Practical Private Range Search in Depth

IOANNIS DEMERTZIS, University of Maryland
STAVROS PAPADOPOULOS, TileDB Inc.
ODYSSEAS PAPAPETROU, EPFL
ANTONIOS DELIGIANNAKIS, Technical University of Crete
MINOS GAROFALAKIS, ATHENA Research Center & Technical University of Crete
CHARALAMPOS PAPAMANTHOU, University of Maryland

We consider a data *owner* that outsources its dataset to an *untrusted server*. The owner wishes to enable the server to answer *range* queries on a single attribute, without compromising the privacy of the data and the queries. There are several schemes on *"practical"* private range search (mainly in database venues) that attempt to strike a trade-off between efficiency and security. Nevertheless, these methods either lack provable security guarantees or permit unacceptable privacy leakages. In this article, we take an *interdisciplinary* approach, which combines the rigor of security formulations and proofs with efficient data management techniques. We construct a wide set of novel schemes with realistic security/performance trade-offs, adopting the notion of *Searchable Symmetric Encryption* (SSE), primarily proposed for keyword search. We reduce range search to *multi-keyword* search using *range-covering* techniques with tree-like indexes, and formalize the problem as *Range Searchable Symmetric Encryption* (RSSE). We demonstrate that, given *any* secure SSE scheme, the challenge boils down to (i) formulating leakages that arise from the index *structure* and (ii) minimizing *false positives* incurred by some schemes under heavy data *skew*. We also explain an important concept in the recent SSE bibliography, namely *locality*, and design generic and specialized ways to attribute locality to our RSSE schemes. Moreover, we are the first to devise secure schemes for answering *range aggregate* queries, such as range sums and range min/max. We analytically detail the superiority of our proposals over prior work and experimentally confirm their practicality.

CCS Concepts: • **Information systems → Data management systems**; • **Security and privacy → Symmetric cryptography and hash functions**; **Management and querying of encrypted data**;

Additional Key Words and Phrases: Private range search, searchable encryption

The work of Minos Garofalakis was partially supported by the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 732907 (MyHealthMyData). This article was also partially supported by the European Commission under ICT-FP7-LEADS-318809 (Large-Scale Elastic Architecture for Data-as-a-Service), ICT-FP7-QualiMaster-619525 (a configurable real-time data-processing infrastructure mastering autonomous quality adaptation), NSF Grants No. 1526950, No. 1514261, and No. 1652259, and a NetApp faculty fellowship.

Authors' addresses: I. Demertzis, University of Maryland, 3409 A.V. Williams Building, College Park, MD, 20740; email: yannis@umd.edu; S. Papadopoulos, TileDB Inc. Boston, 275 River St, Unit 3, Cambridge, MA 02139; email: stavros@tiledb.io; O. Papapetrou, EPFL, IC IINFCOM DIAS, Station 14, Lausanne, Switzerland; email: odysseas.papapetrou@epfl.ch; A. Deligiannakis, Technical University of Crete, University Campus—Kounoupidiana, Chania, Greece; email: adeli@softnet.tuc.gr; M. Garofalakis, ATHENA Research Center & Technical University of Crete, Information Management Systems Institute (IMSI), Artemidos 6 & Epidavrou, 15125 Marousi, Athens, Greece; email: minos@acm.org; C. Papamanthou, University of Maryland, 3409 A.V. Williams Building, College Park, MD, 20740; email: cpap@umd.edu.

https://doi.org/10.1145/3167971

ACM Reference format:
Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. 2018. Practical Private Range Search in Depth. *ACM Trans. Database Syst.* 43, 1, Article 2 (March 2018), 52 pages.
https://doi.org/10.1145/3167971

1 INTRODUCTION

We focus on a setting with two parties: a data *owner* and a *server*. The owner *outsources* its dataset to the server, and gives the latter the authority to answer *range* queries on a single attribute. The server is *untrusted*, and the goal is to protect the privacy of the dataset and the queries. The owner *encrypts* its data prior to sending to the server. The challenge lies in enabling the server to process the owner's queries directly on the encrypted data, while achieving performance and costs close to the non-private case. The benefits of data outsourcing and the importance of privacy have been stressed in numerous earlier works (e.g., References [18, 59, 62, 66]).

