset size, PINED-RQ spends too much time for publishing datasets instead of building indexes. Figure 4.12 gives that the gap between t_1 and t_2 incrementally rises as the size of datasets goes up. In particular, with a dataset of 2M records, PINED-RQ needs at least 41.62s for publishing and at least 17.44s for building the index. Such difference comes from the fact that PINED-RQ must heavily encrypt datasets at publishing time, thereby making the time t_2 constantly longer than t_1. In contrast, (non-)parallel `PINED-RQ++` avoids that heavy computation at publishing time thanks to the index template-based approach.

c) Ingestion throughput: We now evaluate the performance of parallel `PINED-RQ++` by comparing its ingestion throughput with that of non-parallel `PINED-RQ++`. The ingestion throughput is defined as the number of incoming records that a system is able to process per second. To capture the maximum throughput of the two prototypes, the incoming data rate is chosen to be very high, e.g., 200k records/s. Different number of computing nodes are considered in these experiments. For each presented value, we take 10 successive publications and obtain the average.

As we expected, parallel `PINED-RQ++` always has significantly higher throughput compared to non-parallel `PINED-RQ++`. The results in Figure 4.13a show that when the number of computing nodes increases, the ingestion throughput of parallel `PINED-RQ++` is considerably improved and reaches the peaks of ∼46k records/s (Gowalla) and ∼34.8k records/s (USPS). Meanwhile, non-parallel `PINED-RQ++` can only ingest up to ∼7k

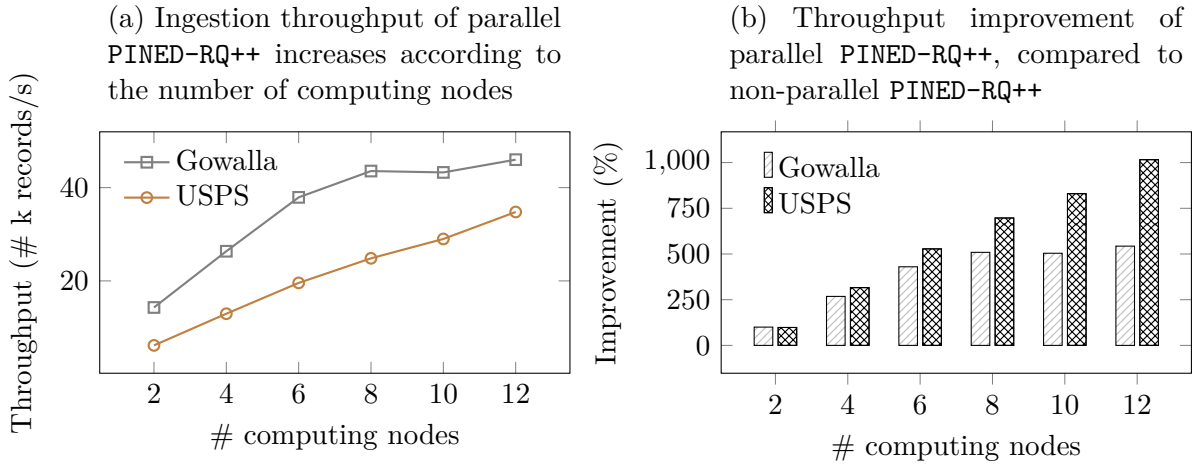


Figure 4.13 – Ingestion throughput improvement of parallel PINED-RQ++. Note that the throughput of non-parallel PINED-RQ++ is ∼7k records/s (Gowalla) and ∼3k records/s (USPS)

records/s (Gowalla) and ∼3k records/s (USPS). As shown in Figure 4.13b, with the setting of 12-computing node cluster, the enhancement of parallel PINED-RQ++ is approximately 5.4× and 10× in Gowalla and USPS, respectively. Note that the gap between the two datasets comes from the fact that the size of the USPS records is larger than the other.

d) Query latency: We turn our attention to query latency over intermediate data (un-indexed data). PINED-RQ keeps intermediate data on disks at the collector. In contrast, PINED-RQ++ holds intermediate data at the cloud as pairs of <*random number, ciphertext*>. For this scenario, the data generator produces different incoming data rates between 1k and 7k records/s. The consumer sends one query per second.

In Figure 4.14, the results indicate that as in rate increases, the latency goes up in all prototypes. As expected, PINED-RQ has lower latency compared to PINED-RQ++ for all range sizes. The reason is that PINED-RQ++ always downloads all intermediate data and locally processes them at the consumer. Meanwhile, in PINED-RQ, the range query is processed at the collector and only the relevant records are returned. As shown in Figure 4.14, the largest gap is witnessed as an in rate of 7k records/s and Gowalla dataset are used. Specifically, PINED-RQ++ takes about 6s while PINED-RQ spends 1.8s for a 25%-range query. However, it is important to note that the overhead of PINED-RQ++ only takes place over immediate data whose size is tiny as compared to historical data. Furthermore, while PINED-RQ suffers from query processing on intermediate data at

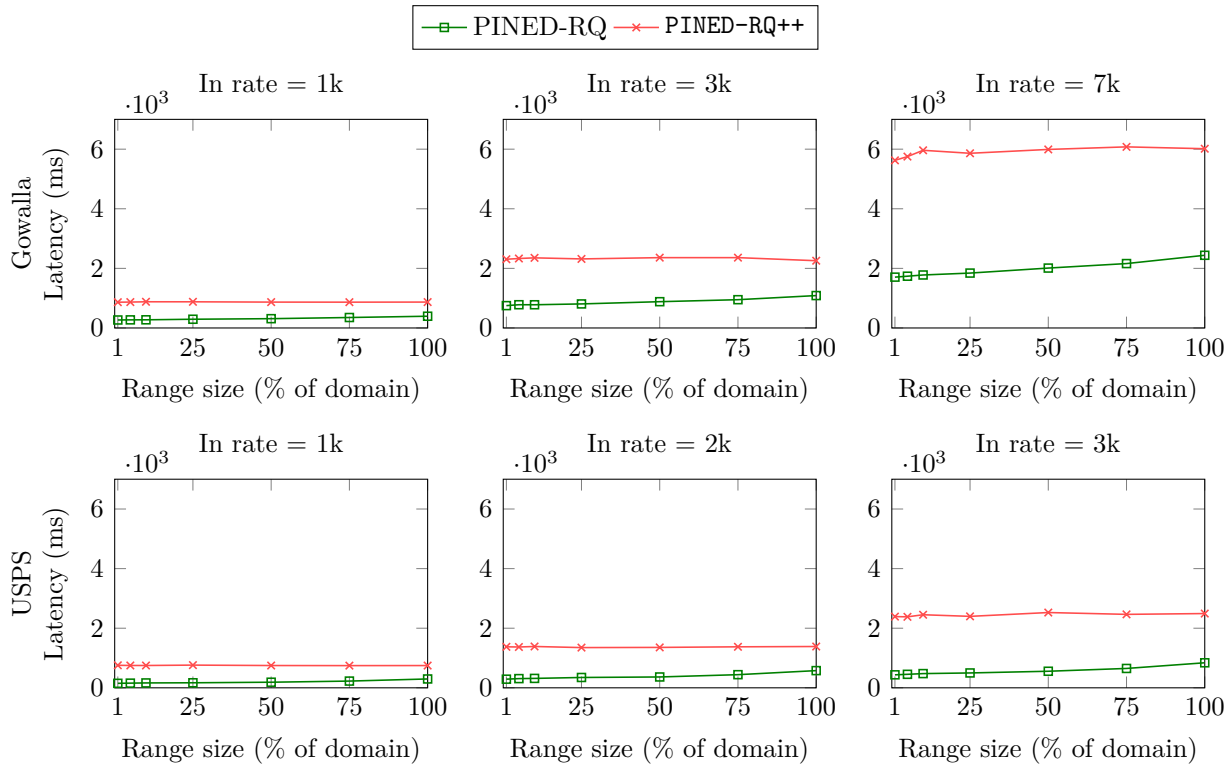


Figure 4.14 – Response time latency of 1000 queries

the collector, PINED-RQ++ almost eliminates such computation at this component. As a result, PINED-RQ++ would avoid overload at the collector in case of high workload.

e) Recall and precision: As discussed in Section 4.3.6, PINED-RQ++ builds the index during a time interval, and does not apply the constrained inference approach [52] to the index. This may give rise to differences in recall and precision rate between PINED-RQ and PINED-RQ++. It is necessary to analyze the utility of the index of the two prototypes. Apparently, the speed of incoming data does not impact the quality of the PINED-RQ++ index, but the size of publications. Thus, we just consider the variation of dataset size instead of incoming data rate. We use variants of Gowalla dataset that represent different periods, 5 days (95,276 records), 10 days (304,055 records), and 25 days (837,402 records). The measures are taken after a matching table is published to cloud. The results in Figure 4.15 indicate that the recall of the two prototypes holds 100% for all range sizes. The reason is that the indexes have no negative inner nodes that often cause missed records in the returned results. Considering the precision, the

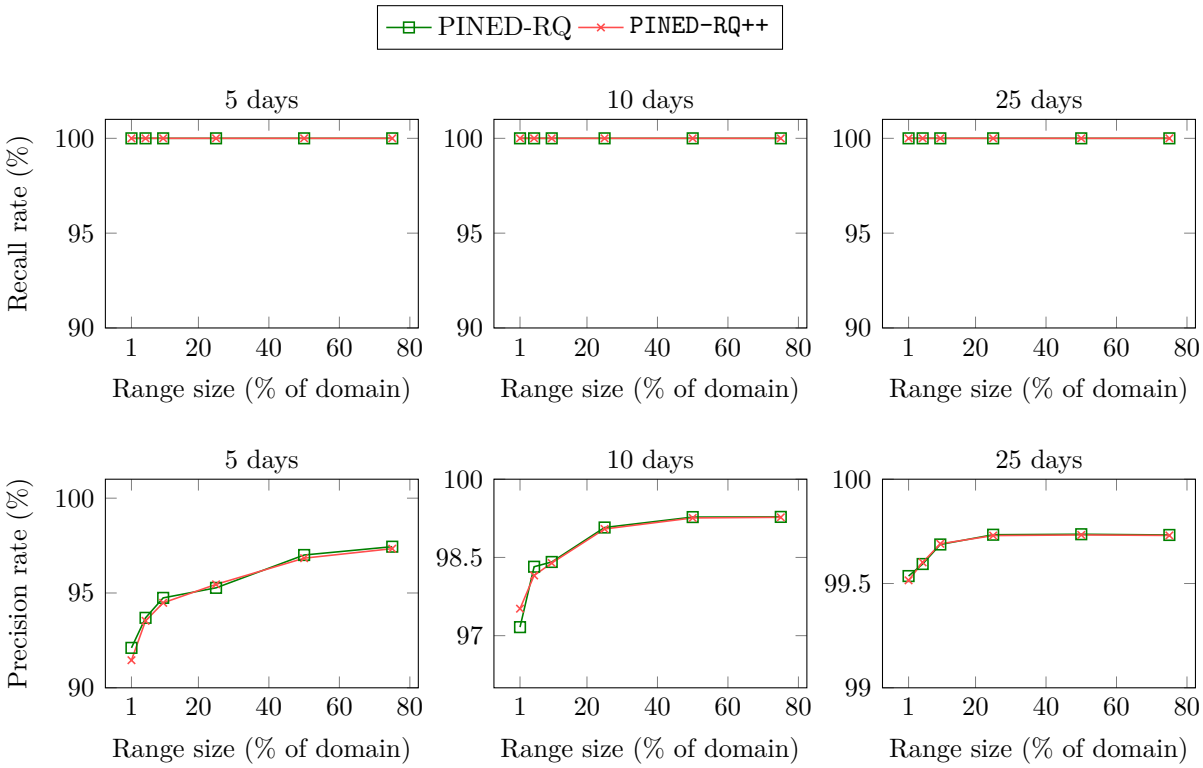


Figure 4.15 – Comparison between PINED-RQ and PINED-RQ++ in terms of recall and precision

two prototypes are competitive and preserve a high percentage, even if the smallest range size (1%) is used, particularly, for the 5-day dataset, the precision rate is at least 92.10% (PINED-RQ) and 91.45% (PINED-RQ++). However, as a 10-day dataset is used, the minimum precision rate of PINED-RQ++ is slightly higher than that of PINED-RQ, with 97.51% in PINED-RQ++ against 97.16% in PINED-RQ. For the 25-day dataset, the precision rate of both is almost the same and constantly higher than 99.51%.

4.8 Conclusion

High rate of incoming data have raised major challenges to prior schemes on secure range queries, specially bottlenecks in the system. Therefore, we developed PINED-RQ++ to address these challenges. To do that, we introduce a new approach to build secure indexes based on the notion of index template. By doing so, PINED-RQ++ allows continuously ingesting arrivals as soon as possible, and hence avoiding bottlenecks in case of high

rate of incoming data. Furthermore, we also propose a parallel PINED-RQ++ that significantly enhances the ingestion throughput, making our solutions practical to real-world applications. In terms of security, we analyze and show that the non-parallel and the parallel version of PINED-RQ++ do not reveal anything about publications as compared to PINED-RQ when the two non-colluding clouds is used. Otherwise, if a single cloud model is used, PINED-RQ++ guarantees that level of security against offline attackers. We finally implemented the non-parallel and the parallel version PINED-RQ++ and extensively carried out experiments on real datasets. The experimental results show that our solution provides better performance than PINED-RQ while privacy is also protected.

Although PINED-RQ++ can prevent bottlenecks from happening in the system in general, both (non-)parallel PINED-RQ++ cannot cope with real-life needs in terms of the ingestion throughput. This make them unable to many applications where the system has to accept huge number of arrivals in very short period. To address this limitation of PINED-RQ++, in the next chapter, we propose a new approach that allows scaling up the ingestion capacity of the system, e.g., over one hundred thousand of records per second.

Chapter 5

FRESQUE

Abstract: The previous chapter presented the solutions for preventing potential bottlenecks from the context where data arrives at a high rate. Nevertheless, similar to prior schemes, their scalability still remains limited. Their ingestion throughput is not scalable enough for real-life needs. To address this limitation, in this chapter we introduce a novel framework for secure range query processing, `FRESQUE`. Indeed, we aim at enhancing `PINED-RQ++` so that it is able to scale the intake ability up as much as possible. To this purpose, we introduce a new architecture relying on a set of shared-nothing machines and make it fully distributed. With such an approach, the ingestion throughput can be scaled up by just adding more nodes into the system. In addition, a data representation and an asynchronous publishing method are presented and integrated into this architecture to avoid throughput degradation or potential congestion. By carefully coordinating all of these elements together, `FRESQUE` can support an intensive consumption throughput, e.g., 165 records/second. As compared to `PINED-RQ++` and parallel `PINED-RQ++`, the ingestion throughput is improved by 43× and 5.6×, respectively. Moreover, we also adapt `FRESQUE` to a stronger type of attackers by presenting a new noise management into the new architecture. Interestingly, this method also increases the practicality of the framework.

5.1 Introduction

Although parallel PINED-RQ++ can help improve ingestion in the system, it is far from satisfactory. Intensive ingestion throughput is demanded by a wide range of modern applications. For instance, seasonal influenza is a primary cause of morbidity and mortality worldwide, 3 to 5 million of serious illnesses [114], and for 290,000 to 650,000 deaths annually. Reducing the impact of seasonal epidemics (e.g., influenza) is demanding for public health officials. To address it, participatory surveillance systems, to name few, have been deployed in recent years such as Flu Near You [39] in North America, Influenzanet [49] in Europe, and FluTracking in Australia and New Zealand. FluTracking [27, 40] is a surveillance system that weekly collects symptoms from the participants to typically track influenza nation-wide. Even recently, some extra coronavirus-related questions have been added to track the spread of COVID-19 [73]. In 2020, FluTracking has around 140 thousand reports every week [40]. Similarly, to control the spread of COVID-19, many governments require travelers and all residences submitting e-medical declaration forms such as [108, 3]. In these systems, a large amount of data is generated every second. However, PINED-RQ++ does not exhibit a high throughput. It is only able to consume up to ∼46K records/s that is much lower than real-life needs, that might need to support hundreds of thousand insertion per second.

Therefore, this chapter aims at developing a scalable ingestion framework dedicated to secure range query processing. Our solution is developed based on the PINED-RQ family [94, 104]. This choice is motivated by the fact that the PINED-RQ family can achieve both fast range query computation and provable security while the clear secure index requires very small space. More specifically, we re-design the architecture of PINED-RQ++ [104] in order to make it fully distributed. We attempt to parallelize all heavy tasks (e.g., parsing and encrypting data) on a set of *shared-nothing* machines [1]. By relying on such architecture, the ingestion throughput can be scaled up/down by just adding/removing nodes into the system. Additionally, a data representation and a asynchronous publishing method are introduced and integrated into this architecture to avoid throughput degradation. By carefully coordinating all of them together, our framework is scalable and supports intensive ingestion throughput. Experimental results show that compared to the non-parallel and parallel version of PINED-RQ++ [104], the ingestion throughput is improved by ∼43× and ∼5.6× in NASA [78] dataset, respectively. Furthermore, we

1. We take the shared-nothing architecture into account since it is highly scalable. In order words, the system can scale up to several nodes easily.

also present a new noise management mechanism to adapt PINED-RQ++ to a stronger type of adversaries who may have good knowledge about the timestamp 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 4.3.3). In this chapter, we develop **FRESQUE**, an 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 [94, 104], and point out obstacles that prevent the existing architecture from achieving a high ingestion throughput. Also, we present an approach to adapt our framework to stronger adversaries.
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 [87, 86, 12].
4. We give a formal proof of the security of FRESQUE.

The chapter is structured as follows. In Section 5.2, we briefly present problem statement. We then describe our framework in Section 5.3. Section 5.4 gives security analyses while Section 5.5 presents our experimental results. We discuss a potential application of our solution and potential extension in Section 5.6. We conclude and give future work in Section 5.7.

5.2 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 $D(A_1, \ldots, A_d)$, where A_i is an attribute. Queries are non-aggregate one-dimensional range queries. A query $\mathcal{Q}$ is evaluated over the attribute A_q of D, which contains numerical values. Periodically, the collector will pre-process the dataset, e.g., building a secure index over the dataset and encrypting it. The processed

dataset is then sent to the cloud for serving range queries.

In this chapter, we mainly focus 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 period.

5.2.1 Threat model

Similar to PINED-RQ++, this architecture targets attackers at the cloud. We thus consider the *honest-but-curious* model [46]. 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. In this chapter, we also consider two types of attackers, *online* and *offline* attackers. Nevertheless, the one that differentiates from PINED-RQ++ is that this chapter considers stronger online attackers who may have good knowledge about the data distribution of the incoming time of real data, named *online-ext*.

5.2.2 Security guarantees

Recall that the security guarantee of PINED-RQ++ is (ϵ, δ)-Probabilistic-SIM-CDP when the two-cloud model is used. However, with the single-cloud model, PINED-RQ++ only ensures that level of privacy protection against offline attackers because it reveals data timestamps to online attackers.

In this chapter, with the single-cloud model, we seek to design FRESQUE to meet the (ϵ, δ)-Probabilistic-SIM-CDP [94].

5.2.3 Limitations of PINED-RQ++

We now identify the obstacles that hinder PINED-RQ++ from achieving a scalable solution, particularly parallel PINED-RQ++.

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 so, the count variables of the index template are updated and referenced by the updater and the checker, respectively, during a time interval. The index template is thus considered as a *shared* data structure, that was not run simultaneously in the 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 4.6).

Unfortunately, the parsing task is heavy and 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 show that the parsing task reduces the throughput of the collector by over 50% when we use NASA dataset [78].

Heavily updating index template. Since PINED-RQ++ uses the whole index template for updating and checking incoming data, leading to 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, incurring 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 to the index template have to be processed sequentially.

Synchronous publishing. 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 example, at the end of each 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. More specific, bottlenecks are more likely to occur as the collector has to publish large overflow arrays within a short time interval.

5.3 FRESQUE

Before describing FRESQUE, we now present its principal modules that enable to tackle the presented problems of PINED-RQ++ [104] being identified in Section 5.2.3. Then, we present a complete architecture of FRESQUE and its instantiation.

5.3.1 Principle designs

a) A fully distributed 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 challenge is that the checker, that resides between the parser and the encrypter in the workflow, cannot be parallelized since it relies on not only the parser, but also 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. 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 5.2. This approach allows to scale the intake ability of the collector without creating unnecessary transmission overheads. However, the question is how the checker processes a record after it is encrypted. To address it, we 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 B+-Tree 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.

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

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 5.2. 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 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 of 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.

5.3.2 Upgrade of `PINED-RQ++`

Injected noise, e.g. dummy/removed records of a publication, may be disclosed as the incoming order of data at the cloud is the same with that at the collector. We note that in this chapter we consider the threat model where attackers have good knowledge about the distribution of incoming time of real data, they might know for sure there is a real record at a certain time point. However, some records may be buffered (removed) by the

checker at the collector since it lies in the interval of a negative leaf. Consequently, the absence of such removed records at the cloud would be recognized by online-ext, and hence the secure index no longer guarantees differentially private. On the other hand, the insertion of dummy data also confronts the same privacy issue. That is, online-ext can easily determine whether an arrival is a dummy or real record based on prior knowledge about real data distribution. For that reason, `PINED-RQ++` can not be protected against online-ext.