Prior Work. Privacy-preserving range queries can be solved with *optimal* security via powerful theoretical cryptographic tools, such as *Oblivious Random Access Machine* (ORAM) [32, 56] and *Fully Homomorphic Encryption* (FHE) [27, 28]. Nevertheless, despite their recent advances, both these tools are prohibitively costly for large database applications [63–65]. Motivated by this, there is a long line of work on private range search (especially in database venues) that attempts to strike a more desirable balance between security and practical efficiency [33, 35, 36, 59, 66].

All existing approaches either lack provable security guarantees or permit unacceptable leakages. For instance, References [33, 35, 36] employ *deterministic* encryption (DET) and bucketization techniques that map tuples with the same query attribute value to the same bucket. DET leaks the distribution of the data on the query attribute. Moreover, these schemes do not offer standard security definitions and proofs, which makes it hard to determine any other possible leakage. Another set of works utilizes *Order Preserving Encryption* (OPE), which has the property that the ciphertexts preserve the ordering of the plaintexts [6, 7, 46, 51, 58]. As such, traditional efficient indexes can be built directly on encrypted data, and queried in the same manner as for plaintexts. Nevertheless, OPE is also deterministic, inheriting the distribution leakage. In addition, it inevitably leaks the ordering of the data.

The work closest to ours is by Li et al. [50], which follows the notion of *Searchable Symmetric Encryption* (SSE) [12, 13, 16, 18, 41, 62]. SSE has been studied extensively for keyword queries. It relaxes the security of ORAM by leaking the *access patterns* of each query (i.e., which data it "touches"), as well as the *search patterns* (i.e., which queries are the same). However, nothing else is leaked (e.g., data distribution). The gain from allowing this leakage is efficiency, since SSE schemes typically build on fast inverted indexes and make use of lightweight cryptography, such as *Pseudorandom Function* (PRF). SSE also provides a rigorous framework for accurately defining *leakage* for any construction. Unfortunately, the scheme by Li et al. [50] relies on weak, obsolete SSE definitions (explained in detail in Section 2.1). Moreover, it unnecessarily introduces *false positives* and does not support updates. Most importantly, it does not define leakages that are introduced by the tree structure utilized in query execution. As we will show later, one of our baseline constructions is built on similar ideas to Reference [50], but offers substantially better security and performance, accurately defining leakage, avoiding false positives, and supporting updates.

Our Contributions. In this article, we revisit practical private range search, taking an *interdisciplinary* approach that combines the state-of-the-art definitional framework of SSE [18], with efficient data management methods. In particular, we reduce range search to *multi-keyword* search

Table 1. Summary of Our RSSE Schemes and Analytical Comparison to Our Closest Competitor

Scheme	Security	Query Size	Search Time	Storage	False Posit.
Li et al. [50]	0	$O(\log R)$	$\Omega(\log n \log R + r)$	$O(n \log n \log m)$	$O(r)$
Quadratic	7	$O(1)$	$O(r)$	$O(nm^2)$	–
Constant-BRC	1	$O(\log R)$	$O(R + r)$	$O(n)$	–
Constant-URC	2	$O(\log R)$	$O(R + r)$	$O(n)$	–
Logarithmic-BRC	3	$O(\log R)$	$O(\log R + r)$	$O(n \log m)$	–
Logarithmic-URC	4	$O(\log R)$	$O(\log R + r)$	$O(n \log m)$	–
Logarithmic-SRC	7	$O(1)$	$O(n)$	$O(n \log m)$	$O(n)$
Logarithmic-SRC-i$_1$	6	$O(1)$	$O(R + r)$	$O(n \log m)$	$O(R + r)$
Logarithmic-SRC-i$_2$	5	$O(1)$	$O(r)$	$O(m + n \log n)$	$O(r)$

n: dataset size, r: result size, m: domain size, R: query range size.
Note: Our schemes are named after the storage expansion factor and the range-covering technique. A higher value in the Security column means better security guarantees. We use the following scale for a coarse-grained analysis of the security levels; 0: Not provable secure solution, 1–2: Order information about the individual tuples is leaked in every query access, 3–4: Order information is leaked by analyzing and combining the previously observed leakages, 5–7: Information about the sizes of the queried results is leaked. Please see the Qualitative comparison of each scheme for further details. Finally, the construction time cost is the same as the storage cost.

using *range-covering* techniques with tree-like indexes, i.e., converting a range into sub-ranges, each representing an index node and receiving a keyword label. Contrary to off-the-shelf multi-keyword SSE schemes [13] that incur a prohibitive linear search time in the dataset size, we design efficient techniques based on single-keyword SSE protocols. We also show that we can use the (single-keyword) SSE security games to prove the security of a *Range Searchable Symmetric Encryption* (RSSE) scheme, by carefully defining the extra *structural leakage* stemming from the use of the tree index to convert a range into a set of keywords. This has the important benefit that *any* secure SSE scheme can be used as a black box to realize an RSSE scheme, which means that any future advances in the active area of SSE can be readily incorporated into an RSSE construction.

We emphasize that expressing a range with sub-ranges mapped to index nodes gives a lot of flexibility in designing RSSE schemes with variable efficiency and security guarantees. We also point out that the choice of the range-covering method affects the security guarantees and could lead to false positives, especially under heavy data *skew*. To capture the above, we devise a wide set of solutions, which revolve around trading storage overhead and (potentially) false positives for security. Our constructions with their performance and security characteristics are summarized in Table 1 and discussed in detail throughout the article. We also quantify the costs of Li et al. [50], which is clearly subsumed by our Logarithmic-BRC scheme.

Logarithmic-SRC-i$_1$ offers the best trade-off between security and efficiency among all our solutions. It employs a novel directed-acyclic graph (DAG) structure that resembles a tree, and entails an extra round of communication between the owner and the server ("i" in its name stands for *interactive*). This index even allows hiding the order of the results, and minimizes the false positives. Logarithmic-SRC-i$_2$ targets at (asymptotically) eliminating the false positives, but introduces some extra space overhead as compared to Logarithmic-SRC-i$_1$. It is noteworthy that two of our schemes, namely Constant-BRC and Constant-URC, rely on the notion of *Delegatable PRFs* [47] and are motivated by a brief discussion included in Reference [47]. Our contribution lies in formalizing the solutions and proving them secure.

In addition, contrary to existing *dynamic* SSE solutions (DSSE) [12, 40, 41, 62], we tackle updates in a manner closer to the one employed in large-scale database systems. In particular, the DSSE works attempt to devise dynamic indexes that handle updates with the minimum possible

leakage. We stress that databases like Vertica [48] perform the updates in *batches*, such that (i) each batch is treated as an independent instance of the dataset and (ii) multiple batches periodically get *consolidated* into a single dataset. This is to amortize the average update cost, and substitute numerous random disk accesses with a few linear scan operations. We formulate updates in our setting by effectively using multiple *static* RSSE instances that periodically get consolidated and re-encrypted.

Our RSSE schemes rely on the performance of the chosen underlying SSE scheme. A typical SSE scheme creates a secure version of an inverted index for efficient keyword search. For the sake of security, if a keyword is associated with multiple documents, these documents must appear in *random* locations in the server's storage. This impacts search performance, since the server may end up performing numerous random accesses (which are expensive particularly on a spinning disk) and PRF computations. More recent SSE literature addresses this issue of *locality*, and designs efficient constructions that place the keyword results in *contiguous* storage locations, at the expense of some extra storage cost for providing provable security (e.g., Reference [2]). We explain that utilizing such an SSE scheme as a black box can attribute locality to any of our RSSE schemes. However, our RSSE schemes inherit the storage expansion of the locality-aware SSE scheme. Motivated by the fact that Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i$_2$ already expand their storage requirements, and by observing the particular structure of their indices, we design a locality-aware variant, called *Logarithmic-SRC-i**, without asymptotically increasing the storage. We experimentally demonstrate that Logarithmic-SRC-i* offers orders of magnitude faster search compared to Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i$_2$.