In addition, the noise management method of `PINED-RQ++` requires the collector to know in advance the distribution of the incoming time of real data. Indeed, dummy records of a publication are released by the collector according to such distribution in order to prevent privacy disclosure. Nonetheless, the distribution of the incoming time of real data is often difficult to be determined in real-life applications. For instance, in FluTracking [40], an admin does not know exactly when a participant submits data to the system. Consequently, such requirement will limit the application of the solution. We now seek to design a new noise management method that must avoid privacy leaks and does not depend upon any pre-defined distribution of the incoming time of real data.

In order to achieve these goals, one may randomly release dummy records over a time interval. This approach is straightforward and allows to eliminate the dependence upon a pre-defined distribution of timestamps. Nonetheless, the problem of this method is the privacy disclosure of injected noise to the cloud. As stated in Section 5.2.1, the adversary may have good knowledge about the distribution of incoming time of real data. As dummy data are randomly released, they might arrive at cloud at an unlikely time point at which there is no real data. Consequently, the privacy of dummy data is disclosed to the adversary. Likewise, removed records will be recognized by the adversary.

To address this, we introduce a new component, randomer, to our architecture (see Figure 5.2). 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 being randomly sent to the buffer of the randomer during a time interval. For example, suppose a publication has 100 dummy records, then 100 time points are randomly chosen over a time interval, each dummy record is released at a such 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, although a new record arrives the cloud at an improbable time point, the adversary cannot conclude whether it is dummy or not. Similarly, when the adversary does not see an expected record at a time point at cloud, she/he cannot be sure it is removed or not due to the uncertainty caused by the randomer. The leakage caused by injected noise is thus addressed by the randomer. Moreover, since the collector randomly releases dummy data, `FRESQUE` does not require knowing in advance a data distribution, hence it disposes of the issue of limited practicality.

Challenges of randomer. One of the challenges of using randomer is how to

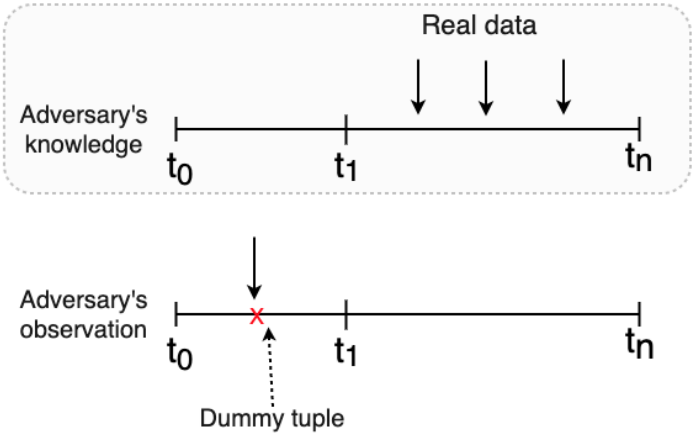


Figure 5.1 – Privacy leak of randomer

choose a right size for its buffer. Intuitively, a large buffer gives high security. However, if the chosen size is huge, the system may confront bottlenecks at collector, particularly at the checking node. Otherwise, a tiny buffer may result in the privacy leak of dummy records, especially when the capacity of the randomer buffer is smaller than the total number of the dummy records of a publication. As an example depicted in Figure 5.1, we first assume that no real data is presence during the period between t_0 and t_1 and the size of the buffer is smaller than the total number of dummy records. Since all dummy records are randomly released over a time interval, there is a case where all dummy records will be released during the period $[t_0, t_1]$. The buffer is subsequently full at this time, the trigger function is 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 smaller than the total number of dummy records. Otherwise, if the randomer buffer is larger than that number, no records will appear at the cloud in that period. Therefore, the buffer size must be chosen to be larger than the total number of the dummy records of a 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 a 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.

We note that since dummy records are generated due to the Laplace noise, the number of dummy records varies for each publication. It is thus difficult to choose a right capacity for the randomer buffer while meeting both (*) and (**). Fortunately, since the noise in `FRESQUE` is sampled from the Laplace distribution, we can 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. Interestingly, it is earlier used to select the size of overflow arrays to protect the privacy of removed records in PINED-RQ [94].

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 $\alpha \geq 2$.

Note that although the randomer size is exposed to the adversary, she cannot infer the chance of picking a dummy/real record when an encrypted record arrives at the cloud at any time point t_i. Indeed, to achieve such chance, the adversary needs to rely on real data distribution, however, such distribution has been perturbed by the randomer before t_i. Hence, when the adversary receives an encrypted record, it cannot distinguish between real and dummy record.

Another question is that where the randomer should be placed at the collector. A

wrong decision would disable the randomer. For example, if we put the randomer after the updater, the privacy of removed records is revealed. The reason is that incoming records are checked and may be removed before being perturbed by the randomer. Otherwise, as it resides at the dispatcher, the collector would incur long publishing time since it has to heavily processes (e.g., parsing and encrypting) the randomer's buffer whose size is adjustable. Therefore, the right choice should be the position between the checker and computing nodes, as shown in Figure 5.2. With this organization, randomer prevents privacy leak of removed records while resulting in a negligible delay at the publishing time.

5.3.3 Architecture of FRESQUE

Following the principal design in Section 5.3.1, we now detail the novel ingestion framework to support efficient range query processing over encrypted data. Especially, we describe how we orchestrate different components in the new architecture.

a) Ingestion life cycle

The collector of FRESQUE runs on a small cluster of commodity machines as shown in Figure 5.2. At the collector, one (and only one) node runs Dispatcher (D) and all worker nodes in the cluster run a Computing Node (CN) while the randomer, the checker and the updater runs on the same Checking node (C).

The dispatcher plays the role of a coordinator. At the beginning of each time interval, only the dispatcher can start a new publication by initiating a new Index Template (IT), Publication Number (PN), and scheduling the releasing time of dummy data according to the method explained in Section 5.3.2. Finally, the dispatcher sends IT and PN to the checking node. The checking node takes IT from the dispatcher and generates the corresponding AL and ALN.

During a time interval, when new records arrive 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 forwarded to the cloud or the merger. At the end of each time interval, the dispatcher notifies all components at the collector by a message. Note that the dispatcher, the computing node, the merger,

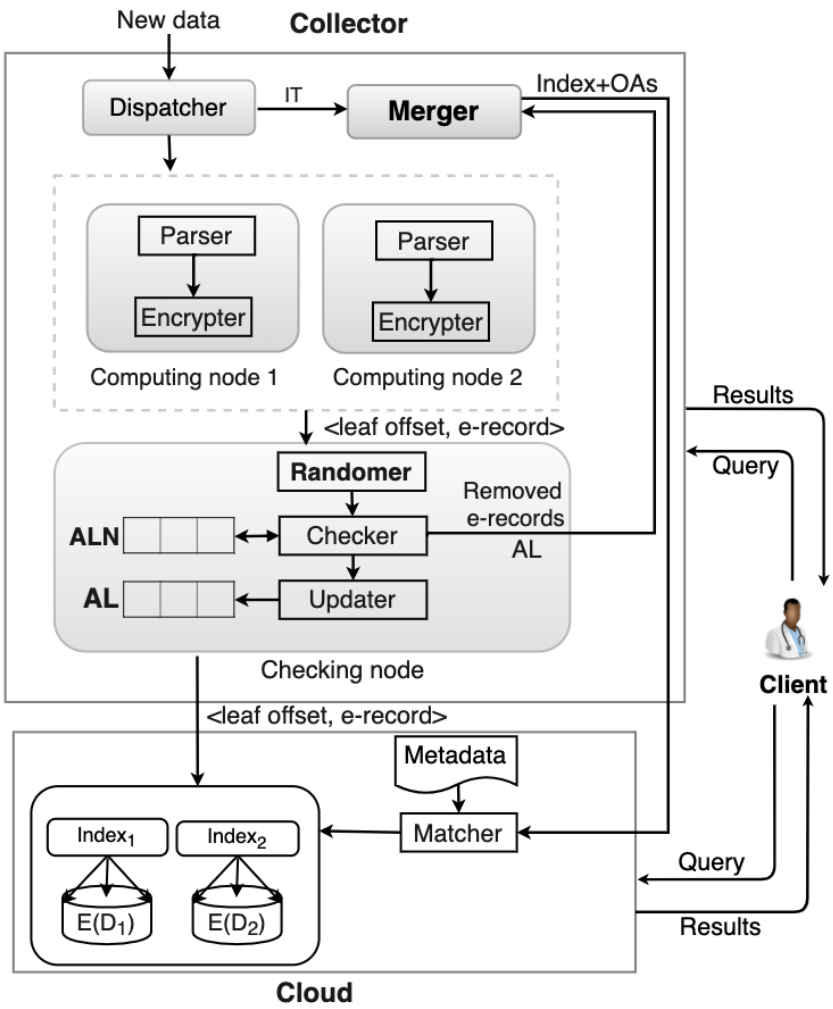


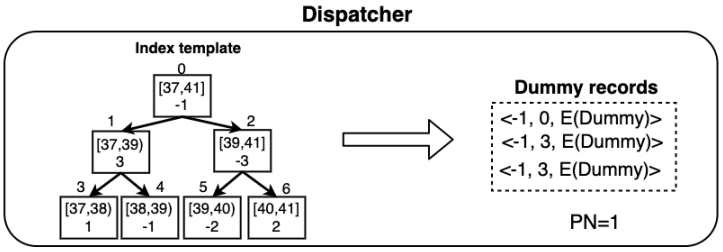
Figure 5.2 – Architecture of FRESQUE

and the checking node can coexist on the same node.

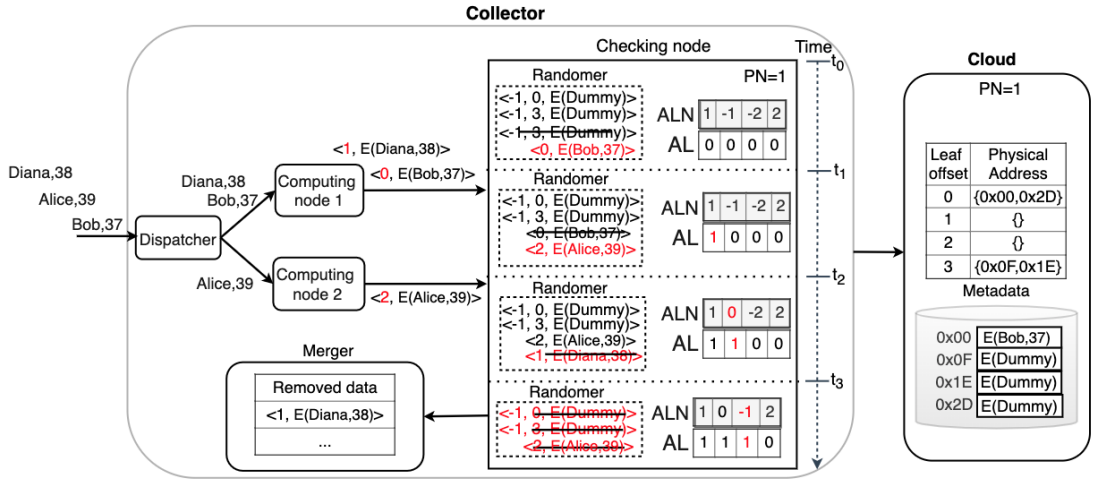
b) Instantiation of FRESQUE

In order to demonstrate how data is processed at the collector and transported to the cloud, Figure 5.2 shows the composition of FRESQUE running on five nodes at the collector and Figure 5.3 gives a running example.

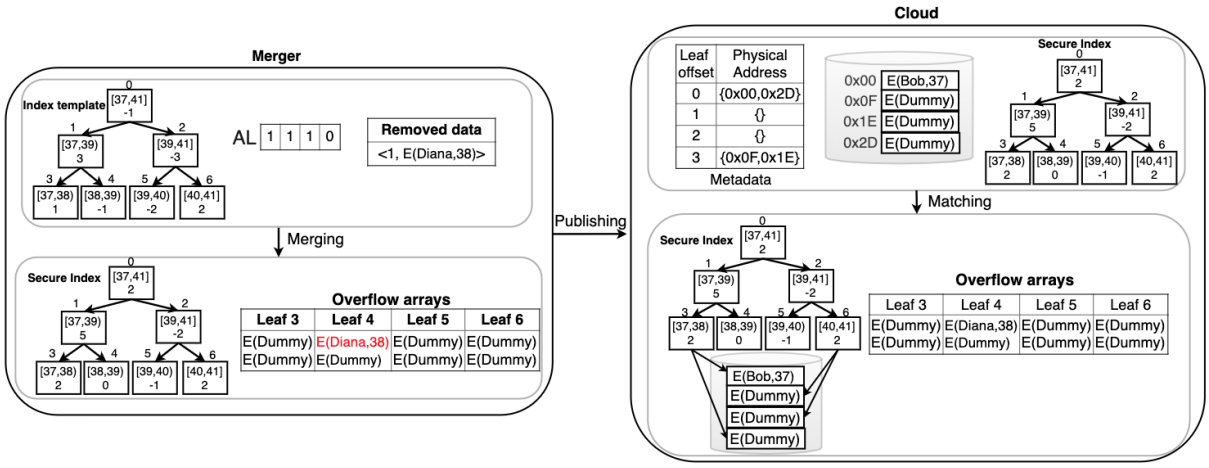
Dispatcher (D): At the beginning of a time interval, the dispatcher initiates an Index Template (IT), dummy records, and a Publication Number (PN), as illustrated in Figure 5.3a. The dispatcher then sends the IT and the PN to the checking node.



(a) Initiation



(b) During a time interval



(c) At publishing time

Figure 5.3 – A simple example demonstrating how FRESQUE process incoming data by using two computing nodes. Assume that the size of randomer buffer is 4 pairs of e-record. For the sake of simplicity, we also assume that all dummy records are released before the arrivals of real data.

At the same time, it also schedules the releasing of all dummy records as presented in Section 5.3.2. During a 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 5.3b, there are three records, (Bob,37), (Alice,39), and (Diana,38), arriving in order at the dispatcher. These records are then distributed to the two computing nodes. At the end of each 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 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 5.3b, after passing the two computing nodes, three records are now parsed, encrypted, and associated with the corresponding leaf offsets, 0, 2, 1, respectively. We assume that these pairs of records come to the checking node in the same order as they arrive at the dispatcher. 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 the 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 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 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 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 5.3b). The checking node then forwards the IT to the merger while the PN is sent to the cloud. During a 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 5.3b, 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^{th} 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 5.3c). 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 order 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 5.3b, 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 time interval, the merger receives IT and PN from the checking node, then keeps them in memory. During a 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., combing 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 5.3c. We note that merger runs independently of the ingestion components, a long delay in this component does not result in any impact on the ingestion throughput of the system, but the availability of secure indexes at the cloud. We delay the discussion of the latter in Section 5.6.

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 5.3b. 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 5.3c).

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 3.5. 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, at the randomer, and the merger are returned to the client. Finally, the client use a shared private key to decrypt and filter dummy data from the results.

5.4 Security analysis

We develop FRESQUE that builds a PINED-RQ index [94] during a time interval. With such an approach, at the end of each 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 satisfies (ϵ, δ)-Probabilistic-SIM-CDP.

The main difference between FRESQUE and PINED-RQ [94] in the index creating function is that FRESQUE immediately publishes encrypted records, that include dummy records and removed records, during a time interval. However, since attackers know the distribution of the incoming time of real data, they are able to infer which record is dummy or real based on the timestamps of the published data. Consequently, FRESQUE (without the randomer) does not meet (ϵ, δ)-Probabilistic-SIM-CDP [94]. Fortunately, thanks to the randomer, such potential privacy disclosure is avoided. Theorem 2 shows the security of the FRESQUE.

Theorem 2 (Security of FRESQUE): The index creating function of FRESQUE satisfies the (ϵ, δ)-Probabilistic-SIM-CDP [94].

Proof. **Considering dummy records**. 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 does not infer any useful information from such case.

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 the full randomer buffer. This means that this randomer buffer contains both real and dummy records. Hence, a record is picked and released to the cloud at this time point is indistinguishable under the adversary's view.

Third, we consider the case where the checking node sends the randomer buffer to the cloud at the end of each time interval. Indeed, dummy records are mixed with real ones at the randomer during a time interval. Additionally, the ratio between real and dummy data at any time point is also private from the adversary (see Section 5.3.2), hence the adversary cannot infer any useful information from this case.

Fourth, we consider the case where the randomer contains all dummy records and no real ones. Note that such situation only occurs as all dummy records are released before real data arrive at the collector. If the buffer of the randomer is smaller than or equal to the total number of dummy records, a dummy record is certainly picked and released to the cloud. As a consequence, the adversary can conclude it is dummy, and `FRESQUE` no longer meets (ϵ, δ)-Probabilistic-SIM-CDP [94]. However, `FRESQUE` requires that the buffer of the randomer is chosen to be larger than the total number of dummy records of a publication (see Section 5.3.2). Hence, this case does not happen.

However, since a dummy record may cause an arrival at the cloud, the randomer may reveal the total number of dummy records. There may be some dummy records do not cause an arrival at the cloud since they are released at the beginning of a time interval, especially when the randomer is not full. Similarly, an dummy record may result in a picking and releasing of a record that is then removed by the checker, hence an arrival of the dummy record does not always reveal to the adversary. This means the total number of dummy records, that the adversary recognised, may be different from the actual total number of dummy records. Clearly, the larger randomer buffer, the less chance of leaking such information. In other words, the possibility of such limited leakage can be controlled by adjusting the coefficient α. Especially, when the randomer buffer size is equal to the publication size, such leakage is eliminated.

Considering removed records. Since dummy records will not be deleted by the checking node, we only consider the case where a real record that belongs to a negative leaf. Without the randomer, this record is removed and sent to the merger by the checker and the adversary can conclude that there is a record being removed from dataset due to its absence at the cloud at an expected time point. This privacy leakage results from the knowledge of the adversary about the incoming order of real data. However, the randomer enables to *randomize* such order before the removing process is performed. This means that a record arrives at the randomer at time point t_i, and it can be released to the cloud at any time point t_j ($t_i \leq t_j \leq t_n$), where t_n is the publishing time point.

Thus, after passing the randomer, the order of a record is randomized. This property makes removed records secret from the adversary. Although the adversary does not know which record is removed from a publication, one may think that FRESQUE will leak the total number of removed records since the adversary can observe whenever an expected record does not arrive at the cloud. However, removed records may be delayed until the time point t_n. This confirms that the total number of missed records during a time interval may not be equal to the total number of removed records. Similar to the case of inserting dummy data, the possibility of such limited leakage can be controlled by adjusting the coefficient α. When the randomer buffer size is equal to the publication size, such leakage is eliminated. □

Comparison with PINED-RQ [94]. The highest security of FRESQUE is achieved as 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 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 case is the same with the publishing process of PINED-RQ [94]. Thus, in that case, the FRESQUE has the same level of privacy protection with the PINED-RQ and also satisfies (ϵ, δ)-Probabilistic-SIM-CDP [94].

Comparison with PINED-RQ++ [104]. FRESQUE has better security since it prevents privacy leaks from stronger adversaries, particularly online-ext.

5.5 Validation

In this section, we evaluate the FRESQUE against the (non-)parallel PINED-RQ++ [104]. 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.

5.5.1 Benchmark environment

We implemented FRESQUE in Java 1.8.0. Data was encrypted by the Java package (javax.crypto). We ran our experiments on the Galactica platform [67] and organized FRESQUE as a cluster of 17 nodes running on Ubuntu 14.04.4 LTS. Each node was used to run one component of the FRESQUE. The configurations of nodes are detailed in Table 5.1. The TCP socket was used for exchanging data among the components of the FRESQUE.

Table 5.1 – Experimental environment of FRESQUE

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 evaluate our solution on two real datasets NASA log [78] (1, 569, 898 records, five attributes), Gowalla [61] (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 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 all experiments were run over ten minutes. Then, we present the averaged results of ten publications in Section 5.5.2.

5.5.2 Results

a) Ingestion throughput

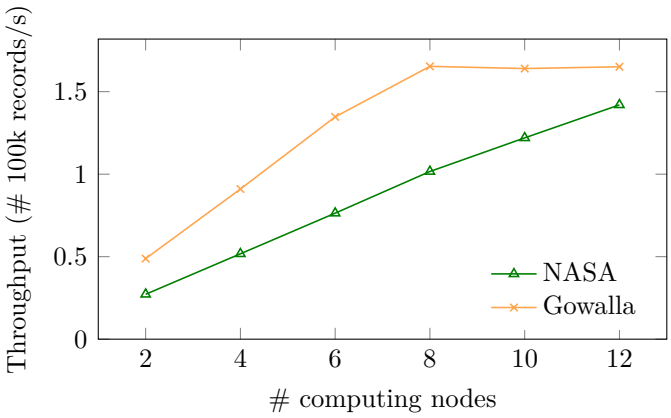


Figure 5.4 – Ingestion throughput of FRESQUE

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 5.4 show that the throughput of FRESQUE significantly increases as the number of computing nodes goes up. Especially, the highest throughput is reached to ∼142k records/second (NASA) and ∼165k records/second (Gowalla) as we use the setting of 12 and 8 computing nodes, respectively. As compared to ArxRange [86], one of the state-of-the-art solutions, FRESQUE ensures the ingestion throughput is at least two orders of magnitude higher.