Finally, we are the first to address *range aggregate queries* in the context of RSSE. For instance, a RSQ retrieves the sum of values on some attribute B of the tuples whose values on some other attribute A fall in a user-specified range. Another range aggregate query is the *range min*, which returns the minimum value instead of the sum of values in the above example. We construct two secure schemes, one for range sum based on *prefix sums* [4], and one for range min based on the *sparse table* technique from Reference [3]. We demonstrate how to support these queries even in the presence of updates. We also explain how these constructions can be easily modified to solve other range aggregate queries, such as range count, average, max, top-k, and bottom-k.

Our contributions are summarized as follows:

- We are the first to formalize private range search in the context of state-of-the-art SSE, effectively introducing the first concrete RSSE framework.
- We devise numerous RSSE schemes, identifying various trade-offs between efficiency and security.
- We tackle updates by adopting techniques from large-scale database systems, while formalizing the leakages.
- We explain how to attribute locality to all our RSSE schemes in a generic manner, at the cost of extra space. We also design a new locality-aware variant of Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i$_2$, called Logarithmic-SRC-i*, which does not sacrifice any space complexity as compared to its counterparts.
- We formalize and solve range aggregate queries. Specifically, we design secure schemes for range sum and range min, which can be further extended to capture more queries, such as range count, average, max, top-k, etc.
- We analytically detail the superiority of our constructions over prior work and experimentally confirm their practicality.

Our proposed schemes focus on the foundational case of one-dimensional private range search. However, we explain that they can be easily tailored to the more general multi-dimensional case

(where one wishes to provide range predicates on multiple attributes). Although it is not in the scope of this work to design a specialized efficient multi-dimensional solution, we hope that our approach can pave the way for a complete study of this important problem.

Our article is self-contained, in the sense that it does not require any particular knowledge on security or data management. We provide both the rigorous definitions and the intuition behind all the formalism. The remainder of the article is organized as follows: Section 2 provides the related work and useful preliminaries, and Section 3 defines the targeted problem. In Sections 4, 5, and 6 we present the quadratic, constant, and logarithmic schemes. Section 7 attributes locality to our schemes. We discuss generic attacks on range queries and propose ways to tackle them in Section 8. Section 9 discusses handling of updates, Section 10 solves range aggregate queries, and Section 11 provides the experimental evaluation. Finally, Section 12 provides concluding remarks and discusses future extensions to the case of multi-dimensional range queries.

2 BACKGROUND

Section 2.1 surveys the related work, whereas Section 2.2 includes necessary preliminary information.

2.1 Related Work

General privacy-preserving queries can be solved with optimal security guarantees using powerful cryptographic protocols, such as FHE [27, 28] and ORAM [32, 56]. FHE enables the execution of *any* function directly on the ciphertexts, without revealing anything about the result. Unfortunately, despite the recent advances, FHE is still impractical due to the prohibitive ciphertext size and computational time. ORAM enables access to an encrypted memory space, without disclosing which memory location is accessed, thus hiding both the data and the *access patterns* of the queries. Even the most efficient ORAM schemes [63–65, 68] suffer from high bandwidth overhead and client storage cost, and require multiple rounds of communication. Kellaris et al. [45] proposed the first approach, based on ORAM and differential privacy, to support private range search, such that the tuple order information is protected from recently proposed attacks. Our solutions are more efficient, as they avoid the poly-logarithmic overhead induced by ORAM. Section 8 explains how we can apply the findings of Reference [45] to our schemes to protect them from the new attacks, without using ORAM protocols. Furthermore, a wide set of techniques attempts to mitigate the above issues, trading security for efficiency. Below we focus mainly on those targeting range queries.