(1) Comparison with non-parallel PINED-RQ++. With the given settings, non-parallel

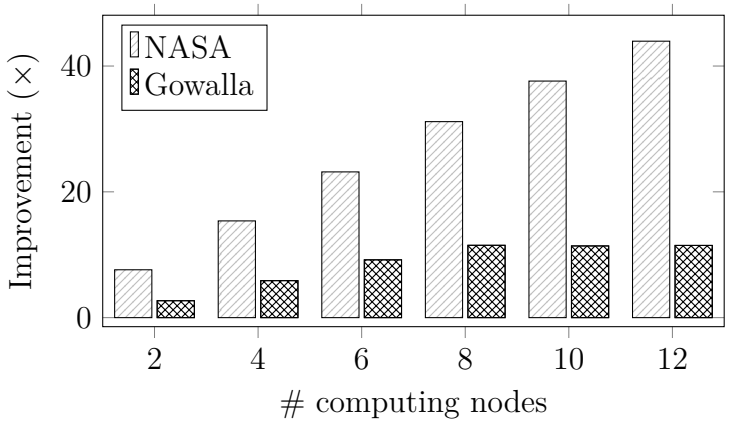


Figure 5.5 – Improvement of FRESQUE, compared to PINED-RQ++

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 5.5 demonstrate the outperformance of 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.

(2) Comparison with parallel PINED-RQ++. The throughput of FRESQUE is always

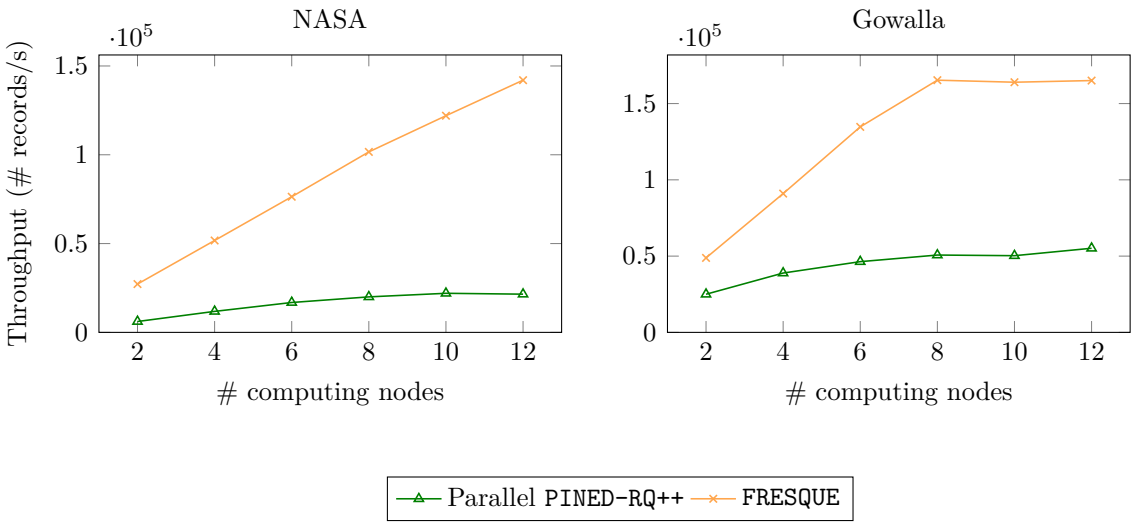


Figure 5.6 – Comparison of ingestion throughput with parallel PINED-RQ++

higher than that of parallel PINED-RQ++ as we vary the number of computing nodes at the collector, as shown in Figure 5.6. The setting of 12-computing node cluster gives the biggest gap, the throughput of FRESQUE is ∼6.6× (NASA) and ∼3.0× (Gowalla) better than that of parallel PINED-RQ++. Noted that since the record size of Gowalla is smaller than that of NASA, the throughput in FRESQUE reaches the peak as we only use 8 computing nodes in Gowalla, thereby the use of more computing nodes does not bring more benefit in this case.

b) 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 5.7, 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.

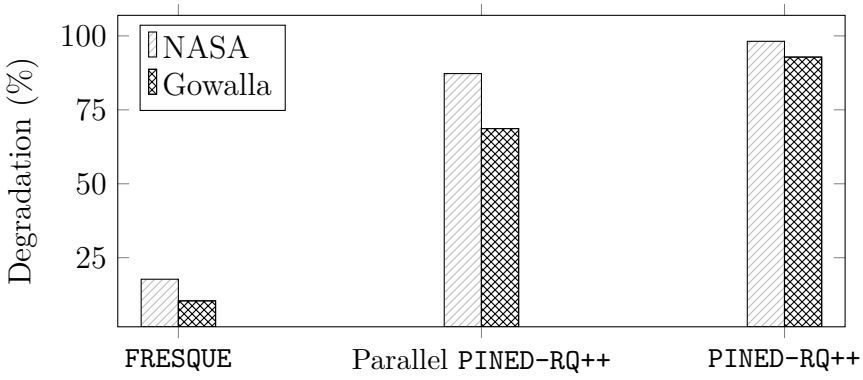


Figure 5.7 – Throughput degradation at the collector

c) Publishing time

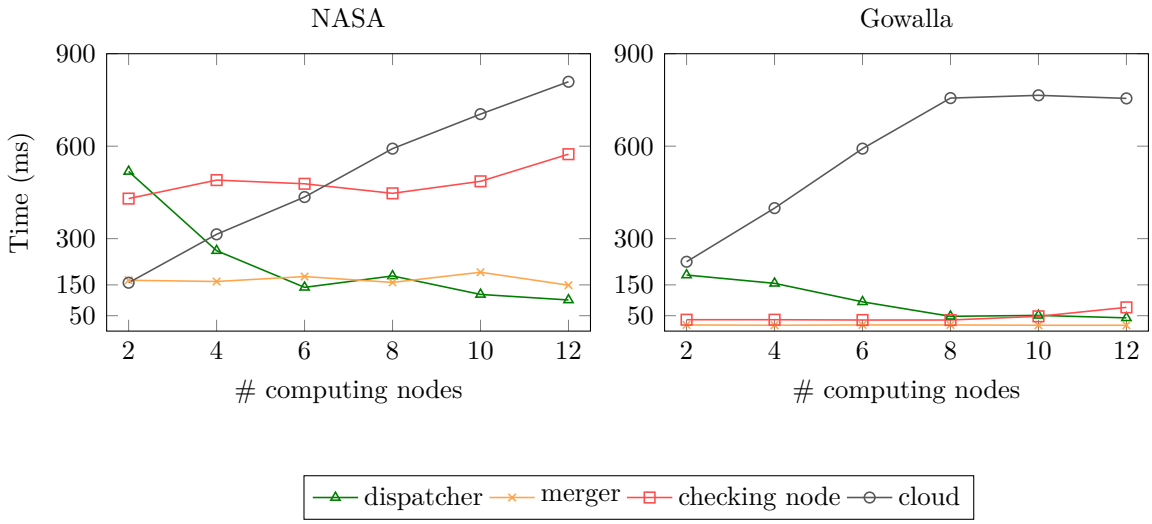


Figure 5.8 – Publishing time in FRESQUE

We now turn our attention into the publishing time metric. That is, the time required to publish a dataset of FRESQUE and 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.

(1) Publishing time at the dispatcher. As shown in Figure 5.8, the time is always lower than 520ms in NASA and 200ms in 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.

(2) Publishing time at the merger. Upon receiving the AL and IT of a publication from the checking node, the merger performs following tasks: combining the AL and IT to get a secure index, inserting removed records into *pre-built* overflow arrays, and publishing them to the cloud. Such tasks are performed independently of the other components at the collector, and may not cause any bottlenecks for the ingesting components (e.g., dispatcher, checking node). However, a small delay at this component would bring much benefit since it allows to early supports indexing over published data. The results in Figure 5.8 indicates that the time is virtually unchanged in the two datasets as their size changes. Specifically, the time in NASA fluctuates between 149ms and 191ms while that in 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.

(3) Publishing time at the checking node. It is worth noting that the data of a new publication is only sent to the cloud as soon as the checking node finishes the publishing job on the previous publication. Thus, we attempt to design `FRESQUE` so that the checking node has a lightweight publishing job and does not impact much 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 time interval. The results in Figure 5.8 give that the time is under 600ms in NASA and 80ms in Gowalla. It is also easy to understand 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 would result in long publishing time at this component, however, thanks to the local buffer of computing nodes, the ingestion throughput is not degraded much. We delay the evaluation of using the randomer in the later part of this section and discuss the scalability of the system with regard to the randomer in Section 5.6.

(4) 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 5.8, the time in FRESQUE goes up according to publication size. This is not because the complexity of the matching algorithm, but the size of the metadata. The larger publication, the larger metadata the cloud maintains, and the longer time the cloud takes to write such metadata to file after it is associated with the corresponding index. 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.

(5) Comparison with parallel PINED-RQ++. We now compare publishing time at

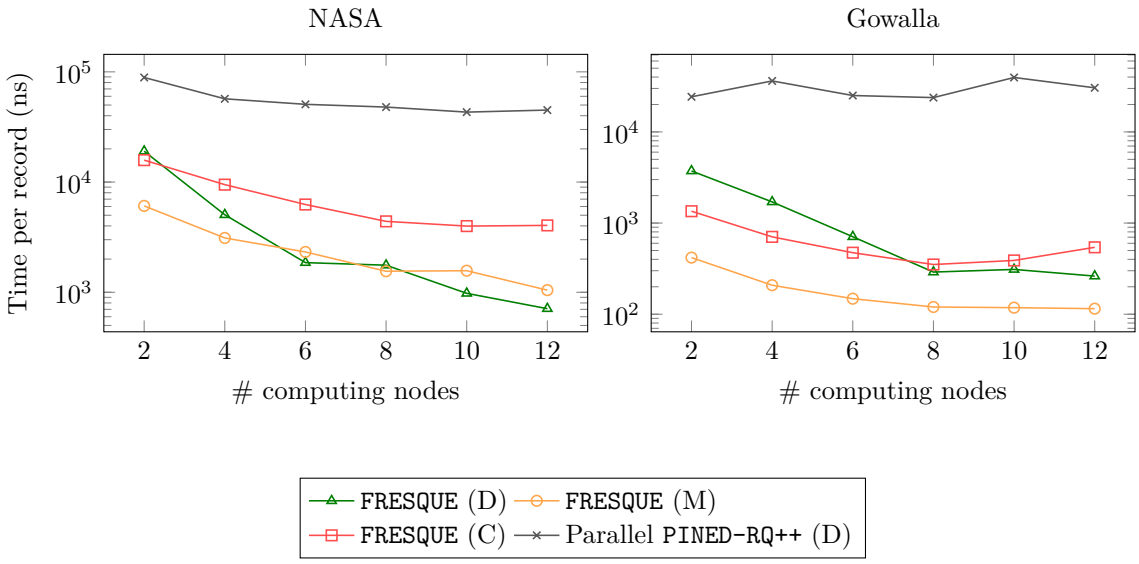


Figure 5.9 – Comparison of publishing time at collector with parallel PINED-RQ++

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 5.9 show that parallel PINED-RQ++ (dispatcher) takes longer delay than FRESQUE (dispatcher, checking node, and merger) for the two datasets used. Regarding the dispatcher, the publishing time of FRESQUE is at most ∼62× and ∼127× lower in NASA and Gowalla,

respectively, compared to parallel PINED-RQ++.

d) Matching time

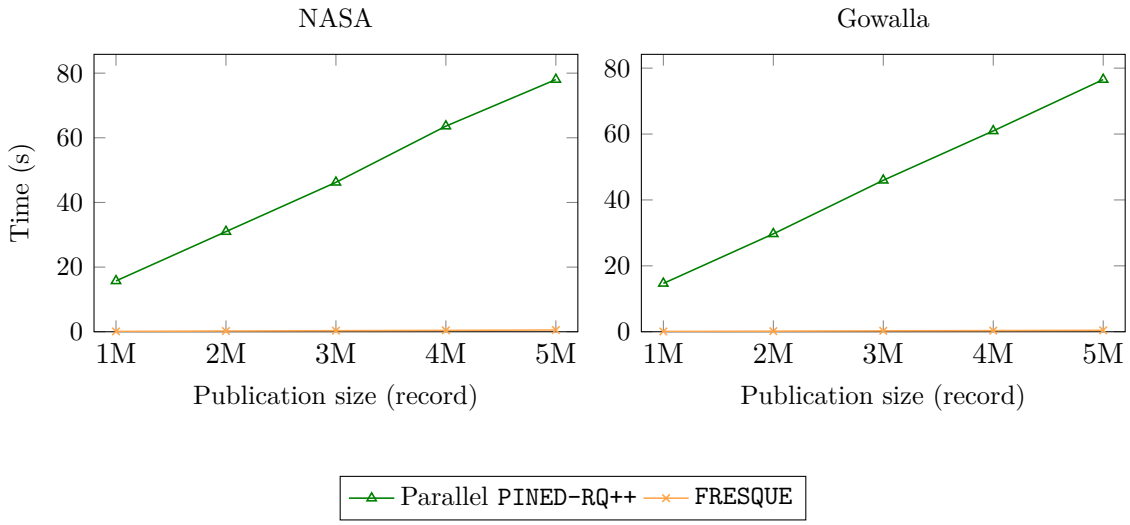


Figure 5.10 – Comparison of matching time at cloud with parallel PINED-RQ++

PINED-RQ++ uses a matching table to keep the relationships (i.e., pointers) between published data and an index template during a time interval. When a matching table is published to the cloud, a *matching job* will immediately be triggered to reconstruct such pointers. The matching job however has a complexity of $O(n^2)$, where n is dataset size. Obviously, as n is large, the cloud needs long time for the matching job, leading to potential bottlenecks at the cloud. We thus evaluate the matching time is needed to process a publication in parallel PINED-RQ++ and FRESQUE. The results in Figure 5.10 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++.

e) Impact of the randomer

Due to privacy dimension, we propose to use the randomer that always maintains a local buffer for perturbing incoming data. A large size of the buffer may introduce a bottleneck at the collector. We thus take the impact of this component into account. 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.

(1) Privacy budget ϵ. We now consider the impact of the randomer in terms of

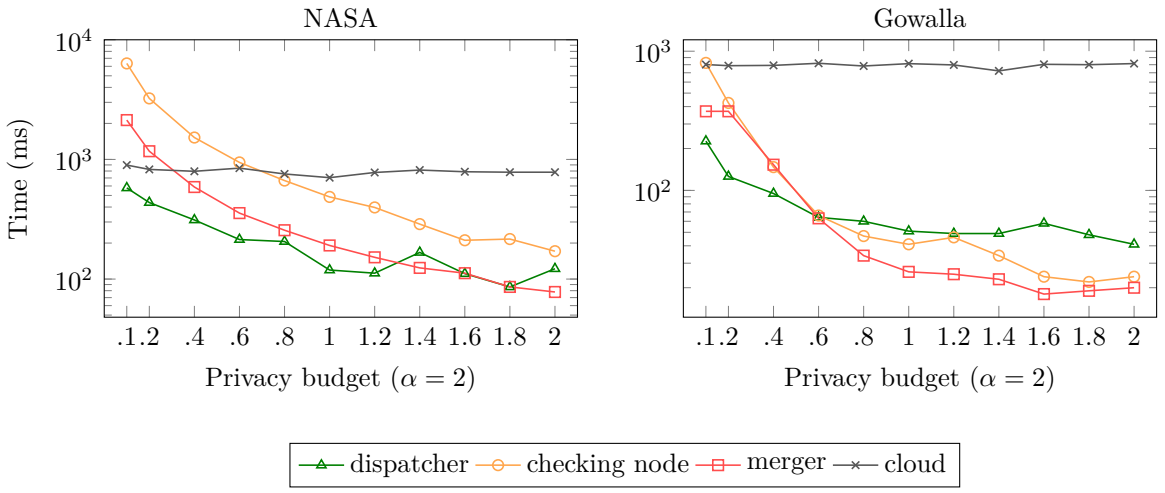


Figure 5.11 – Publishing time with different privacy budgets when the coefficient is set to 2.

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 5.11 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.

(2) Coefficient α. We adjust the value of α to see the impact of randomer on

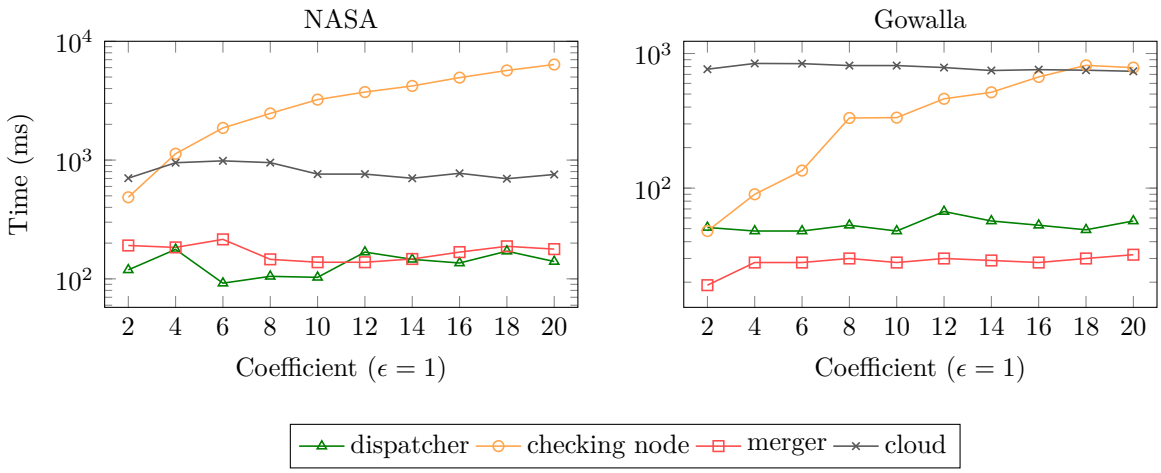
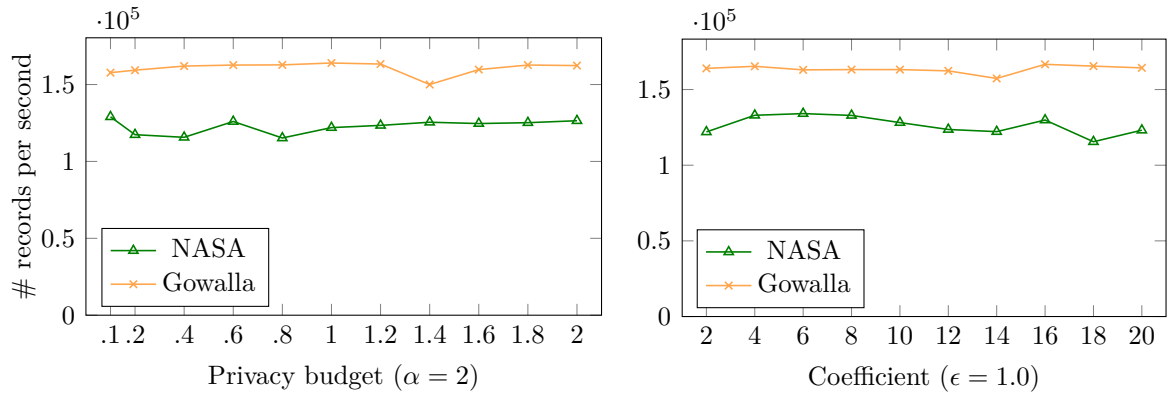


Figure 5.12 – Publishing time with different coefficients when the privacy budget is set to 1.

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 5.12). 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, the merger and the cloud.

(3) Impact of the randomer on ingestion throughput. We also consider the inges-



(a) Varying privacy budgets

(b) Varying coefficients

Figure 5.13 – Ingestion throughput of `FRESQUE` with randomer

tion 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 5.13a and Figure 5.13b, the results show that the throughput in NASA dataset fluctuates between ∼115K records/s and ∼134K records/s while that of Gowalla ranges from ∼150K records/s to ∼166K records/s.

5.6 Discussion

a) An application of FRESQUE

We present a possible real-life application of FRESQUE, that is to say FluTracking [40].

Flutracking [40] 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 [40], 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.

b) Pre-building overflow arrays

One of the challenges of PINED-RQ++ [104] is the time for building overflow arrays. Specifically, when the number of bin of index is large, the time for building overflow arrays is high, resulting in potential bottlenecks. Although FRESQUE avoids bottlenecks due to the asynchronous publishing method, the merger confronts will take time in such case. This thus delays the availability of secure at the cloud. To address it, we can simply pre-build overflow arrays. In particular, at the beginning of each time interval, FRESQUE builds overflow arrays for the publication. However, there may be a situation where the time interval is small and the domain size is large. Pre-building overflow arrays during a time interval may not be enough. To solve it, we can prepare a database of dummy records in advance. The building process of overflow arrays can use such database to boost the building time.

c) Scalability of randomer

The randomer buffer mainly depends on the in-memory storage for fast access. This would raise scalable challenges on the checking node as the system needs a huge buffer. Recently, *persistent memory* (PM) technology [16] has had significant achievements, it features close-to-DRAM and could be larger than DRAM in capacity. Thus, when a randomer buffer is huge, such technology could be taken into account. We would let such extension as one of our future works.

5.7 Conclusion

This chapter presents a scalable ingestion framework for secure range query processing over encrypted data on clouds, FRESQUE. Particularly, we thoroughly analyze and identify the problems, degrading the ingestion throughput, of the-state-of-the-art solutions, especially those of PINED-RQ++. To address these drawbacks, we design a new architecture such that it is fully distributed processing incoming data 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 165K records/s. Moreover, we introduce and carefully integrate the randomer into our new architecture to adapt FRESQUE to a stronger type of adversary (e.g., online-ext) as well as improving the practicality of the framework. We give formal proof to prove the security guarantees of FRESQUE. Lastly, we discuss a potential application of FRESQUE as well as its scalability.