One class of techniques relies on DET that maps two equal plaintexts to the same ciphertext [33, 35, 36] (its counterpart being *probabilistic encryption*), enabling *equality* queries on encrypted data. Hacigümüş et al. [33] were the first to introduce the problem of private database search, by proposing a bucketization-based data representation for efficiently searching on encrypted data. References [33, 35, 36] perform *bucketization* (accompanied by indexing) of data by encrypting based on the query attribute, and reduce range search to a set of equality queries that retrieve matching buckets. Although these schemes are quite efficient, they inherit the drawback of DET, which discloses the distribution of the data (since the bucketization essentially reveals a histogram of the data on the query attribute). Moreover, they do not offer rigorous security definitions and proofs. In the database community, another class of methods, OPE, was introduced by Agrawal et al. [1]. OPE has been extensively studied by both the database and the crypto community [6, 7, 42, 46, 51, 58][1]. OPE schemes have the property that the ciphertexts preserve the

[1]The paper of Karras et al. [42] belongs to the family of works related to OPE schemes, but lacks formal provable security guarantees; the recent work of Horst et al. [37] provides attacks against the schemes proposed by Karras, denoting the importance of rigorous security analysis of newly proposed schemes.

order of the plaintexts. Therefore, efficient traditional indexes can be built on the ciphertexts, in the same manner as on plaintexts. OPE is deterministic and, thus, inherits the data distribution leakage of DET. In addition, it also leaks the *order* of the data and, hence, offers even weaker security than DET. A related primitive, called *Order Revealing Encryption* [8, 17, 49], achieves slightly better security guarantees than OPE at the cost of some performance loss. However, it still reveals the order of the plaintexts. A recent work of Naveed et al. [54] shows that any database encrypted using DET and OPE is not reliable, as it is rendered vulnerable to severe attacks. This allows an attacker to *decrypt the actual encrypted records*. However, DET and OPE can be used in the particular case of encrypting unique attributes, and attributes that cannot be ordered.

SSE [2, 12–16, 18, 21, 23, 29, 40, 41, 55, 61, 62, 67] has been initially proposed in the context of *keyword search*, having as main goals to (i) introduce rigorous security definitions and (ii) enable the design of schemes that avoid the leakages of DET, while retaining high efficiency. There is currently no off-the-shelf SSE scheme that supports range queries. As we shall see, in our work (as well as in Reference [50] described below), we effectively reduce range search into *multi-keyword* search. The only SSE schemes that seem applicable to multi-keyword search is that of References [13, 39], which target *Boolean queries* and can express multi-keyword search as a *disjunction* of keywords. Unfortunately, both solutions suffer from increased leakages. In addition, disjunctions in Reference [13] are answered in *linear* time in the number of documents and Reference [39] improves the aforementioned search time but it stores an encrypted index that is not linear in the number of documents, ID pairs, which conflict both with our performance and security desiderata.

A closely related work to ours is the basic scheme of Li et al. [50]. Note that the authors also introduce two other schemes, which, however, share similarities with OPE and, hence, inherit its drawbacks explained above. In the sequel, any reference to Reference [50] implies the *basic scheme*. This scheme assumes a binary tree over the query attribute domain, and computes for every tuple d in the dataset D the $\log m$ dyadic ranges covering its attribute value, where m is the domain size. Let the dyadic ranges of item d be denoted by $DR(d)$. Li et al. create a binary tree as follows. The root initially corresponds to all data items, and stores a *Bloom filter* [5] over $\{DR(d) : d \in D\}$. The algorithm works recursively, starting from the root and working *top-down*. At each node, it randomly *permutes* and splits the data items in two sets, each corresponding to one child. Then, it stores a Bloom filter over the DR values for the data items corresponding to each node. Eventually, each leaf contains a Bloom filter indexing only $DR(d)$ for a single d. A range query is answered by splitting the range into its $O(\log R)$ minimal dyadic ranges, where R is the range size over the domain, and traversing the tree by checking whether the Bloom filter in each node "contains" some minimum dyadic range.