Although FRESQUE allows to efficiently ingest huge arrivals per second, like PINED-RQ++ it supports only a limited number of updates. This is because PINED-RQ, PINED-RQ++, and FRESQUE have to use a portion of the fixed privacy budget for each update to the same individual, and hence running out of budget soon in case of numerous updates. Moreover, such situation can lead to high sensitivity that causes large injected noise to the private index, hence not only destroying the index utility, but also resulting in high overhead of storage. These drawbacks make PINED-RQ impractical to applications where an individual has many records. To deal with these limitations, in Chapter 6, we introduce a novel scheme that practically supports an unlimited number of updates while ensuring good range query performance.

Chapter 6

PARADOT

Abstract: Many studies, including PINED-RQ++ and FRESQUE, have been proposed to address range query processing on encrypted data, however, none of them can meet the requirements of strong privacy protection, efficiency, lightweight updates, and practical space needs. This chapter thus presents a novel scheme for secure range query processing over encrypted data, PARADOT, that practically satisfies all these requirements. This means that PARADOT can achieve both efficiency and strong privacy protection while ensuring practical space overhead and lightweight update operations. To achieve these goals, our scheme relies on equal-size buckets and secure indexes. The former enables to protect the privacy while the latter allows to provide fast range query processing. In addition to these components, we propose to decouple secure indexes from their buckets by using equal-size bitmaps. These bitmaps privately maintain links between secure indexes and buckets of outsourced data. This decoupling approach allows PARADOT to efficiently support unlimited lightweight updates without revealing anything about underlying plaintexts. Moreover, the proposed approach exhibits practical space overhead of encrypted bitmaps. With thorough experiments on real-world datasets, we show that PARADOT outperforms the state-of-the-art solutions in various metrics. For example, as compared to PINED-RQ [94], PARADOT is two orders of magnitude faster in terms of query response latency and uses at most $\sim$111$\times$ less space requirement. More importantly, PARADOT is able to efficiently support numerous lightweight updates, that are either very costly or are not supported by previous schemes.

6.1 Introduction

Many studies have been proposed to address privacy-preserving range query processing on encrypted data. However, they are still far from acceptable solutions. Particularly, none of existing schemes can satisfy the requirements of efficiency, high security, lightweight updates, and small space overhead. Although `PINED-RQ++` and `FRESQUE` can avoid congestion in case of high speed of incoming data, they fail to support numerous updates. These schemes also incur high space overhead when an individual has many records. In this chapter, we thus develop a novel scheme that meets all the target requirements, namely efficiency, high security, lightweight updates, and small space overhead. The efficiency and high security are enabled by the use of existing secure indexes, e.g., PINED-RQ [94], while the scalability in terms of update and space usage relies on encrypted bitmaps. Furthermore, we use the two non-colluding semi-honest cloud providers in this scheme. We emphasize that this model is not new and has widely been used in several recent works such as [112, 76, 37, 69].

We first present an algorithm that partitions a dataset into a set of equal-size buckets. Each bucket is assigned a unique identifier (id). We then use equal-size bitmaps [81, 80] to encode these bucket ids for keeping the relationship between the buckets and distinct values in the indexed attribute. A secure index is then built over these bitmaps instead of buckets or records. These bitmaps are subsequently encrypted by an additively homomorphic encryption scheme (e.g., Paillier [84]) while the buckets are encrypted by a symmetric encryption scheme (e.g., AES in CBC mode [113]). The private index and encrypted bitmaps are finally published to the first cloud while the encrypted buckets sent to the second cloud. By using equal-size bitmaps and equal-size buckets, we prove that the adversary cannot infer any useful information from the published data while the problem of high storage overhead in prior schemes are addressed.

With such an approach, every update to existing publications results in two interactions. One is an appending to all encrypted bitmaps at the first cloud so that the size of all encrypted bitmaps is always kept equal. The second interaction is the sending of the encrypted buckets, that contains the update, to the second cloud. We note that the condition of equal size for bitmaps is necessarily ensured to make updates to existing data indistinguishable. Hence, updates do not disclose any useful information about underlying plaintexts at the first cloud. Similarly, by being separated from the secure indexes, the second cloud cannot also infer any sensitive information when the encrypted equal-size

buckets are arrived. With this updating mechanism, PARADOT is able to efficiently support an unlimited number of updates.

The proposed scheme provides not only efficient secure range query processing, but also high privacy protection, even in case of numerous updates. Experimental results show that PARADOT significantly outperforms prior solutions in various metrics when both uniform and skewed distributions are used. For instance, as compared to PINED-RQ, PARADOT is ∼176× faster in terms of query response latency and uses at most ∼111× less space requirement. Furthermore, with dataset sizes of 0.5M and 5M records, the index size of PARADOT is about 42MB and 406MB while PBtree [66] requires 1.598GB and 18.494GB, respectively. Therefore, we develop **PARADOT**, a novel scheme for **p**rivate and sc**a**lable **ra**nge query processing on encrypte**d** **o**utsourced da**t**a, including the following main contributions.

1. A novel publication approach for efficient range query processing on encrypted data, consisting of a partitioning algorithm, an encoding method based on bitmaps, and a decoupling storage model.
2. A lightweight updating strategy that requires only two simultaneous interactions.
3. A privacy proof for the proposed scheme.
4. A thorough empirical evaluation demonstrating the superiority of our solutions compared to the state-of-the-art solutions such as PINED-RQ [94] and BPtree [66].

The chapter is structured as follows. In Section 6.2, we briefly present the problem statement. We then describe PARADOT in Section 6.3. We analyze the security of PARADOT in Section 6.4. Section 6.5 presents our experimental results. We finally give conclusion in Section 6.6.

6.2 Problem definition

A dataset is a relation $D(A_1, \ldots, A_d)$, where A_i is an attribute. An index can be built over a subset of attributes $A_k = \{A_i, \ldots, A_j\}$ $(1 \leq i \leq j \leq d)$. Queries are non-aggregate one-dimensional range queries. A query $\mathcal{Q}$ is evaluated over A_k. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of distinct values from the domain of A_k. The dataset may contain multiple records that have the same value v_i on A_k.

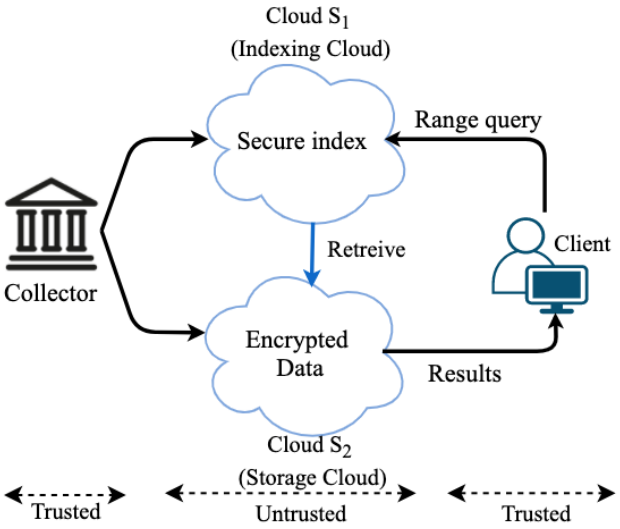


Figure 6.1 – Overview architecture of PARADOT

6.2.1 Overview architecture of PARADOT

We consider the architecture as depicted in Figure 6.1. We model our solution with an assumption of the existence of two non-colluding *semi-honest* cloud providers, Cloud S_1 and Cloud S_2, where Cloud S_1 holds private indexes and Storage Cloud stores encrypted records. We refer Cloud S_1 as *Indexing Cloud* and Cloud S_2 as *Storage Cloud*. We emphasize that there are several cloud services that are often provided by large companies, namely Amazon, Google, and Microsoft. A collusion among them is highly unlikely to happen as it will destroy their reputation.

In this architecture, data is first gathered at the collector. Periodically, the collector independently pre-processes the collected data. This process partitions the dataset into equal-size buckets and creates two complementary data structures, a secure index (e.g., PINED-RQ [94], Logarithmic-SRC [32], or Arx [86]) and a set of bitmaps. The index is built on these bitmaps which keep private links to buckets. Before being published, the buckets are encrypted by an encryption scheme while the bitmaps are encrypted by an additively homomorphic encryption scheme (e.g., Paillier [84]). Then, the private index is published to Indexing Cloud along with the encrypted bitmaps while the encrypted buckets are transferred to Storage Cloud. A client with a private key can send a range query to Indexing Cloud. Based on the encrypted bitmaps and the private index, Indexing Cloud obtains the links to relevant buckets. These links are then sent to Storage Cloud for retrieving the corresponding buckets before returning them to the client. The returned

buckets are lastly decrypted for final results. In our architecture, we consider the two clouds as untrusted components while the others are trusted. This chapter considers the same threat model as the previous one (Section 5.2.1).

6.2.2 PINED-RQ index

In this chapter, we integrate PINED-RQ index into our scheme to facilitate the query processing. We choose PINED-RQ [94] due to the fact that it ensures efficiency and formal security guarantees while a clear secure index requires very small space. Unfortunately, it encounters high sensitivity as an individual links to many records. To get rid of the privacy budget issues in PINED-RQ, we only consider indexing *identity* attributes. A value of such attribute allows to identify uniquely an individual. It can be direct identifiers (e.g., driver's license, bank account number, social security number, etc.) or a group of quasi-identifiers (e.g., zip code, race, gender, and date of birth, etc.). Interestingly, our approach allows PINED-RQ to support unlimited number of updates in that setting.

6.3 PARADOT

This section overviews PARADOT. In particular, given a dataset, $\mathcal{D}$, the collector first partitions it into a set of buckets. Each bucket has a unique id and only contains records that have the same value of the indexed attribute, A_k. However, records with the same value of A_k may be distributed to more than one bucket. The number of buckets thus varies according to the evolution of the data collection. The collector then creates bitmaps on the bucket ids to keep links to these buckets. To do that, for each unique value v_i of A_k, a bitmap is built to encode the ids of the buckets that contain records whose indexed value is v_i. Then, a private index (e.g., PINED-RQ [94]) is built over the created bitmaps instead of records. Lastly, PARADOT uses a symmetric encryption scheme (e.g., AES) to encrypt the dataset (now it is buckets) and a partially homomorphic encryption scheme (e.g., Paillier) to encrypt the bitmaps before publishing them to the clouds, as illustrated in Figure 6.2.

Key management. In PARADOT, the public key, k_1^{pub}, of the PHE scheme is held at the collector and Indexing Cloud while the corresponding private key, k_1^{priv}, is shared with Storage Cloud. With such an approach, after bitmaps are encrypted by

the collector, Indexing Cloud is able to perform the addition of the encrypted bitmaps, but cannot know the underlying plaintexts. Only Storage Cloud can decrypt the data encrypted by k_1^{pub}. On the other hand, the secret key, k_2, of the symmetric encryption scheme is shared among the two components, namely the collector and the client, for encrypting/decrypting buckets.

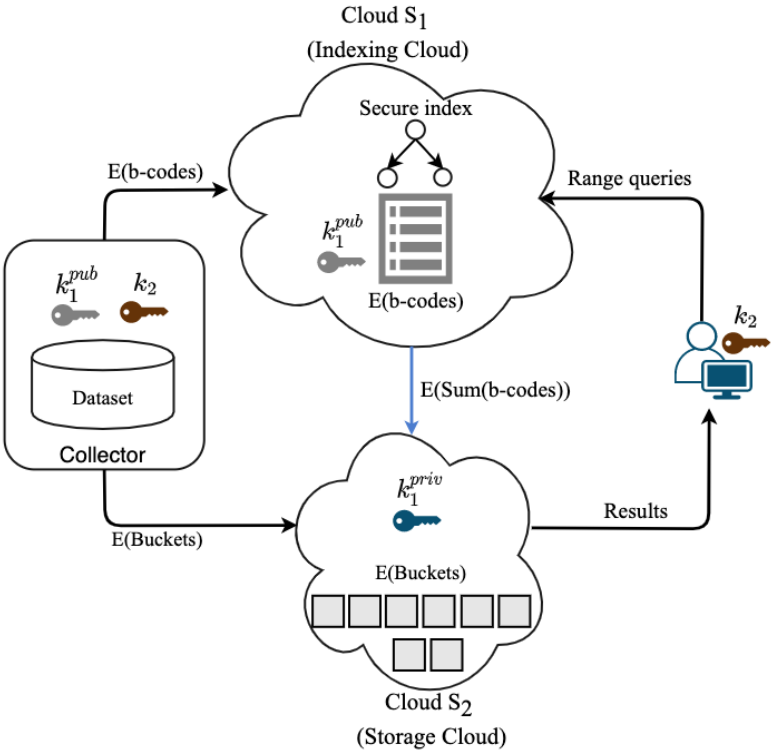


Figure 6.2 – PARADOT protocol

6.3.1 Scheme overview

We now delve into details of how the collector processes and publishes a dataset. This process includes the following main steps.

a) Partition

This step aims to partition a dataset into a set of equal-size buckets so that each bucket contains only records of the same indexed attribute value. However, the challenge is that the input dataset is often non-uniform and results in buckets with different sizes, that can be exploited by the adversary. Hence, our partitioning strategy must ensure that

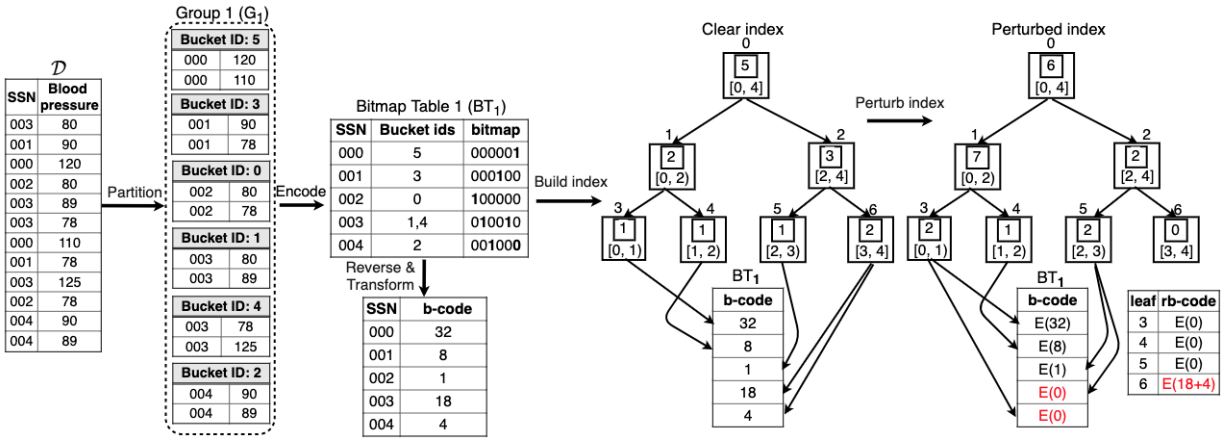


Figure 6.3 – Sample publication

all buckets are equal in size before they are encrypted and published to the cloud. One may propose to pad small buckets with fake data to achieve equal-size buckets. But this approach would incur significant storage overhead, especially when data distribution is highly skewed. Instead, PARADOT seeks to use multiple smaller buckets for values with high frequency. That is, records with the same value of A_k can be distributed to more than one buckets with a smaller size. The question is now how to find a bucket size, s^*, so that all s^*-size buckets completely contains the dataset while avoiding inserting fake data as much as possible. To address it, we design an algorithm that searches for s^*. Basically, our algorithm tries all potential sizes, s_p ($1 \leq s_p \leq s_{max}$), where s_{max} is the maximum frequency of a value in A_k. For each s_p, the algorithm calculates the total number of injected dummy records that are necessary to make the size of output buckets equal. Then, it picks the one that leads to the smallest number of dummy records, as illustrated in Algorithm 1. It is worth noting that since s_p starts from 1, two possible values that the algorithm can return, 1 or the minimum frequency. This also means that with such setting, our protocol does not insert any dummy record into buckets such that they are equal in size. However, this reveals the minimum frequency in dataset. To avoid such leakage, s_p can be initiated by a value larger than 1. Then, equal-size buckets may contain dummy records. We let this choice as an application-specific parameter since the leakage is limited.

After the size, s^*, is found, PARADOT simply obtains n s^*-size buckets for all values in A_k. The i-th bucket is assigned a unique random identifier q_i, where $q_i \in [0, n-1]$. As demonstrated in Figure 6.3, there are in total 6 buckets for the whole toy dataset, $\mathcal{D}$, and each bucket has size of 2. Finally, we obtain a list of unique values to which a list of

buckets belongs.

Algorithm 1: Partition

```
input  : A list of potential sizes (slist)
output: A bucket size
smin:= 1;
smax:= Max (slist);
for k ← smin to smax do
    // Find total fake records for each k
    fakeRecordSize ← 0;
    for s ∈ slist do
        numBucket ← s div k;
        if s mod k == 0 then
            numBucket ← numBucket + 1;
        end
        fakeRecordSize ← numBucket ×k − s
    end
    map [k] ← fakeRecordSize ;
end
// Return the element having the least number of fake records
foundSize ← findMin (map);
return foundSize
```

b) Encode and Transform

This step considers using bitmaps [81] to privately keep links to the buckets which are generated in the previous step. More specifically, suppose the dataset $\mathcal{D}$ is partitioned into a set of n buckets, $B = \{B_1, \ldots, B_n\}$, with the corresponding identifiers, $\{q_1, \ldots, q_n\}$. For each distinct value v_i of the indexed attribute, `PARADOT` allocates a bitmap b_i to encode the id of buckets belonging to v_i. The j-th bit of b_i is turned on (true or 1) if and only if there exists a bucket whose id is j belongs to value v_i. The number of bits in a bitmap is equal to the number of buckets of the publication. Due to the same length, these bitmaps become indistinguishable from the adversary after they are encrypted. As an example shown in Figure 6.3, because there are 6 buckets, each bitmap has the length of 6. As the 1-st and 4-th buckets whose SSN is `003`, the corresponding bitmap is `010010`. With this approach, range queries over the indexed attribute can be answered efficiently with these bitmaps. For instance, the range query, (Q_1) *Select * From database Where SSN Between 002 And 003*, can be answered by performing `OR` over the bitmaps of `100000` and `010010`.

The result of such operation is `110010`, and then the buckets whose id is `0`, `1`, and `4` are returned. Nonetheless, the question is how to privately perform such bit-wise operation at the cloud.

One may delegate the bit-wise computation to another trusted component, the collector for example. But such an approach would not be scalable since it results in high communication overhead and puts more pressure on the collector. Instead, we propose to use a partially homomorphic encryption scheme to privately perform that task on the cloud. To achieve it, bitmaps should be converted into numbers that partially homomorphic encryption schemes often work on. Particularly, when all bucket ids are encoded, the bitmaps are then transformed into numerical representations. In other words, the collector first reverses the order of bits in a bitmap. The reversed bitmap is now considered as a binary number and is transformed into the corresponding decimal representation, hereafter referred to as *b-code* (bitmap code). For example, considering the first bitmap (`000001`), its binary representation (reversed bitmap) and its b-code are `100000` and `32`, respectively. This step finally outputs a *bitmap table*, BT, where the first column contains the indexed attribute values and the second column holds the corresponding b-codes. To answer the query Q_1 with b-codes, `PARADOT` now performs the sum operation over b-codes of the values between `002` and `003`, particularly `1` and `18`, to obtain `19`. This number is then converted to binary representation (`010011`) and reversed to the corresponding bitmap (`110010`). Based on that bitmap, the corresponding buckets (`0`, `1`, and `4`) are returned.

c) Build a private index

We now describe how to build a secure index on the b-codes obtained from the previous step, particularly the PINED-RQ index [94]. More importantly, with our approach, PINED-RQ gets rid of the problem of high sensitivity and avoids a huge number of injected dummy data. The main reason is that a distinct value in the indexed attribute (or an individual) is represented by only one b-code, and when the PINED-RQ index is built on such b-codes, the sensitivity is constantly one.

Build a clear index. The process of building a clear index in `PARADOT` is the same as that in PINED-RQ [94]. The main difference is that instead of building an index over the records of a dataset, `PARADOT` constructs it over the corresponding bitmap table BT. More precisely, a clear index will be constructed over the indexed attribute A_k of

that bitmap table.

Note that since each row in the bitmap table represents only one unique value of the indexed attribute (or an individual in case of identity attributes), the absence/presence of an individual will change the counts on the corresponding path from root to leaf by at most 1. This makes the sensitivity of secure index constant ($\Delta f = 1$) and independent of the frequency in A_k. Such property allows PARADOT to avoid significantly injected noise as the Laplace mechanism is applied to perturb the index. Considering the example in Figure 6.3, PINED-RQ needs to set the value of the sensitivity parameter to 4 to guarantee privacy protection while that is only 1 in PARADOT.

PARADOT also creates links from the index leaves to the corresponding rows in the bitmap table. After these links are formed, the first column of the bitmap table is removed. This means that the index leaves only have references to b-codes at the end of this process, as illustrated in Figure 6.3.

Perturb the index. Before publishing the index to Indexing Cloud, the collector needs to perturb it with the Laplace noise as in PINED-RQ. However, the noise is independently added to each index node and may be positive or negative, leading to inconsistency between the count of a leaf and its pointers to b-codes. To deal with this issue, PINED-RQ adds dummy records in case of positive noise and removes real records in case of negative noise. Removed records will be concealed into the corresponding overflow arrays. Our approach also maintains such operations to ensure the consistency. However, the query processing needs to add all b-codes of queried leaves together, including dummy and real ones, such dummy b-codes may change the information encoded in real ones. Thus, to make sure dummy b-codes have no impact when they are added to real b-codes, we propose to used zero b-codes (see Figure 6.3).

Considering positive noise, when a leaf receives a positive noise c, PARADOT then randomly adds c zero b-codes (dummy data) to the existing set of b-codes and links them to that leaf. For example, since node 3 receives positive noise (+1), one zero b-code is generated and randomly linked to that node.

Regarding negative noise, we can indeed use overflow arrays for hiding removed b-codes as in PINED-RQ, however, these overflow arrays unnecessarily result in a large number of dummy records. Instead, PARADOT adds all removed b-codes of a leaf into a single b-code, that is considered as the *rb-code* (removed b-code) of that leaf. As demonstrated in Figure 6.3, two b-codes are removed from node 6 due to the negative noise (-2). These

removed b-codes and its zero rb-code are then summed together to get a final rb-code (22). It is also worth noting that since the id of buckets is unique, all removed b-codes of a leaf can be completely hidden into a single rb-code without incurring overflow or losing the information held by b-codes (see Section 6.3.4). This differentiates from PINED-RQ overflow array, that only ensures to hide all removed records with an adjustable probability. Moreover, our approach now needs only one number for each leaf and does not incur any dummy data for concealing removed b-codes while PINED-RQ would add a large amount of dummy data as high security is required.

d) Publish

PARADOT encrypts all b-codes, rb-codes, and buckets before sending them to Indexing Cloud and Storage Cloud, respectively. In particular, the collector shuffles and encrypts the (r)b-codes by using the public key, k_1^{pub}, and all buckets by using the secret key k_2. The encrypted (r)b-codes and the private index are published to Indexing Cloud while the encrypted buckets are shuffled and sent to Storage Cloud. Note that although the id of buckets is kept in clear as they published, no sensitive information can be exploited from such ids since they are random. As discussed in Step a), the only thing that the adversary can learn from these buckets is the minimum frequency.

6.3.2 Updates

This section describes how PARADOT supports updates to existing publications. The updating feature here includes insertion, modification, and deletion. PARADOT considers updates as new inserts of new records to the existing publication and each new record contains an extra attribute to indicate the corresponding action on it, namely insertion, modification, or deletion. Based on this extra attribute, when the consumer receives results, modifications and deletions are then processed locally. Since PARADOT stores encrypted dataset (buckets) on Storage Cloud and private index on Indexing Cloud, updates also require interactions to both of them. We consider the two following cases.

a) Updates of existing values

Given a delta dataset, $\Delta\mathcal{D}$, that contains updates to existing values of the indexed attribute, the collector partitions the delta dataset into buckets, then it builds the corresponding bitmap table. After that, the collector encrypts these components and sends

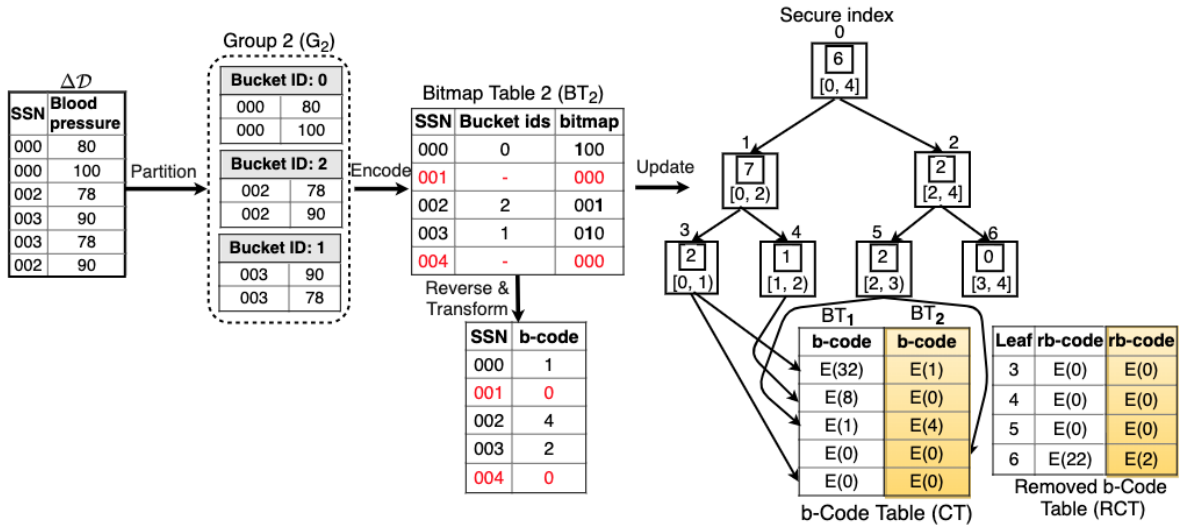


Figure 6.4 – Updating the existing index that is presented in Figure 6.3

them to Storage Cloud and Indexing Cloud, respectively. Apparently, the updating process of existing values is similar to the publishing process of a new dataset, however, the difference is that it does not require building a new secure index. The question is how to manage several updates, including (r)b-codes and buckets, of an existing publication at the clouds.

To address it, PARADOT uses sequential numbers to track updates to the same publication. In particular, for a new publication/update, a sequential number is generated and tagged to their components (e.g., private index, a set of buckets and (r)b-codes), as shown in Figure 6.4. The (r)b-codes of an update will be appended to the (r)b-codes of the previous one of the same index at Indexing Cloud. Such organization subsequently forms tables of b-codes and r-codes, denoted as CT and RCT, respectively, for each index at Indexing Cloud. Each CT row contains b-codes of an indexed attribute value or dummy b-codes and each column (from the second one) implies the corresponding update to that index. Similarly, each RCT row contains r-codes of a leaf and each column (from the second one) represents the corresponding update.

With this organization, when a range query reaches leaves of an index, the corresponding rows in the tables CT and RCT are retrieved, and the sum is performed over cells of the same column. This outputs a row of encrypted sums, each is associated with its sequential number. When this row is sent to Storage Cloud, it is decrypted and Storage Cloud is able to find the buckets belonging to each update. Since the addition of columns is independent, PARADOT can take advantages of parallelism to boost the summing com-

putation.

b) Inserts of new values

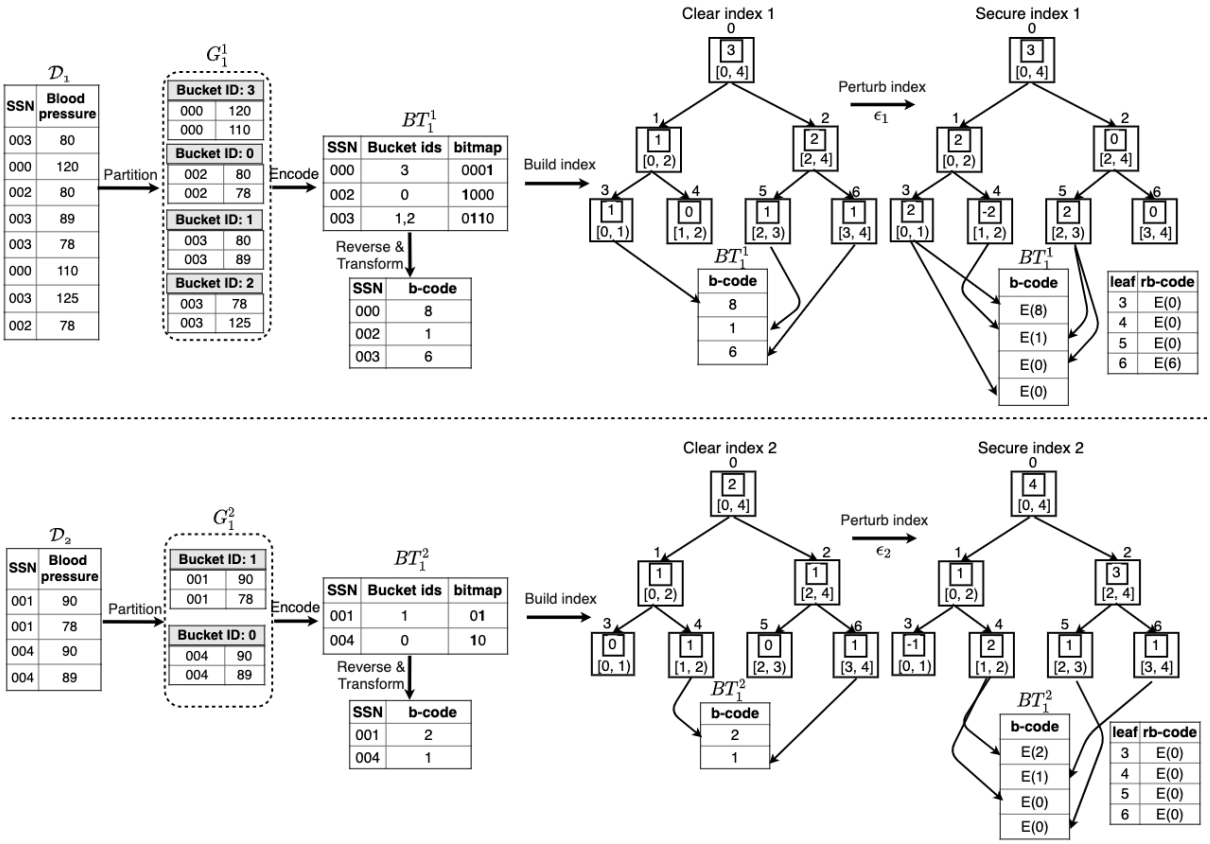


Figure 6.5 – Inserts of new values: Two toy datasets, D_1 and D_2, are disjoint such that two distinct privacy budgets ϵ_1 and ϵ_2 are in turn used to build the corresponding secure indexes.

In case of new values, PARADOT simply treats records of these values as a new dataset and publishes it as a new publication. This means that, PARADOT needs to build a new secure index for such new publication. With regards to PINED-RQ index, we can use the whole privacy budget to perturb the index since datasets are disjoint. As a simple example illustrated in Figure 6.5, there are two datasets D_1 and D_2 are disjoint and assume that D_2 contains updates of new values. In this case, we use the whole privacy budget ϵ_2 to build a new secure index for this dataset.

6.3.3 Unbounded bitmap size challenge

Note that the domain of b-codes is mainly decided by the size of bitmaps (or the total number of generated buckets). However, the latter varies according to the evolution of data. In the worst case, the largest number of buckets can be achieved when the bucket size is one. This means that the encoding process may produce huge b-codes (huge integers). The difficulty is that partially homomorphic encryption schemes like Paillier [84] only properly encrypts/decrypts bounded values for a given public key. This means that it does not properly work on b-codes whose value exceeds the upper bound of a generated key. Using only one key that can encrypt huge numbers for all publications can solve the challenge, however, this solution causes inefficient computations when publications have very small number of buckets. Likewise, we can adaptively use different keys for different publications, however, this approach would create unnecessary complexity for the key management.

To address this, `PARADOT` uses a fixed-domain key and smaller groups of buckets. In particular, suppose the maximum value that Paillier [84] can encrypt/decrypt is k_u bits, this means that bitmaps can maximally encode $k_u - 1$ bucket ids. If the number of buckets is larger than $k_u - 1$, they will be split into smaller groups of buckets so that the size of each group is less than or equal to $k_u - 1$. The first group is considered as a new publication while the rest is updates to this publication. Regarding the example in Figure 6.3, assume that the value of k_u is 4 bits, then the original group of buckets is split into two groups whose size is 3 (see Figure 6.6). `PARADOT` then constructs a secure index over the first group G_1 while the second group G_2 is considered as an update to that secure index.

6.3.4 Query processing

As illustrated in Figure 6.2, when a range query is posed by the client, it is sent to Indexing Cloud. The query is then evaluated over private indexes as in PINED-RQ (see Section 3.5). This evaluation results in a set of encrypted (r)b-codes retrieved. Indexing Cloud performs the addition of these (r)b-codes to obtain the encrypted sums which are subsequently forwarded to Storage Cloud. Upon receiving these encrypted sums, Storage Cloud uses the private key, k_1^{priv}, to performs a series of operations on them such as decrypting, reversing, and transforming to get the corresponding bitmaps. Based on these bitmaps, the corresponding encrypted buckets are retrieved and returned to the client.

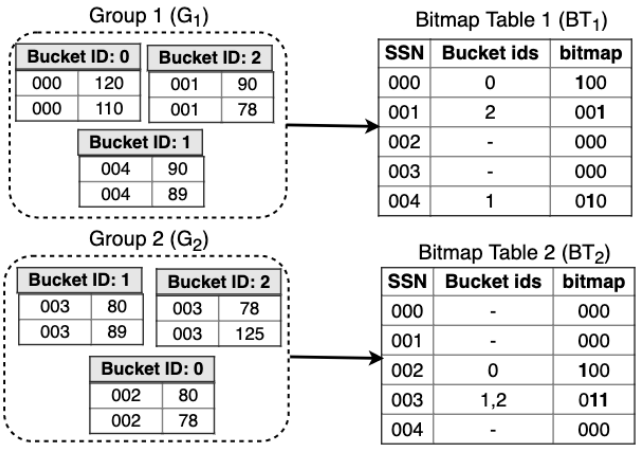


Figure 6.6 – Split large group of buckets into smaller ones

The client lastly uses the secret key k_2 to decrypt the returned buckets for the final results.

Correctness. We now prove the correctness of the query processing in PARADOT. Let C be a function that converts a binary number to the corresponding decimal representation. Assume that there are n bitmaps of size m, that are reversed (R) and transformed (T) as belows.

$$\begin{array}{ccccc} a_{11}a_{12}\dots a_{1m} & & a_{1m}\dots a_{12}a_{11} & & c_1 = C(a_{1m}\dots a_{12}a_{11}) \\ a_{21}a_{22}\dots a_{2m} & \xrightarrow{(R)} & a_{2m}\dots a_{22}a_{21} & \xrightarrow{(T)} & c_2 = C(a_{2m}\dots a_{22}a_{21}) \\ \dots & & \dots & & \dots \\ a_{n1}a_{n2}\dots a_{nm} & & a_{nm}\dots a_{n2}a_{n1} & & c_n = C(a_{nm}\dots a_{n2}a_{n1}) \end{array} \tag{6.1}$$

, where $a_{ij} \in \{0, 1\}$ is the j-th bit of bitmap i ($1 \leq j \leq m$ and $1 \leq i \leq n$).

Traditionally, to answer a range query by using these bitmaps, a logical operation such as OR is applied on relevant bitmaps to get a resulting bitmap, denoted B. Based on this returned bitmap, the corresponding buckets are retrieved. By contrast, PARADOT performs the addition of b-codes of these bitmaps, then it transforms and reverses this sum to the resulting bitmap, denoted B'. We now prove that B' is equal to B.

First, let b'_i be the sum of all bits at the position i of n bitmaps.

$$\begin{aligned} b'_1 &= \textstyle\sum_{i=1}^{n} a_{i1} \\ b'_2 &= \textstyle\sum_{i=1}^{n} a_{i2} \\ &\dots \\ b'_m &= \textstyle\sum_{i=1}^{n} a_{im} \end{aligned} \tag{6.2}$$

Since a b-code is decimal representation of the corresponding reversed bitmap (or binary number), the addition of b-codes is the same as the addition of the corresponding binary numbers (*). Additionally, these bitmaps are used to encode unique random identifiers. This means that at the same position of n bitmaps, there is at most one bit 1 and at least $(n-1)$ 0s. The addition of these binary numbers does not create any *carry* (**). From (*) and (**), we can write the sum of n b-codes as $b'_m \dots b'_2 b'_1$ and $C(b'_m \dots b'_2 b'_1) = \sum_{j=1}^{n} c_j$. When the sum is obtained, it is reversed to get $B' = b'_1 b'_2 \dots b'_m$.

To obtain B, we perform a logical operation `OR` ($|$) on n bitmaps. Similarly, since identifiers are unique, $B = b_1 b_2 \dots b_m$ and

$$\begin{aligned} b_1 &= (a_{11}|a_{21}|\dots|a_{n1}) = \textstyle\sum_{i=1}^{n} a_{i1} \\ b_2 &= (a_{12}|a_{22}|\dots|a_{n2}) = \textstyle\sum_{i=1}^{n} a_{i2} \\ &\dots \\ b_m &= (a_{1m}|a_{2m}|\dots|a_{nm}) = \textstyle\sum_{i=1}^{n} a_{i1} \end{aligned} \tag{6.3}$$

From (6.2) and (6.3), we have

$$\begin{aligned} b_1 &= b'_1 \\ b_2 &= b'_2 \\ &\dots \\ b_m &= b'_m \end{aligned} \tag{6.4}$$

We can then conclude that $b_1 b_2 \dots b_m = b'_1 b'_2 \dots b'_m$ (or $B = B'$)

6.3.5 Space overhead of bitmaps

Suppose a publication/update has b buckets. Each distinct value in A_k is associated with a b-bit bitmap and a leaf is allocated an additional b-bit r-code for hiding removed b-codes. Let m and l be in turn the total number distinct values in A_k and the total

number of leaves, then the storage overhead in plaintext at Indexing Cloud is $b * (m + l)$ bits. These bits must be encrypted by a PHE scheme before being published. Hence, the actual storage overhead of such bitmaps is $O(|k_1^{pub}| * (m + l))$, where k_1^{pub} is the public key of a PHE scheme used to encrypt a b-bit number. As experimentally demonstrated in Section 6.5.6, this overhead is very small as compared to the state-of-the-art solutions, especially when a uniform distribution is used.

6.4 Security analysis

We now analyze the privacy protection of PARADOT. Since PARADOT uses the two clouds model, both Indexing Cloud and Storage Cloud are necessarily considered in this analysis.

Theorem 2 (Security of PARADOT): Security guarantee at Indexing Cloud relies on the used secure index, particularly ϵ_n-SIM-CDP in case of PINED-RQ. Meanwhile, PARADOT ensures semantic security at Storage Cloud with a limited leakage (e.g., the minimum frequency of data) as the minimum of bucket size is one.

Proof. **Indexing Cloud**. PARADOT exposes two main data structures, secure index and encrypted b-codes, to Indexing Cloud. Since the secure index is perturbed by the Laplace noise, this data structure itself satisfies ϵ-differential privacy. This index is linked to a set of b-codes that are encrypted by a partially homomorphic encryption scheme (e.g., Paillier). Such encryption scheme also provides semantic security [84]. Moreover, since removed b-codes of a leaf are completely added into a complimentary rb-code, that is also encrypted before being published, they also ensure semantic security. Hence, PARADOT meets ϵ_n-SIM-CDP at Indexing Cloud.

Storage Cloud. We consider the case where the collector sends buckets to Storage Cloud. We note that before sending buckets to Storage Cloud, the collector encrypts them by using a semantic encryption scheme (e.g., AES in CBC mode) and then shuffles the encrypted buckets. Additionally, the identifier of the published buckets is a unique random number. Thus, the adversary cannot infer any information from these ciphertexts and random identifiers. As a result, PARADOT meets semantic security at Storage Cloud. However, as the minimum size of bucket is equal to one, the partitioning algorithm reveals the minimum frequency in dataset, as discussed in Section a). □

Updating function. We also consider the case where updates are sent to the clouds. Indeed, every update sends encrypted b-codes and encrypted buckets to Indexing Cloud and Storage Cloud, respectively. At Indexing Cloud, the b-code table CT of an existing index is appended by a column that contains the corresponding updating b-codes. Although attackers are aware of some modifications to existing indexed values, they can not determine which one is updated due to equal-size b-codes. Meanwhile, the process of sending the corresponding buckets to Storage Cloud is similar to that of publishing a new publication. Therefore, the privacy protection of such operations is also guaranteed in `PARADOT`.

Query processing. When a range query arrives at Indexing Cloud, after scanning indexes, `PARADOT` directly performs the addition of the corresponding encrypted b-codes. This operation enables to achieve the encrypted sums of b-codes without revealing any useful information about underlying plaintexts. Upon receiving the encrypted sums from Indexing Cloud, Storage Cloud uses the private key, k_1^{priv}, to decrypt them for retrieving the corresponding buckets. Note that since Storage Cloud only receives encrypted sums from Indexing Cloud, the adversary does not know which b-codes contribute to those sums. In other words, she cannot infer to which leaf/attribute value a fetched bucket belongs. In addition, the posed range queries are inaccessible to Storage Cloud. Therefore, the adversary at Storage Cloud cannot infer any useful information from these sums. However, since the same query will result in the same retrieved buckets at Storage Cloud, `PARADOT` is vulnerable to access pattern attacks at this component. To deal with it, ORAM such as [99, 95] can be taken into account. Note that the integration of such approach is beyond the scope of this thesis.

Comparison to PINED-RQ(++). PINED-RQ satisfies $(\epsilon, \delta)_n$-Probabilistic-SIM-CDP privacy model [94] while `PINED-RQ++` [104] meets that model with an assumption of the existence of the two non-colluding clouds. In contrast, `PARADOT` meets ϵ_n-SIM-CDP at Indexing Cloud, semantic security at Storage Cloud, and only discloses the minimum frequency to the adversary at Storage Cloud. Note that this limited leakage can be eliminated when the size of buckets is larger than one. Apparently, with such setting and the assumption about two non-colluding clouds ensured, `PARADOT` provides stronger privacy protection than PINED-RQ(++) since `PARADOT` does not rely on the δ parameter, that is introduced by the use of overflow arrays in PINED-RQ(++).

6.5 Validation

6.5.1 Benchmark environment

For the sake of simplicity, we use LevelDB [62] to store data at cloud servers. It is an open-source on-disk key-value store developed by Google. At Indexing Cloud, secure indexes are persisted into separate files while their b-code tables are put in LevelDB. Each cell (encrypted b-code) in this table forms a key-value pair in LevelDB. Particularly, the key of a pair is formed by the combination of four components, namely index identifier, leaf identifier, the column and row number of the corresponding b-code. Meanwhile, the value is the content of a cell in the table CB. On the other hand, Storage Cloud needs to store bunches of encrypted buckets, each is associated with a sequential updating number. The value is the ciphertext of a bucket while the key of a bucket in LevelDB is the combination of three components, namely publication (or index) identifier, group identifier, and bucket identifier.