The costs of Reference [50] are included in Table 1. The scheme *fixes* the ratio of the *false positives* (inherent to Bloom filters) at each node. This results in $O(n \log n \log m)$ storage cost, and $O(r)$ false positives, where n is the dataset size and r the result size. Moreover, the query size is $O(\log R)$, whereas the search time is $\Omega(\log n \log R + r)$. Note that it is difficult (and out of our scope) to find a tight upper bound for the actual search time, due to the random perturbation of the items in the tree and the false positives, but Reference [50] points out that this could be $O(r \log n)$. Performance aside, the most severe drawback of Reference [50] is its security. First, it relies on the SSE definitions from Goh [29], which have been proven weak by References [15, 18]; at a very high level, Goh [29] implies that the privacy of the encrypted queries (trapdoors) is not protected, whereas Reference [18] protects both the data and the queries. Second, Reference [50] focuses on *non-adaptive adversaries*; from a practical point of view, this means that Reference [50] is secure only in applications that allow the users to ask all their queries *once in a batch*, and then they *shut down*. Contrary, it is not secure against adversaries that ask some queries first and, based on the

responses they get, then *adapt* their attacks and ask more targeted queries later. This considerably limits the applicability of this solution in realistic settings. Finally, Reference [50] does not support updates. We introduce Constant-BRC/URC (Section 5) and Logarithmic-BRC/URC (Section 6.1) that subsume Reference [50] in all aspects.

After our conference submission [22], References [26, 34] proposed new schemes for private range queries. Reference [34] has more leakage than our Constant-URC/BRC and it is also less efficient than all of our proposed schemes, since it has poly-logarithmic ($O(log^2 m)$) amortized search time (while the worst case is $O(n \log^2 m)$ even in the static case). Additionally, Fuhry et al. [26] proposed a new scheme for private range queries using secure hardware (Intel SGX). In addition to the fact that this work is incomparable with ours (since it relies on the existence of secure hardware), this article has more leakage than our schemes and is vulnerable to various side channel attacks. Faber et al. [24] also independently proposed schemes for range querying. Two of their schemes are practically the same as our Logarithmic-BRC/URC, whereas their three-range cover solution is similar to our Logarithmic-SRC scheme (SRC stands for single-range cover). However, they did not consider false positives under data skew, which is efficiently captured by our most advanced methods, Logarithmic-SRC-i_1 and Logarithmic-SRC-i_2.

There is a long line of work on creating *dynamic SSE* (DSSE) schemes to handle updates [12, 40, 41, 62]. The challenge in these works is to achieve a property called *forward privacy*; the server should not learn that a newly added item satisfies a query issued in the past. Most solutions focus on creating a dynamic secure index. In our article, we take an alternative approach that satisfies forward privacy, by utilizing only *static* SSE schemes and combining them with efficient bulk-loading techniques adopted from large-scale database systems (such as Vertica [48]).

Range queries have also been studied in a different setting where there are multiple parties contributing to the owner's dataset using her public key, and the owner issues its range queries on this collective dataset with her secret key [9, 60]. This setting is based on asymmetric cryptography and entails considerably higher computational costs than our schemes.

We are the first to propose private range aggregate queries. The closest work for aggregate queries is Reference [70], which, however, does not provide a solution for range aggregate queries and is based on the existence of secure hardware (Intel SGX).

In the conference version of this article [22], we formalized the RSSE problem and designed the Quadratic, Constant-BRC/URC, Logarithmic-BRC/URC/SRC, and Logarithmic-SRC-i_1 schemes. In this long version, we introduce two new schemes, namely Logarithmic-SRC-i_2 that can significantly reduce (asymptotically) the false positives, and Logarithmic-SRC-i^* that provides locality and, hence, much faster search at the server. In addition, we design new searchable schemes for range aggregate queries, such as range sum and min. Finally, we add proof sketches and experiments for all proposed schemes.

2.2 Preliminaries

We explain, in turn, two range-covering techniques with binary trees, the PRF and DPRF cryptographic tools, the required definitions and constructions we adopt from the SSE literature, and useful techniques for efficiently answering range aggregate queries.