`PARADOT` is implemented in Java. All experiments in this study are run on the private cloud Galactica [67]. Specifically, the whole system is configured as a set of several virtual machines running on the Ubuntu 14.04.4 LTS. Each component is run on an independent machine. The configuration of instances is detailed in Table 6.1. The TCP socket is used for exchanging data among these components.

Table 6.1 – Experimental environment of `PARADOT`

Component	vCPU (2.4 GHz)	Memory (GB)	Disk (GB)
Collector	4	8	80
Consumer	4	8	80
Indexing Cloud	8	16	80
Storage Cloud	8	16	80

6.5.2 Datasets

We use two real datasets, UMass Smart [107] and ExtraSensory [110, 109], for all experiments. UMass Smart dataset, hereafter referred to as Smart dataset, contains 57,543,030 records of electricity usage of 114 single-family apartments in 2016. ExtraSensory dataset, hereafter referred to as Sensor dataset, consists of 377,346 records of 60 users. Records

are sampled from user's smart phone. We construct a private index over the identifier of apartments (Smart) and users (Sensor).

6.5.3 Settings

The branching factor (bf) is set to 16 and the total privacy budget ϵ_{total} to 1. The domain of A_k is [1, 114] and [1, 60] for Smart and Sensor datasets, respectively. The security parameter of Paillier scheme is set to 1024 bits.

6.5.4 Query Set

In the experiments, we create various query sets of ranges corresponding to 1%, 5%, 10%, 25%, 50%, and 75% of the entire domain. For each set of query ranges, we sample 1000 queries uniformly over the domain. All experiments are conducted using a uniform workload.

6.5.5 Evaluation metrics

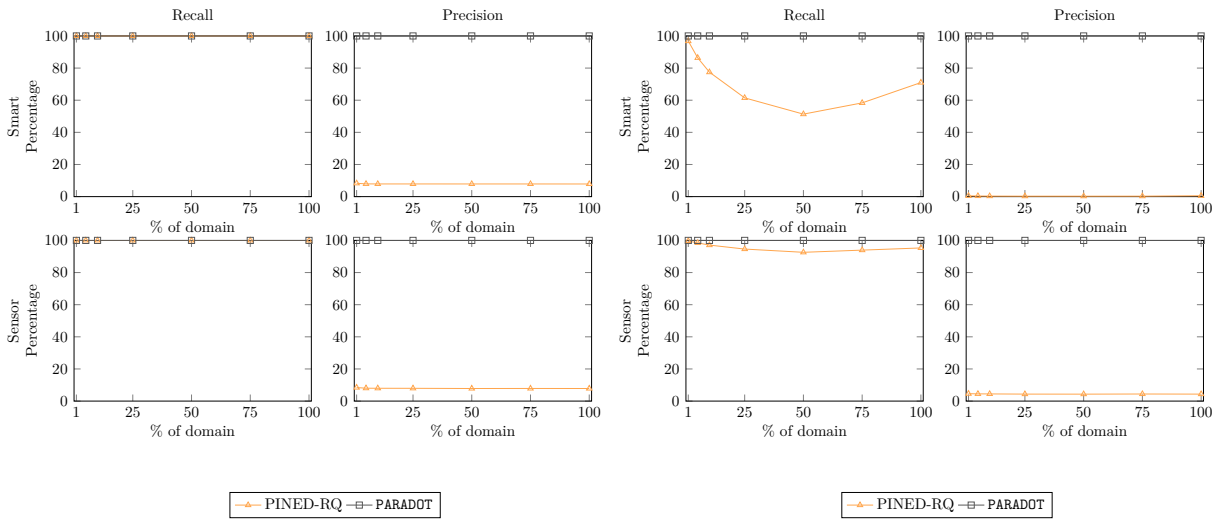
We evaluate PARADOT on three main metrics, namely storage overhead, communication costs, and query latency. We seek to compare PARADOT's performance with its counterparts such as PINED-RQ [94], PBtree [65], and IBtree [64]. Additionally, we take the recall and precision metric [94] into account in order to compare PARADOT with PINED-RQ.

6.5.6 Experimental results

a) Recall and precision

This section compares the recall and precision rate between PARADOT and PINED-RQ. More precisely, we consider two distinct scenarios, uniform and non-uniform distribution of datasets.

Uniform. We first consider the uniform scenario where all indexed attribute values have the same number of records, particularly 1000 records per value, for these experiments. The results in Figure 6.7a show that PARADOT always provides better precision, but equal recall as compared to PINED-RQ. For example, PARADOT constantly holds 100% of recall and precision for all experiments. The reason is that the publications



(a) Uniform (b) Non-uniform

Figure 6.7 – Recall and Precision

of PARADOT do not have any dummy records or inner negative nodes in the index. Despite maintaining the same recall rate as compared to PARADOT, PINED-RQ experiences a very low precision percentage for all range sizes due to high overhead of dummy data, being constantly lower than 8.5% in both datasets. This low precision mainly results from the high sensitivity in PINED-RQ ($\Delta f = 1000$) that causes large injected noise and a huge number of dummy records in the index.

Non-Uniform. With the non-uniform scenario, we use a variant of Smart dataset that consists of 200k records and follows the Zipfian distribution (skew = 1). Meanwhile, the whole Sensor dataset is also used for these experiments since it is naturally non-uniform. Interestingly, PARADOT still achieves 100% of recall and precision rate in both datasets, as shown in Figure 6.7b. This once again proves that the recall and precision of PARADOT are independent of the frequency of indexed attribute values. In contrast, PINED-RQ encounters a substantial decline in recall and precision. The worst recall and precision rate are witnessed in PINED-RQ when the 50%-domain range queries are used. With that setting, the recall rate falls to ∼51.34% (Smart) and ∼92.6% (Sensor) while the precision rate is only ∼0.26% (Smart) and ∼4.23% (Sensor).

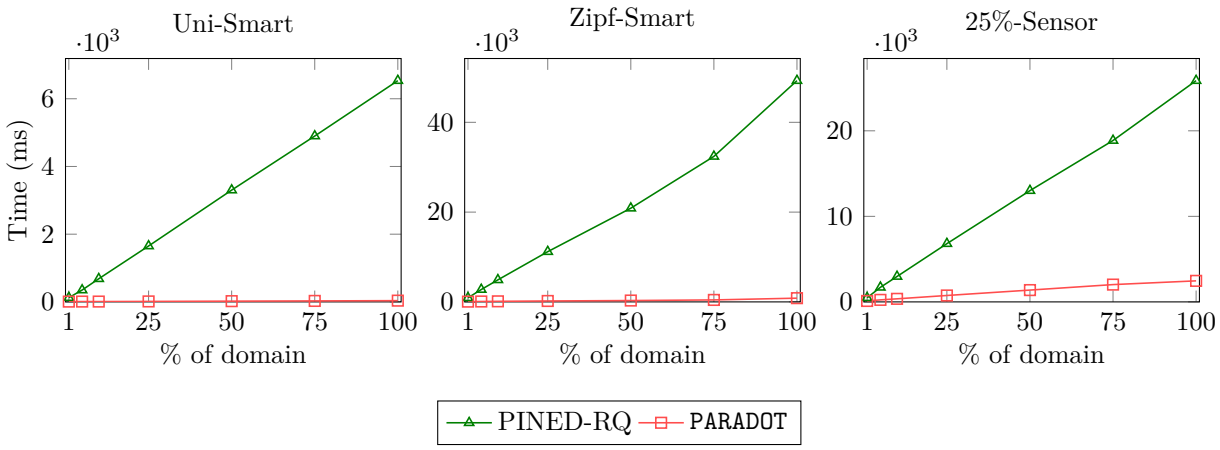


Figure 6.8 – Latency comparison

b) Query latency

We now take the query response latency into account. Moreover, we also compare PARADOT's latency to that of PINED-RQ. To the best of our knowledge, PINED-RQ provides the best index scanning time as compared to the state-of-the-art solutions. Two variants of Smart dataset are used for this scenario, with 144K records per each. One is uniform (1000 records per indexed attribute value), denoted Uni-Smart, and the other adheres to the Zipfian distribution (skew = 1), denoted Zipf-Smart. For Sensor dataset, we extract 25% of the original that is non-uniform as default, named 25%-Sensor. For fair comparison, we also store the publications of PINED-RQ in LevelDB with the same settings. We then measure the time response of 1000 range queries. The average of these measurements is obtained for evaluation.

The results in Figure 6.8 exhibit that PARADOT significantly outperforms PINED-RQ in terms of response time latency, and at most two orders of magnitude faster than PINED-RQ. The non-uniform dataset has higher latency overhead as compared to the uniform one in both prototypes. In particular, the maximum latency in Uni-Smart is ∼6.53s (PINED-RQ) and ∼0.04s (PARADOT) while those of Zipf-Smart is in turn ∼49.28s and ∼0.83s. Nonetheless, PARADOT constantly experiences significant lower latency compared to PINED-RQ. For example, as the largest range is utilized in the uniform scenario, PINED-RQ takes ∼6.53s and being ∼176× longer than that in PARADOT (∼0.04s). On the other hand, in the non-uniform case, the biggest gap is witnessed as the 100%-range query is considered, the average response time is ∼49.28s in PINED-RQ and ∼0.83s in PARADOT. The similar patterns can be seen in 25%-Sensor, PINED-RQ takes at least ∼4.39× (1%

range size) and at most ∼10.51× (100% range size) longer as compared to PARADOT. PINED-RQ suffers from high latency response due to injected dummy data. Moreover, the volume of such injected dummy data is proportional to the number of records per individual (or the sensitivity). Therefore, as a dataset has larger record size and/or higher number of updates per individual, the gap between the two prototypes still rises.

c) Storage overhead

Our solution uses encrypted b-codes to secretly keep links to buckets. These encrypted b-codes mainly contribute to the storage overhead in PARADOT. Meanwhile, the space overhead in PINED-RQ primarily comes from dummy data that is added to dataset as well as overflow arrays. We use four datasets, namely Uni-Smart (2.7MB, 144K records), Uni-Smart* (74.4MB, 5.7M records), Zipf-Smart (2.7MB, 144K records), 25%-Sensor (123.8MB, 93,330 records), and Sensor (490MB, 377,346 records). In PARADOT, we measure the storage overhead at both Indexing Cloud and Storage Cloud. The overhead at Indexing Cloud is primarily caused by encrypted b-codes while that at Storage Cloud results from the encryption of buckets.

Storage overhead of PARADOT. The results in Figure 6.9 demonstrate that when

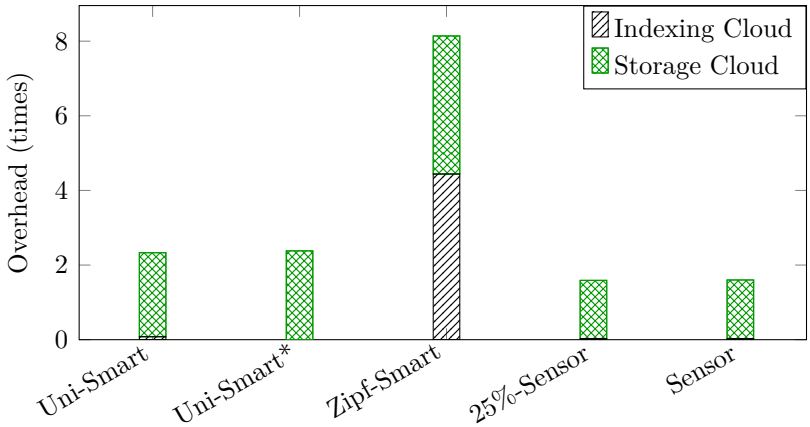


Figure 6.9 – Storage overhead of PARADOT

data distribution is uniform, the storage overhead at Indexing Cloud is negligible, occupying only ∼120KB of extra storage in Uni-Smart*. This is because when dataset is uniform, the number of generated buckets is minimal, and hence leading to the smallest number of b-codes. In contrast, the space requirement at Indexing Cloud increases as the

non-uniform datasets are considered. Particularly, PARADOT allocates 6.2MB (Zipf-Smart) and 14MB (Sensor) at Indexing Cloud, experiencing an overhead of ∼2.3× and ∼0.03×, respectively. Interestingly, the storage overhead for 25%-Sensor and Sensor datasets is the same at this component, about 0.03×.

On the other hand, at Storage Cloud, the space overhead is at least ∼1.56× (25%-Sensor) and at most ∼3.7× (Zipf-Smart). It is worth noting that the space overhead of Indexing Cloud (∼4.44×) is slightly larger than that of Storage Cloud (∼3.7×) in case of Zipf-Smart dataset. This is not because the encrypted b-codes take too much space, but the record size of this dataset is too small, particularly, the size of a 144K-record dataset is 2.7MB.

Comparison with PINED-RQ. As can be seen from Figure 6.10, PINED-RQ

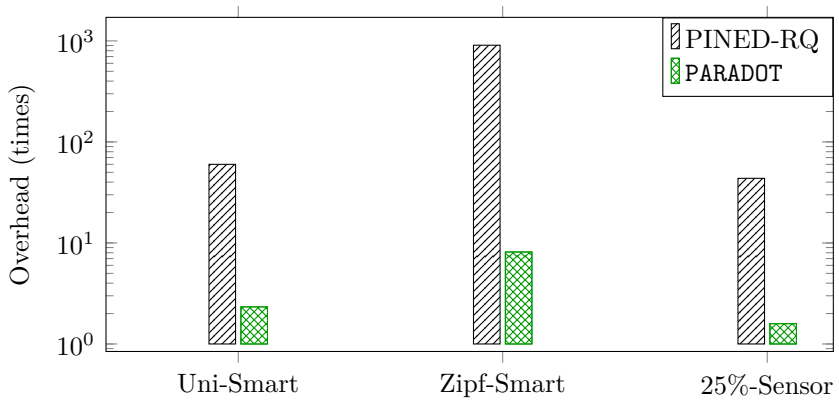


Figure 6.10 – Comparison of storage overhead

incurs much more space overhead as compared to PARADOT. Indeed, the storage overhead of PINED-RQ is at least ∼25.75× (Uni-Smart) and at most ∼111× (Zipf-Smart) higher than that of PARADOT.

Comparison with PBtree [65]. To compare PARADOT with this index, we use two non-uniform variants of Smart dataset, whose size is 0.5M and 5M records. Note that, we use the non-uniform datasets for fair comparison, that is, PARADOT experiences the highest overhead in such setting. Experimental results present that the index sizes, including the size of encrypted (r)b-codes), in PARADOT are 42MB and 406MB, respectively. Meanwhile, with the same number of data items for each dataset, PBtree requires in turn 1.598GB and 18.494GB [66]. In addition, an improved version of this

index, IBtree [64], also incurs the same space complexity, $O(n \log n)$ (with compression), where n is the number of data items.

d) Communication costs

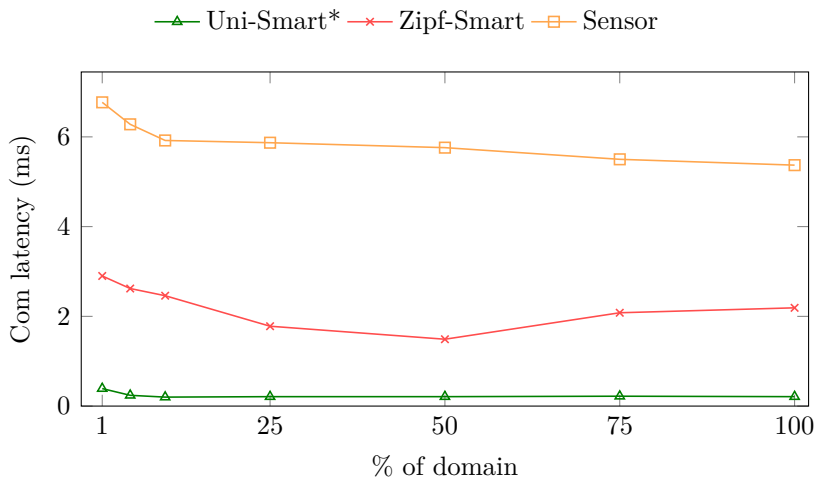


Figure 6.11 – Communication latency between Indexing Cloud and Storage Cloud in PARADOT

Since PARADOT relies on the two-cloud providers model, we evaluate communication latency between Indexing Cloud and Storage Cloud. Such latency results from the sending of encrypted sums from Indexing Cloud to Storage Cloud for every query processing. The transferring of these encrypted sums basically just leads to small extra communication overhead. As shown in Figure 6.11, the communication latency is relatively modest, being under 7ms, in all these experiments. Such delay primarily relies on the size of the b-code table CT. As data distribution is uniform like Uni-Smart, the CT is small, hence resulting in short latency, under 0.5ms. Nevertheless, when Zipf-smart is used, PARADOT experiences a higher delay, fluctuating between 1.5ms and 2.9ms. The largest delay is witnessed in Sensor, ranging from 5.5ms to 6.8ms. It is worth nothing that our experiments are carried out on the same private cloud. It is expected there some additional delays as the two clouds reside at different regions/countries. However, such evaluation is beyond the scope of this dissertation.

e) Updates

We focus on two cases for update operations, existing values (U-1) and new values (U-2). Two mains metrics, query latency and communication overhead, are also taken into account in these experiments.

U-1: Existing values. We first consider the case where all updates contain data of the existing values. This context is common in a wide range of real-world applications, especially in IoT. For example, Internet-connected devices (e.g., sensors) are configured to periodically send the sensing data to a central server. To mimic such real world situation, for Smart dataset, we sample 100 updates, and each consists of 164,160 records. This represents the number of records are generated every day of 114 users with an interval of 1 minute (a record/user/minute). This also means that a user contributes 1440 records to each update. Regarding Sensor dataset, we also sample 100 updates and each contains 1% of the original dataset that is equivalent to 3,744 records. That is, if a user has 100 records, every update contains one record of that user.

Query latency. The results in Figure 6.12 give that the total time increases when

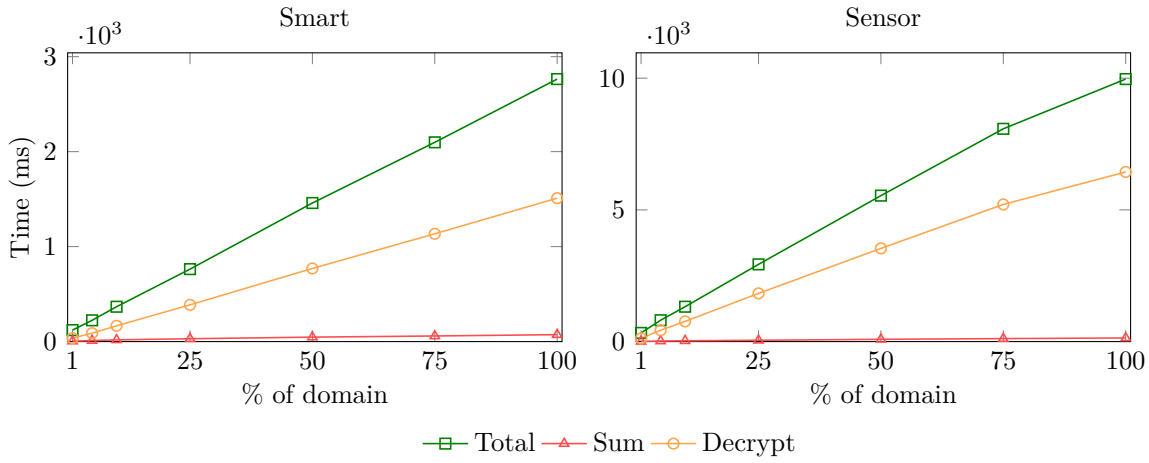


Figure 6.12 – Query latency of PARADOT

larger range sizes are used. More importantly, PARADOT often spends much time decrypting the returned results at the client side. For instance, in Smart dataset, the decryption occupies at least 29.75% (1%-range size) and at most 54.63% (100%-range size) of the total time. Likewise, the decryption time reaches up to 64.61% (100%-range size) of the

total latency when the Sensor dataset is taken into account. By contrast, PARADOT spends very short time performing the addition of encrypted (r)b-codes at Indexing Cloud. As depicted in Figure 6.12, even though the largest range size (100%) is used, such task takes at most ∼73ms (Smart) and ∼138ms (Sensor), accounting for ∼2.64% and ∼1.38% of the total time, respectively. Thanks to the parallelism at Indexing Cloud, as the range size grows, the contribution of the summing time gradually declines from ∼4.13% to ∼2.64% in Smart and from 2.43% to 1.38% in Sensor. This somehow demonstrates the use of encrypted b-codes at Indexing Cloud does not much degrade the query performance in general.