Range-Covering Techniques. Let A be a domain. We construct a full binary tree over its values bottom-up. Given a range (i.e., a sequence of *contiguous* values) over A, a *range-covering* technique selects a set of nodes whose subtrees cover the given range entirely. We will describe two techniques, *best range cover* (BRC) and *uniform range cover* (URC). BRC essentially selects the *minimum* number of nodes that cover exactly the range (also called *minimum dyadic intervals*). For range

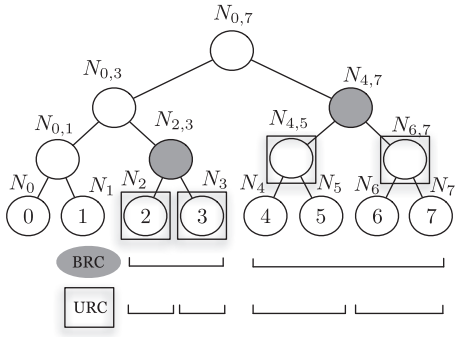


Fig. 1. Covering range [2, 7] with BRC and URC.

size R, there are $O(\log R)$ such nodes. In Figure 1, for $A = \{0, \ldots, 7\}$, BRC covers range $[2, 7]$ with nodes $N_{2,3}$ and $N_{4,7}$ (shown in gray).

Consider now range $[1, 6]$, which has the same size as $[2, 7]$. BRC covers $[1, 6]$ with nodes $N_1, N_{2,3}, N_{4,5}$ and N_6, i.e., with a *different number* of nodes at *different levels*. We shall see later that this leads to extra leakage, since the number of nodes covering a range may *imply* where an encrypted range query may or may not be. Motivated by this fact, Reference [47] introduces URC in the context of DPRFs (explained below). In particular, Reference [47] points out that there is always a *worst-case decomposition* of *any* range of a given size R into a *certain number* of nodes at *certain levels*. Interestingly, this decomposition retains the $O(\log R)$ complexity, *regardless of where* this range is placed over the domain. Briefly stated, URC starts with the set of nodes output by BRC, and keeps on "breaking" certain nodes into their two children, until there is at least one node for each level $0, \ldots, max$, where max is the highest level of nodes in the result. In the example of Figure 1, for range $[2, 7]$, URC initially invokes BRC and retrieves $N_{2,3}$ and $N_{4,7}$. These nodes are at levels 1 and 2, respectively, but there is no node at level 0. Subsequently, it breaks $N_{4,7}$ into $N_{4,5}$ and $N_{6,7}$, as well as $N_{2,3}$ into N_2 and N_3. Now $[2, 7]$ is represented by nodes $N_2, N_3, N_{4,5}, N_{6,7}$ (enclosed in boxes in Figure 1), i.e., by two nodes at level 0 and two nodes at level 1. Observe that $[1, 6]$ is also represented by the same number of nodes at respective levels. We refer the reader to Reference [47] for further details and for the formal analysis of URC.

PRFs and DPRFs. A *PRF family* $\mathcal{F}$ is a set of functions $\{f_k : A \to B \mid k \in \mathcal{K}\}$, where $A, B, \mathcal{F}, \mathcal{K}$ are indexed by a security parameter λ and such that $f_k(\cdot)$ is efficiently computable. The main property of a PRF is that an adversary that does not know the secret key k can distinguish $f_k \in \mathcal{F}$ from a truly random function only with negligible probability in λ, written as $\text{negl}(\lambda)$.

A *Delegatable PRF (DPRF)* [47] is an enhancement of a PRF with an extra property: the party that knows the secret key k of f_k can allow another party that does not possess k to derive DPRF values for a *subset* of the domain A. The benefit is performance: the secret holder generates and outsources a small set of intermediate values, which can be used by an untrusted third party to produce an exponential number of DPRF values. Similar to a PRF, the intermediate and final DPRF values appear to be random.