Communication overhead. As shown in Figure 6.13, the size of the transferred

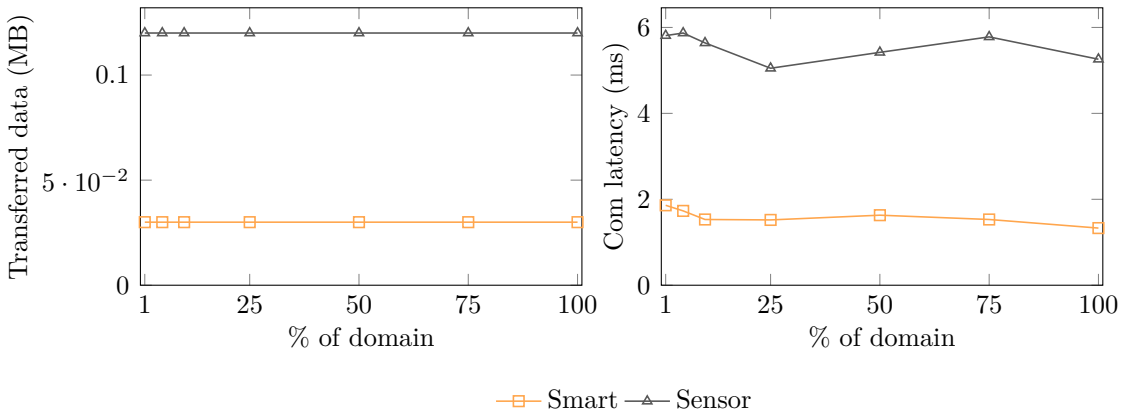


Figure 6.13 – Communication overhead of PARADOT

data between the two clouds is constantly ∼0.03MB in Smart dataset and ∼0.12MB in Sensor dataset for all range sizes. The difference in size between the two datasets comes from the fact that the number of encrypted sums in Sensor is larger than that in Smart. More precisely, since Sensor dataset is non-uniform, there are more buckets created as compared to Smart dataset, and hence more columns of the b-code table are formed in Sensor. As a result, the number of encrypted sums in Sensor is larger than that of Smart. Due to the small size of the transferred data, the communication latency is relatively low in all experiments, at most 1.86ms in Smart and 5.87ms in Sensor.

U-2: New values. In these experiments, we reuse the sampled Smart dataset from U-1. However, we uniformly partition the dataset into two smaller equal-size

datasets, denoted Smart-1 and Smart-2, so that each contains data of 50% distinct values of the indexed attribute, that is, one of them contains data of 57 users. Every dataset is then equally divided into 100 parts, each has the size of 82,080 records. This implies the number of records are generated every day of 57 users with an interval of 1 minute (a record/user/minute). Smart-1 is used as an initial publication while Smart-2 is used to generate updates to this publication. More precisely, one part of Smart-1 is initially sent to the clouds, and then one part of Smart-2 is published to the clouds as an update of the previous one. Obviously, there are two secure indexes are created at Indexing Cloud. The rest of these datasets are sequentially sent to the clouds as further updates.

Query latency. The results in Figure 6.14 show that there is a similar pattern as

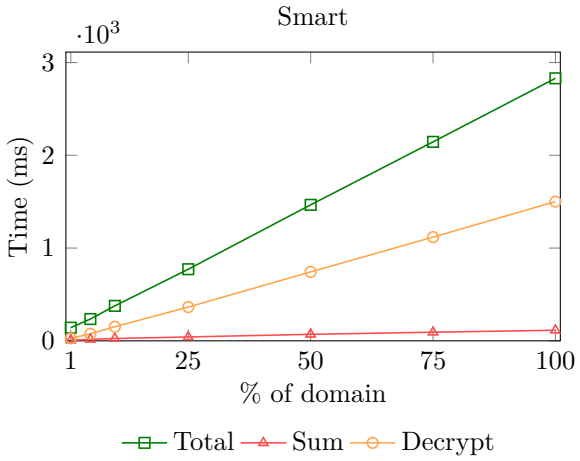


Figure 6.14 – Query latency of PARADOT

compared to U-1. This means that when a larger range is used, the query latency increases accordingly. Despite the fact that the storage overhead at Indexing Cloud in U-2 is nearly double that in U-1, 16MB against 8.3MB, the latency in U-2 is slightly longer than that in U-1. For example, when the largest query range is used, U-1 and U-2 in turn exhibit a latency of 2.7s and 2.8s.

Communication overhead. As shown in Figure 6.15, the experimental results show that the size of the transferred data is permanently ∼0.06MB and the communication latency fluctuates between 2.6ms and 3.0ms.

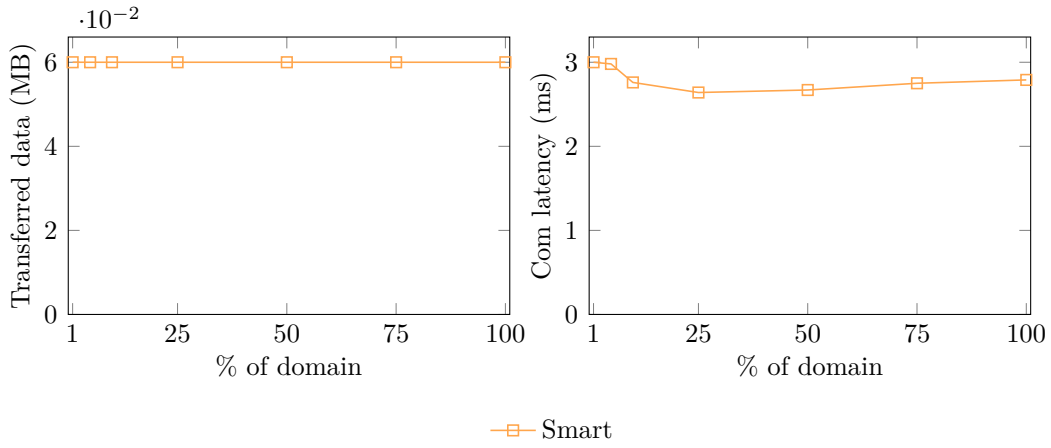


Figure 6.15 – Communication overhead of PARADOT

f) Scalability of using bitmaps

We now turn our attention to the scalability of using encrypted bitmaps in PARADOT. To achieve it, we seek to use varying size for the b-code table. In particular, we create several small updates from Smart dataset, ranging from 1K to 10K updates. Each update is uniform and has small number of records (e.g., 144 records). This means that each update results in one column in the b-code table, hence the number of columns of that table varies between 1K and 10K. Then, we measure various metrics that are primarily impacted by the increase in size of the b-code table. Additionally, we present significant performance gains that come from the parallelism at Indexing Cloud.

Performance gains of parallelism. Figure 6.16 gives the considerable gains of using parallelism for improving range query processing in general, and particularly the summing operation on encrypted (r)b-codes at Indexing Cloud. The parallel summing operation is from one to two orders of magnitudes faster than the non-parallel version. With the smallest range size, the summing time of the parallel Indexing Cloud is at least $\sim$89.7$\times$ and at most $\sim$149.6$\times$ shorter than that of the non-parallel one. Even if the largest range size is considered, the improvement of the parallel version remains still very high, at least 10.78$\times$ among all experiments.

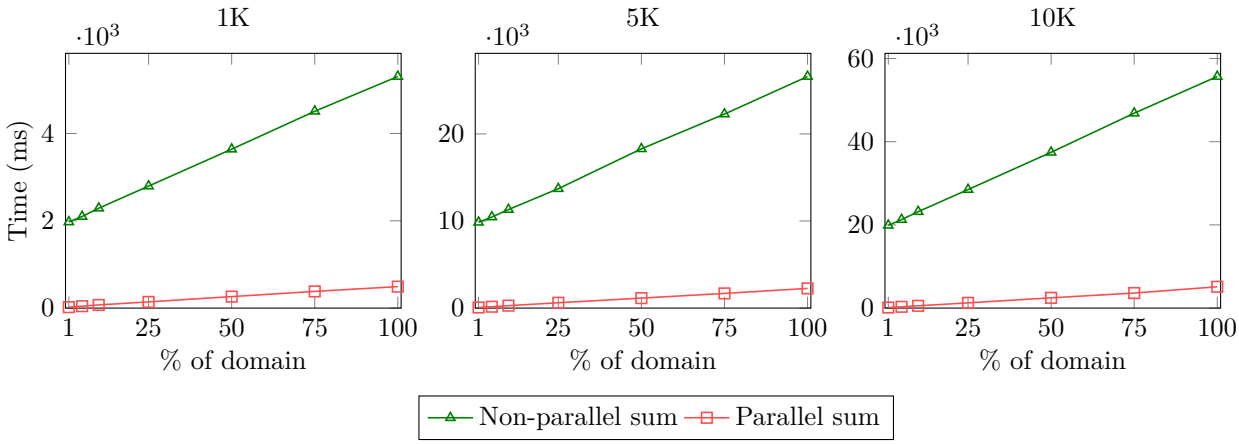


Figure 6.16 – Parallelism on summing operation at Indexing Cloud

Latency overhead of the b-code table. When a query is processed at the two clouds,

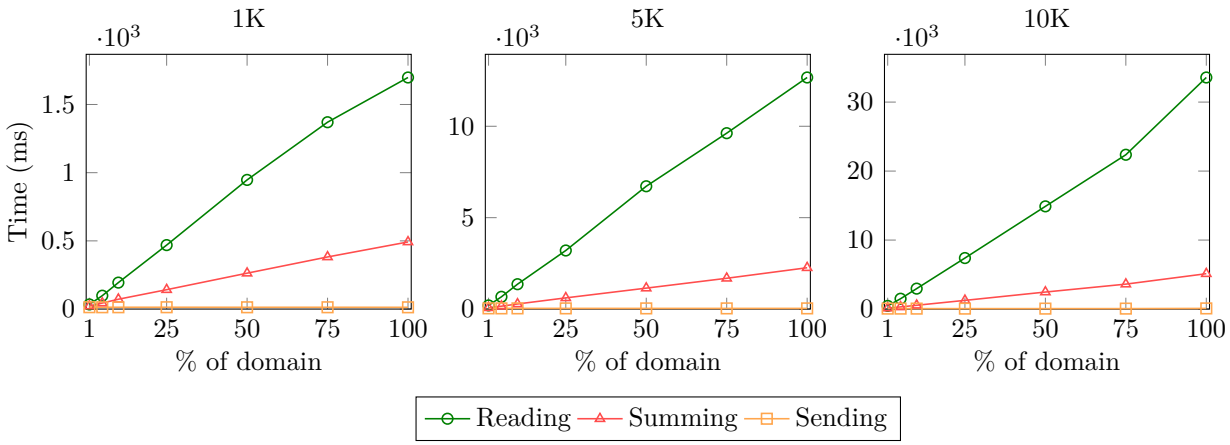


Figure 6.17 – Scalability

the latency of the b-code table includes reading relevant rows from LevelDB, summing these rows, and sending the summing row from Indexing Cloud to Storage Cloud. The results in Figure 6.17 show that when the size of the table grows, the time is needed for the three operations gradually rises accordingly. However, the sending time is negligible as compared to the summing and reading time. With the largest b-code table, Indexing Cloud takes at most ∼95ms to send the summing row to Storage Cloud. In contrast, Indexing Cloud spends significant time reading the table from disks. For example, for the 1K-column table, the reading time rises from 34ms to 1698ms, being at most 131.8× and 3.6× higher than that of the sending and summing time, respectively. The longest

delay of the reading operation is experienced as the 10K-column table and the largest range are taken into account, roughly ∼33s. On the other hand, thanks to the parallelism at Indexing Cloud, the summing time is practically acceptable. Even though the worst case is considered (10K-column table, 100%-range size), Indexing Cloud only spends approximately 5s on the summing operation.

Storage overhead of the b-code table. With a fixed number of indexed attribute values, the storage overhead of the b-code table is linear to its size. In particular, with the number of columns grows from 1K to 10K, the size of the b-code table on disks grows from 90MB to 828MB, respectively. It is worth noting that such overhead is partially caused by the use of additional keys for each b-code as they are stored in LevelDB.

6.6 Conclusion

In this chapter, we proposed `PARADOT`, a scalable scheme for one-dimensional range query processing over encrypted data stored at clouds. It can achieve all scalability requirements, namely fast computation, lightweight updates, and practical storage overhead, while the privacy is highly protected. To achieve it, we first designed a partitioning algorithm that enables to divide a dataset into equal-size buckets. We then proposed an encoding method based on equal-size bitmaps. An existing secure index is built on these bitmaps for fast query processing. Meanwhile, equal-size buckets and equal-size bitmaps enable high security in `PARADOT`. To support unlimited lightweight updates, we take a decoupling storage model into account. With such an approach, while our solution is able to support an unlimited numbers of updates, the storage overhead is practically acceptable. We finally show the practicality and scalability of `PARADOT` through extensive experiments performed on real datasets with various metrics.

Chapter 7

Conclusion and Future Works

7.1 Summary of contributions

In this dissertation, we focused on three main problems of the state-of-the-art solutions, namely bottlenecks in case of high rate of incoming data, modest ingestion throughput, and scalability in terms of storage and updates.

Regarding the issue of bottlenecks, rather than publishing datasets in large batches, we aimed to immediately send incoming data to the cloud one by one. To this purpose, we integrated a notion of index template into one of the most efficient solutions, particularly PINED-RQ index. To ensure privacy protection, we also designed a noise management method as well as a complementary data structure (e.g., matching table). Our solution, `PINED-RQ++` thus allows to smoothly ingest incoming data while still highly keeping the confidentiality of outsourced data. Nonetheless, with only these elements, the problem of bottlenecks might not be addressed since the collector endures heavy workflows. We thus proposed the parallel `PINED-RQ++` that mainly enhances the ingestion throughput of the system in general. Moreover, we gave thorough privacy analyses with regards to the two models, namely single cloud and two non-colluding clouds. The former ensures the privacy protection as strong as PINED-RQ [94] in case of offline attackers while the latter guarantees that `PINED-RQ++` has the same security level as compared to PINED-RQ. To evaluate our solutions, we extensively carried out experiments of both `PINED-RQ++` and parallel `PINED-RQ++` on real datasets with various scenarios. The experimental results showed that `PINED-RQ++` outperforms PINED-RQ in two main metrics, such as network traffic and publishing time, that mainly influence the congestion in the system. For exam-

ple, the publishing time of the NASA dataset (∼0.5M records) is decreased up to ∼35x while the maximum data rate experiences a reduction of up to ∼2.7x. Furthermore, parallel PINED-RQ++ also experiences a significant improvement in ingestion throughput, e.g., with the setting of 12-computing node cluster, the enhancement of parallel PINED-RQ++ is approximately 5.4× and 10× in Gowalla and USPS, respectively. Such gains make our solutions practical for a wide range of real-life applications.

Considering the problem of modest ingestion throughput, although PINED-RQ++ can avoid bottlenecks when data arrives rapidly, like many existing solutions, its ingestion throughput is still far from the needs of several real-world applications. To address this problem, we re-designed the architecture of parallel PINED-RQ++ such that it is fully distributed processing in coming data at the collector. Moreover, we introduced a data representation and an asynchronous publishing mechanism to avoid throughput degradation as much as possible. All of them together allows FRESQUE to achieve high ingestion throughput. Besides, we adapted FRESQUE to a stronger type of attackers as well as improving its practicality. To this end, we introduced and integrated a new component, e.g., randomer, into FRESQUE. We implemented and evaluated FRESQUE on real-world datasets. FRESQUE gave the lowest ingestion degradation and the highest throughput among the three prototypes, namely FRESQUE, parallel PINED-RQ++, and PINED-RQ++. FRESQUE is able to achieve over 165K insertions in a second when 12-computing node cluster is used, outperforming PINED-RQ++ and parallel PINED-RQ++ by ∼43× and ∼5.6×, respectively. We also discussed an application of FRESQUE and the possibility of using cutting-edge technologies to increase the scalability.

For the scalability dimension, we proposed a novel scheme for scalable and private range rang query processing, PARADOT. This scheme achieves all scalability requirements, namely efficient computation, practical storage overhead, and lightweight updates, while strong privacy protection is still ensured. To achieve these goals, we first designed a partitioning algorithm that enables to divide a dataset into equal-size buckets. We then proposed an encoding method based on equal-size bitmaps. PARADOT relies on existing secure indexes (e.g., PINED-RQ) for fast range queries. While the bitmaps privately keep links to the buckets, a secure index is built on these bitmaps. We then decoupled the secure index from buckets by using the two non-colluding cloud providers model. By doing it, PARADOT not only gave very good performance, but also efficiently supported numerous updates. More importantly, our solution incurs practical space complexity. For instance, as compared to PINED-RQ, our solution is ∼176× faster in terms of query response latency

and uses at most ∼119.55× less space requirement. Furthermore, with dataset sizes of 0.5M and 5M records, the index size of PARADOT is in turn about 42MB and 406MB while PBtree requires 1.598GB and 18.494GB [66], respectively.

7.2 Future Works

7.2.1 Privacy budget management

The current method of managing privacy budget of FRESQUE is straightforward and works well when at most one record of an individual comes every fixed-period (e.g., weekly). However, it may not be efficient in terms of privacy budget usage as multiple records of an individual comes at any time points. This drawback should be involved in our future work.

7.2.2 Optimizations on encrypted bitmaps

Although PARADOT achieves very good performance and high scalability, it was implemented for identity attributes due to the limitation of PINED-RQ index. Thus, a more general approach should be considered in the future. Moreover, physical organization of b-code tables in this study relies on LevelDB and needs an additional key for each cell of the tables, and thereby PARADOT encounters higher overheads as it is, namely data retrieving latency and storage. We plan to consider a more appropriate approach, i.e. physical pointers, to improve this situation.

We used bitmap index to privately keep links between buckets and secure indexes, however, the biggest weakness of the bitmap index is that its size grows linearly with the number of distinct values. Consequently, PARADOT would incur performance degradation in case of long-term storage. To address this, various strategies to control the bitmap index sizes and improve the query response time have been considered, such as compression, encoding and binning [111, 101]. Nevertheless, the major challenge is how to align these techniques with the query processing in PARADOT without incurring any privacy leak. This question opens potential future research.

7.2.3 Reducing query latency

Most of the efficient schemes can provide fast query processing. Nevertheless, when the continuous range queries are considered, they may not meet latency requirements. To deal with this situation, caching techniques [92, 60, 26] can be applied to reduce the query latency. Indeed, such techniques enable to reuse the results of the previous queries, hence shortening the query latency.

Similarly, `PARADOT` may take advantages from such idea for boosting the addition of encrypted bitmaps. This means that we can cache requested sums at Storage Cloud for further queries. This is expected to relieve the overhead of query processing and communication between Storage Cloud and Indexing Cloud. However, one of the challenges of this approach is how to protect privacy of cached data from the adversary.

7.2.4 Edge computing

The increased adoption of edge computing primarily results from the resource-constrained of IoT devices, e.g., limited computation and storage capabilities. By shifting data computing and storage to the network edge [97], traffic and computational pressure on the centralized cloud are significantly mitigated and response time of cloud services is also reduced. However, edges are often untrusted and the question is how to protect data stored at these edges. A straightforward application of existing solutions like `FRESQUE` or `PARADOT` may not work since they always require a trusted component in their architectures. Seeking a solution for this challenge should be included in our future works.

Appendices

The contributions of this thesis include the following publications:

1. **Van-Hoang Tran**, Tristan Allard, Laurent d'Orazio and Amr El Abbadi. Range Query Processing for Monitoring Applications over Untrustworthy Clouds. International Conference on Extending Database (EDBT), Lisbon, Portugal, 2019. (short paper)
2. **Van-Hoang Tran**, Tristan Allard, Laurent d'Orazio and Amr El Abbadi. FRESQUE: A Scalable Ingestion Framework for Secure Range Query Processing on Clouds. International Conference on Extending Database (EDBT), Nikosia, Cyprus, 2021
3. **Van-Hoang Tran**, Tristan Allard, Laurent d'Orazio and Amr El Abbadi. Paradot: Scalable Private Range Query Processing of Dynamic Outsourced Data (In submission)

List of Figures

List of Tables

Bibliography

[1] Rakesh Agrawal et al. « Order Preserving Encryption for Numeric Data ». In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: ACM, 2004, pp. 563–574.

[2] Mazhar Ali, Samee U. Khan, and Athanasios V. Vasilakos. « Security in cloud computing: Opportunities and challenges ». In: *Information Sciences* 305 (2015), pp. 357–383.

[3] *All citizens, foreigners living in Vietnam to provide health declarations*. `https://www.nationthailand.com/news/30383833`. Nov. 2020.

[4] Cloud Security Alliance. *Top Threats to Cloud Computing + Industry Insights*. `https://downloads.cloudsecurityalliance.org/assets/research/top-threats/treacherous-12-top-threats.pdf`.

[5] Amazon. *Amazon EC2*. `https://aws.amazon.com/ec2/`.

[6] Amazon. *AWS Elastic Beanstalk*. `https://aws.amazon.com/elasticbeanstalk/`.

[7] Amazon. *Global Infrastructure*. `https://aws.amazon.com/about-aws/global-infrastructure/`.

[8] A. Arasu et al. « Transaction processing on confidential data using cipherbase ». In: *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 435–446.

[9] Matt Bishop and Carrie Gates. « Defining the Insider Threat ». In: *Proceedings of the 4th Annual Workshop on Cyber Security and Information Intelligence Research: Developing Strategies to Meet the Cyber Security and Information Intelligence Challenges Ahead*. CSIIRW '08. Oak Ridge, Tennessee, USA: ACM, 2008.

[10] Burton H. Bloom. « Space/Time Trade-Offs in Hash Coding with Allowable Errors ». In: *Commun. ACM* 13.7 (July 1970), pp. 422–426.

[11] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. « Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions ». In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 578–595.

[12] Alexandra Boldyreva et al. « Order-Preserving Symmetric Encryption ». In: *Advances in Cryptology - EUROCRYPT 2009*. Ed. by Antoine Joux. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 224–241.

[13] Dan Boneh, David Mazières, and Raluca A. Popa. *Remote Oblivious Storage: Making Oblivious RAM Practical*. 2011.

[14] Dan Boneh and Brent Waters. « Conjunctive, Subset, and Range Queries on Encrypted Data ». In: *Proceedings of the 4th Conference on Theory of Cryptography*. TCC'07. Amsterdam, The Netherlands: Springer-Verlag, 2007, pp. 535–554.

[15] Thomas Brewster. *Anti-Virus Firm BitDefender Admits Breach, Hacker Claims Stolen Passwords Are Unencrypted*. `https://www.forbes.com/sites/thomasbrewster/2015/07/31/bitdefender-hacked/`.

[16] Brian Bulkowski. *Aerospike Performance on Intel Optane Persistent Memory*. `https://www.aerospike.com/blog/performance-on-intel-optane-persistent-memory/`. 2019.

[17] *Capital One Data Breach Compromises Data of Over 100 Million*. `https://www.nytimes.com/2019/07/29/business/capital-one-data-breach-hacked.html`. 2019.

[18] Daniele Catteddu. « Cloud Computing: Benefits, Risks and Recommendations for Information Security ». In: *Web Application Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 17–17.

[19] Yan-Cheng Chang and Michael Mitzenmacher. « Privacy Preserving Keyword Searches on Remote Encrypted Data ». In: *Proceedings of the Third International Conference on Applied Cryptography and Network Security*. ACNS'05. New York, NY: Springer-Verlag, 2005, pp. 442–455.

[20] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[21] Richard Chow et al. « Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control ». In: *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. CCSW '09. Chicago, Illinois, USA: ACM, 2009, pp. 85–90.

[22] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009.

[23] Microsoft Corporation. *Microsoft Azure*. `https://azure.microsoft.com/`.

[24] Microsoft Corporation. *Microsoft Office 365*. `https://www.office.com/`.

[25] Reza Curtmola et al. « Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions ». In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. CCS '06. Alexandria, Virginia, USA: ACM, 2006, pp. 79–88.

[26] Laurent d'Orazio et al. « Building Adaptable Cache Services ». In: *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*. MGC '05. Grenoble, France: ACM, 2005, pp. 1–6.

[27] Craig Dalton et al. « Insights From Flutracking: Thirteen Tips to Growing a Web-Based Participatory Surveillance System ». In: *JMIR Public Health Surveill* 3.3 (Aug. 2017), e48.

[28] Ernesto Damiani et al. « Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs ». In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. CCS '03. Washington D.C., USA: ACM, 2003, pp. 93–102.

[29] Tobias Gauster Daniel Koffler and Gregor Laaha. *lfstat: Calculation of Low Flow Statistics for Daily Stream Flow Data*. `https://rdrr.io/cran/lfstat/`. 2016.

[30] Asbury Park Press Data Universe. *US Postal Service Salaries*. `https://content-static.app.com/datauniverse/caspio/bundle/USPS.html`. 2019.

[31] Dawn Xiaoding Song, D. Wagner, and A. Perrig. « Practical techniques for searches on encrypted data ». In: *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*. 2000, pp. 44–55.

[32] Ioannis Demertzis et al. « Practical Private Range Search in Depth ». In: *ACM Trans. Database Syst.* 43.1 (Mar. 2018), 2:1–2:52.

[33] Dropbox. *Cloud File Sharing and Storage for your Business*. `https://www.dropbox.com/`.

[34] Cynthia Dwork. « Differential Privacy ». In: *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II*. ICALP'06. Venice, Italy: Springer-Verlag, 2006, pp. 1–12.

[35] Cynthia Dwork and Aaron Roth. « The Algorithmic Foundations of Differential Privacy ». In: *Found. Trends Theor. Comput. Sci.* 9.3-4 (2014), pp. 211–407.

[36] Cynthia Dwork et al. « Calibrating Noise to Sensitivity in Private Data Analysis ». In: *Proceedings of the Third Conference on Theory of Cryptography*. TCC'06. New York, NY: Springer-Verlag, 2006, pp. 265–284.

[37] Yousef Elmehdwi, Bharath K. Samanthula, and Wei Jiang. « Secure k-nearest neighbor query over encrypted data in outsourced environments ». In: *2014 IEEE 30th International Conference on Data Engineering*. 2014, pp. 664–675.

[38] Christoph Fehling et al. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.

[39] *Flu Near You*. `https://flunearyou.org/`. 2019.

[40] *FluTracking.net*. `http://info.flutracking.net/`. 2019.

[41] T. Ge and S. Zdonik. « Fast, Secure Encryption for Indexing in a Column-Oriented DBMS ». In: *2007 IEEE 23rd International Conference on Data Engineering*. 2007, pp. 676–685.

[42] Craig Gentry. « A fully homomorphic encryption scheme ». `crypto.stanford.edu/craig`. PhD thesis. Stanford University, 2009.

[43] Craig Gentry. « Computing Arbitrary Functions of Encrypted Data ». In: *Commun. ACM* 53.3 (Mar. 2010), pp. 97–105.

[44] Craig Gentry. « Fully Homomorphic Encryption Using Ideal Lattices ». In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. STOC '09. Bethesda, MD, USA: ACM, 2009, pp. 169–178.

[45] O. Goldreich, S. Micali, and A. Wigderson. « How to Play ANY Mental Game ». In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: ACM, 1987, pp. 218–229.

[46] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. New York, NY, USA: Cambridge University Press, 2004.

[47] Oded Goldreich and Rafail Ostrovsky. « Software Protection and Simulation on Oblivious RAMs ». In: *J. ACM* 43.3 (May 1996), pp. 431–473.

[48] *Google fires engineer for privacy breach*. `http://edition.cnn.com/2010/TECH/web/09/15/google.privacy.firing/`. 2019.

[49] Caroline Guerrisi et al. « Factors associated with influenza-like-illness: a crowdsourced cohort study from 2012/13 to 2017/18 ». In: *BMC Public Health* 19.1 (2019), p. 879.

[50] Hakan Hacigümüş et al. « Executing SQL over Encrypted Data in the Database-service-provider Model ». In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. SIGMOD '02. Madison, Wisconsin: ACM, 2002, pp. 216–227.

[51] Keiko Hashizume et al. « An analysis of security issues for cloud computing ». In: *Journal of Internet Services and Applications* 4.1 (2013), p. 5.

[52] Michael Hay et al. « Boosting the Accuracy of Differentially Private Histograms Through Consistency ». In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 1021–1032.

[53] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. « A Privacy-preserving Index for Range Queries ». In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 720–731.

[54] Bijit Hore et al. « Secure Multidimensional Range Queries over Outsourced Data ». In: *The VLDB Journal* 21.3 (June 2012), pp. 333–358.

[55] Shamim Hossain. *The "Internet of Things" and cloud computing*. `https://www.ibm.com/blogs/cloud-computing/2013/05/01/internet-of-things-cloud-computing/`. May 2013.

[56] IDG. *2020 Cloud Computing Study*. `https://www.idg.com/tools-for-marketers/2020-cloud-computing-study/`. Aug. 2020.

[57] *Insider threat examples: 7 insiders who breached security*. `https://www.csoonline.com/article/3263799/insider-threat-examples-7-insiders-who-breached-security.html`. 2019.

[58] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. « Inference Attack against Encrypted Range Queries on Outsourced Databases ». In: *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. CODASPY '14. San Antonio, Texas, USA: ACM, 2014, pp. 235–246.

[59] Hasan Kadhem, Toshiyuki Amagasa, and Hiroyuki Kitagawa. « A Secure and Efficient Order Preserving Encryption Scheme for Relational Databases ». In: *KMIS*. 2010.

[60] Arthur M. Keller and Julie Basu. « A predicate-based caching scheme for client-serverdatabase architectures ». In: *The VLDB Journal* 5.1 (1996), pp. 35–47.

[61] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014.

[62] *LevelDB*. `https://github.com/google/leveldb`. 2020.

[63] Chao Li et al. « A Data- and Workload-Aware Algorithm for Range Queries under Differential Privacy ». In: *Proc. VLDB Endow.* 7.5 (Jan. 2014), pp. 341–352.

[64] R. Li and A. X. Liu. « Adaptively Secure Conjunctive Query Processing over Encrypted Data for Cloud Computing ». In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 2017, pp. 697–708.

[65] R. Li et al. « Fast and Scalable Range Query Processing With Strong Privacy Protection for Cloud Computing ». In: *IEEE/ACM Transactions on Networking* 24.4 (Aug. 2016), pp. 2305–2318.

[66] Rui Li et al. « Fast Range Query Processing with Strong Privacy Protection for Cloud Computing ». In: *Proc. VLDB Endow.* 7.14 (Oct. 2014), pp. 1953–1964.

[67] LIMOS. *Galactica*. `https://galactica.isima.fr/index.html`. 2019.

[68] David Linthicum. *The cloud is the secret weapon in the Internet of things*. `https://www.infoworld.com/article/2608029/the-cloud-is-the-secret-weapon-in-the-internet-of-things.html`. 2014.

[69] A. Liu et al. « Efficient secure similarity computation on encrypted trajectory data ». In: *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 66–77.

[70] Google LLC. *Google APP Engine*. `https://cloud.google.com/compute/`.

[71] Google LLC. *Google Compute Engine*. `https://cloud.google.com/compute/`.

[72] Google LLC. *Google Suite*. https://gsuite.google.com/.

[73] Matt Martino. *Coronavirus isolation measures are reducing all flu-like diseases, not just COVID-19*. https://www.abc.net.au/news/2020-04-17/coronavirus-numbers-flu-tracking-data/12134082. Apr. 2020.

[74] Charalampos Mavroforakis et al. « Modular Order-Preserving Encryption, Revisited ». In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 763–777.

[75] Frank D. McSherry. « Privacy Integrated Queries: An Extensible Platform for Privacy-preserving Data Analysis ». In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 19–30.

[76] X. Meng, H. Zhu, and G. Kollios. « Top-k Query Processing on Encrypted Databases with Strong Security Guarantees ». In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. Apr. 2018, pp. 353–364.

[77] Ilya Mironov et al. « Computational Differential Privacy ». In: *Advances in Cryptology - CRYPTO 2009*. Ed. by Shai Halevi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 126–142.

[78] *NASA Log*. http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html. 1996.

[79] Muhammad Naveed, Seny Kamara, and Charles V. Wright. « Inference Attacks on Property-Preserving Encrypted Databases ». In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 644–655.

[80] E. O'Neil, P. O'Neil, and K. Wu. « Bitmap Index Design Choices and Their Performance Implications ». In: *11th International Database Engineering and Applications Symposium (IDEAS 2007)*. 2007, pp. 72–84.

[81] Patrick O'Neil and Dallan Quass. « Improved Query Performance with Variant Indexes ». In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. SIGMOD '97. Tucson, Arizona, USA: ACM, 1997, pp. 38–49.

[82] R. Ostrovsky. « Efficient Computation on Oblivious RAMs ». In: *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*. STOC '90. Baltimore, Maryland, USA: ACM, 1990, pp. 514–523.

[83] *Overleaf - A Web-based Collaborative LaTeX Editor*. `https://www.overleaf.com/`.

[84] Pascal Paillier. « Public-Key Cryptosystems Based on Composite Degree Residuosity Classes ». In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238.

[85] Antonis Papadimitriou et al. « Big Data Analytics over Encrypted Datasets with Seabed ». In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 587–602.

[86] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. « Arx: An Encrypted Database Using Semantically Secure Encryption ». In: *Proc. VLDB Endow.* 12.11 (July 2019), pp. 1664–1678.

[87] Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. « An Ideal-Security Protocol for Order-Preserving Encoding ». In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 463–477.

[88] Raluca Ada Popa et al. « CryptDB: Protecting Confidentiality with Encrypted Query Processing ». In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 85–100.

[89] Ben Popken. *Google sells the future, powered by your personal data*. `https://www.nbcnews.com/tech/tech-news/google-sells-future-powered-your-personal-data-n870501`. 2018.

[90] K. Popović and Ž. Hocenski. « Cloud computing security issues and challenges ». In: *The 33rd International Convention MIPRO*. 2010, pp. 344–349.

[91] Matthew Portnoy. *Virtualization Essentials*. 1st. USA: SYBEX Inc., 2012.

[92] Qun Ren and Margaret H. Dunham. *Semantic Caching and Query Processing*. 1998.

[93] Paul Rosenzweig Richard Falkenrath. *Op-ed: Encryption, not restriction, is the key to safe cloud computing*. https://www.nextgov.com/it-modernization/2012/10/op-ed-encryption-not-restriction-key-safe-cloud-computing/58608/. Oct. 2012.

[94] C. Sahin et al. « A Differentially Private Index for Range Query Processing in Clouds ». In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. Apr. 2018, pp. 857–868.

[95] C. Sahin et al. « TaoStore: Overcoming Asynchronicity in Oblivious Data Storage ». In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 198–217.

[96] Pierangela Samarati. « Data Security and Privacy in the Cloud ». In: *Information Security Practice and Experience*. Ed. by Xinyi Huang and Jianying Zhou. Cham: Springer International Publishing, 2014, pp. 28–41.

[97] W. Shi and S. Dustdar. « The Promise of Edge Computing ». In: *Computer* 49.5 (2016), pp. 78–81.

[98] Israel Spiegler and Rafi Maayan. « Storage and retrieval considerations of binary data bases ». In: *Information Processing & Management* 21.3 (1985), pp. 233–254.

[99] Emil Stefanov and Elaine Shi. « ObliviStore: High Performance Oblivious Cloud Storage ». In: *2013 IEEE Symposium on Security and Privacy* (2013), pp. 253–267.

[100] Emil Stefanov et al. « Path ORAM: An Extremely Simple Oblivious RAM Protocol ». In: *J. ACM* 65.4 (Apr. 2018).

[101] Kurt Stockinger and K. Wu. *Bitmap Indices for Data Warehouses*. 2006.

[102] Marianthi Theoharidou et al. « The insider threat to information systems and the effectiveness of ISO17799 ». In: *Computers & Security* 24.6 (2005), pp. 472–484.

[103] Bhanu P Tholeti. *Learn about hypervisors, system virtualization, and how it works in a cloud environment*. https://developer.ibm.com/articles/cl-hypervisorcompare/. Sept. 2011.

[104] Hoang Van Tran et al. « Range Query Processing for Monitoring Applications over Untrustworthy Clouds ». In: *EDBT'19*. 2019, pp. 666–669.

[105] Stephen Tu et al. « Processing Analytical Queries over Encrypted Data ». In: *Proc. VLDB Endow.* 6.5 (Mar. 2013), pp. 289–300.

[106] Stephen Tu et al. « Processing Analytical Queries over Encrypted Data ». In: *Proc. VLDB Endow.* 6.5 (Mar. 2013), pp. 289–300.

[107] UMassTraceRepository. *Smart* Data Set for Sustainability*. http://traces.cs.umass.edu/index.php/Smart/Smart.

[108] *Updates on Border Control Measures in Response to COVID-19 (Coronavirus Disease 2019)*. https://www.ica.gov.sg/covid-19. 2020.

[109] Y. Vaizman, K. Ellis, and G. Lanckriet. « Recognizing Detailed Human Context in the Wild from Smartphones and Smartwatches ». In: *IEEE Pervasive Computing* 16.4 (2017), pp. 62–74.

[110] Yonatan Vaizman et al. « ExtraSensory App: Data Collection In-the-Wild with Rich User Interface to Self-Report Behavior ». In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, pp. 1–12.

[111] Jianguo Wang et al. « An Experimental Study of Bitmap Compression vs. Inverted List Compression ». In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 993–1008.

[112] M. Wen et al. « PaRQ: A Privacy-Preserving Range Query Scheme Over Encrypted Metering Data for Smart Grid ». In: *IEEE Transactions on Emerging Topics in Computing* 1.1 (June 2013), pp. 178–191.

[113] Wikipedia. *Block cipher mode of operation*. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation. 2020.

[114] *World health organization: influenza (seasonal)*. https://www.who.int/en/news-room/fact-sheets/detail/influenza-(seasonal). 2019.

[115] K Wu et al. « FastBit: interactively searching massive data ». In: *Journal of Physics: Conference Series* 180 (July 2009), p. 012053.

[116] Zhiqiang Yang, Sheng Zhong, and Rebecca N. Wright. « Privacy-Preserving Queries on Encrypted Data ». In: *Proceedings of the 11th European Conference on Research in Computer Security*. ESORICS'06. Hamburg, Germany: Springer-Verlag, 2006, pp. 479–495.

[117] Andrew Chi-Chih Yao. « How to Generate and Exchange Secrets (Extended Abstract) ». In: *FOCS*. 1986.

[118] Yinqian Zhang et al. « Cross-VM Side Channels and Their Use to Extract Private Keys ». In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 305–316.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Traitement des Requêtes d'Intervalle sur des Nuages Non Fiables

Mot clés : Requêtes de gamme, données chiffrées, cloud computing

Résumé : Le nuage informatique est devenu de plus en plus un standard pour réduire les coûts et permettre l'élasticité. Alors que les fournisseurs de nuages élargissent leurs services, les préoccupations relatives à la sécurité des données externalisées empêchent une adoption généralisée des technologies des nuages. Pour y remédier, le chiffrement est généralement utilisé pour protéger les données confidentielles stockées et traitées sur des nuages non fiables. Le chiffrement des données externalisées diminue toutefois les fonctionnalités des applications parce que la prise en charge de certaines fonctions fondamentales sur les données chiffrées est encore limitée.

Cette thèse se concentre sur le problème de la prise en charge des requêtes d'intervalle sur des données chiffrées stockées dans les nuages. De nombreuses études ont été introduites dans ce domaine. Néanmoins, aucun des schémas précédents ne montre des performances satisfaisantes pour les systèmes modernes, qui exigent non seulement des réponses à faible latence, mais aussi une haute évolutivité. En particulier, la plupart des solutions existantes souffrent soit d'un traitement inefficace des requêtes d'intervalle, soit d'un manque de confidentialité. Même si certaines peuvent assurer à la fois une protection élevée de la vie privée et un traitement rapide, elles ne satisfont pas aux exigences d'évolutivité, à savoir un haut débit d'ingestion, une surcharge de stockage pratique et des mises à jour légères.

Pour surmonter ces limites, nous proposons des solutions évolutives sur le traitement des requêtes d'intervalle sécurisées tout en préservant l'efficacité et une sécurité forte. Nos contributions sont les suivantes : (1) Nous adaptons l'une des solutions de pointe au contexte de haut débit de données entrantes qui crée souvent des goulets d'étranglement. En d'autres termes, nous introduisons et intégrons la notion de modèle d'index dans l'une des solutions de pointe afin qu'elle puisse s'adapter au contexte cible. (2) Nous développons un cadre d'ingestion intensive dédié au traitement de requêtes d'intervalle sécurisée sur des données chiffrées. En particulier, nous reconcevons l'architecture de la première contribution pour la rendre entièrement distribuée. Une présentation des données et une méthode asynchrone sont ensuite introduites. Ensemble, elles augmentent significativement la capacité de réception du système. En outre, nous adaptons le cadre à un type d'adversaires plus forts (par exemple, les attaquants en ligne) et améliorons son aspect pratique. (3) Nous proposons un schéma pour le traitement des requêtes d'intervalle privées sur des ensembles de données externalisés. Ce schéma répond au besoin d'une solution évolutive en termes d'efficacité, de haute sécurité, de surcharge de stockage pratique et de nombreuses mises à jour, qui ne peuvent être pris en charge par les protocoles existants. À cette fin, nous développons notre solution basée sur des conteneurs de données de taille égale et des index sécurisés. Le premier permet de protéger la confidentialité des données contre l'adversaire, tandis que le second permet l'efficacité. Pour permettre des mises à jour légères, nous proposons de découpler les index sécurisés de leurs conteneurs en utilisant des bitmaps de taille égale.

Title: Range Query Processing over Untrustworthy Clouds

Keywords: Range queries, encrypted data, cloud computing

Abstract: Cloud computing has increasingly become a standard for saving costs and enabling elasticity. While cloud providers expand their services, concerns about the security of outsourced data hinder cloud technologies from a widespread adoption. To address it, encryption is usually used to protect confidential data stored and processed on untrustworthy clouds. Encrypting outsourced data however mitigates the functionalities of applications since supporting some fundamental functions on encrypted data is still limited.

This thesis focuses on the problem of supporting range queries over encrypted data stored on clouds. Many studies have been introduced in this line of work. Nevertheless, none of prior schemes exhibits satisfactory performances for modern systems, that require not only low-latency responses, but also high scalability. Particularly, most existing solutions suffer from either inefficient range query processing or privacy leaks. Even if some can achieve both strong privacy protection and fast processing, they do not satisfy scalability requirements, namely high ingestion throughput, practical storage overhead, and lightweight updates.

To overcome this limitation, we propose scalable solutions on secure range query processing while still preserving efficiency and strong security. Our contributions are: (1) We adapt one of the state-of-the-art solutions to the context of high rate of incoming data that often creates bottlenecks. In other words, we introduce and integrate the notion of index template into one of the state-of-the-art solutions so that it can cope with the target context. (2) We develop an intensive ingestion framework dedicated to secure range query processing on encrypted data. Particularly, we redesign the architecture of the first contribution to make it fully distributed. A data presentation and asynchronous method are then introduced. Together, they significantly increase the intake ability of the system. Besides, we adapt the framework to a stronger type of adversaries (e.g., online attackers) and enhance its practicality. (3) We propose a scalable scheme for private range query processing on outsourced datasets. This scheme addresses the need of a scalable solution in terms of efficiency, high security, practical storage overhead, and numerous updates, which can not be supported by existing protocols. To this purpose, we develop our solution relying on equal-size chunks (buckets) of data and secure indexes. The former helps to protect privacy of the underlying data from the adversary while the latter enables efficiency. To support lightweight updates, we propose to decouple secure indexes from their buckets by using use equal-size bitmaps.