In Reference [47] the DPRF values are computed using the seminal GGM pseudorandom generator [31]. This is defined as $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$, i.e., on a λ-bit input x, $G(x)$ produces *two* λ-bit outputs $G_0(x)$ and $G_1(x)$ that appear to be random. Let $a_{\ell-1} \ldots a_0 = a \in A$ be some ℓ-bit domain value. Its DPRF value using secret key k is computed as $f_k(a_{\ell-1} \ldots a_0) = G_{a_0}(\ldots(G_{a_{\ell-1}}(k)))$, i.e., k serves as the *seed* to successive computations of G. For example, the binary representation of 6

is $(110)_2$. In Figure 1, observe that 6 is reached by a path starting from the root that chooses right, right and left. Assigning 0 to left and 1 to right, this path traversal uniquely identifies the binary expression of value 6. The DPRF of 6 is $f_k(6) = G_0(G_1(G_1(k)))$. The GGM values are organized into a binary tree, hereafter called GGM tree.

The purpose of this particular construction is to permit delegation. Let us focus on range [4, 7], completely covered by node $N_{4,7}$ in Figure 1. Observe that, given $G_1(k)$ and without possessing k, one can derive *all* DPRF values for 4-7; $G_1(k)$ is associated with node $N_{4,7}$, and all values corresponding to its descendants can be derived by applying G successively using $G_1(k)$ as the seed and choosing the G_0 (left) or G_1 (right) output based on the path.

When the input range is not covered completely with a single node, it is decomposed into a set of multiple nodes covering the range (following BRC or URC), and the appropriate GGM values corresponding to those nodes (paired with the node level) are provided. The receiver of these values can then easily derive all the DPRFs within the range. The GGM values are called *tokens* and produced by a function T that implements BRC or URC, whereas the derivation of the actual DPRF values (corresponding to the leaves) is performed by a function C. Both T and C are part of the specification of the DPRF function family. Formally, T is defined as a function that takes as input a secret key k and a predicate P and outputs a token. As mentioned above, T implements either the BRC or the URC policy. C is an algorithm that, given a token, outputs a PRF value for every input x that satisfies the predicate P. For more details, we refer the reader to the original DPRF paper [47].

SSE Definitions. Let D be a collection of *documents*, where a document can be any data item, even a tuple. Each document $d \in D$ has a unique ID, which is an *alias* that allows easy mapping to d. Every d is also associated with a unique identifier $d.id$ and a set of keywords from a dictionary Δ, each denoted as $d.w$. We represent by $id(w)$ the IDs of the documents that contain w and $|id(w)|$ the number of documents that contain w. We also define $n \triangleq \sum_{w \in \Delta} |id(w)|$ as the size of dataset D, i.e., the number of all $(d.id, d.w)$ pairs for all $d \in D$. SSE schemes focus on building an *encrypted index* I on the document IDs. For simplicity, we concentrate only on the IDs, since the actual documents are encrypted independently and stored at the server separately from I; once some ID is retrieved during search, the server can send the corresponding document to the owner, who decrypts in a final step that is *orthogonal* to the SSE instantiation.

An SSE *protocol* involves an *owner* and a *server* and consists of the following algorithms:

- $k \leftarrow \mathsf{Setup}(1^\lambda)$. A probabilistic algorithm run by the owner before commencing the system. It takes as input security parameter λ and outputs a secret key k.
- $I \leftarrow \mathsf{BuildIndex}(k, D)$. A probabilistic algorithm run by the owner prior to sending its data to the server. It takes as input the secret key k and the data collection D, and outputs an encrypted index I built on the document IDs. Index I is sent to the server, along with the actual encrypted documents.
- $t \leftarrow \mathsf{Trpdr}(k, w)$. A deterministic algorithm executed by the owner when issuing a query. It takes as input key k and keyword w, and outputs a token t.
- $X \leftarrow \mathsf{Search}(t, I)$. A deterministic algorithm run by the server to retrieve the IDs of the documents containing the query keyword. It takes as input a token t corresponding to the query keyword and the encrypted index I, and outputs a set X of document IDs.

In state-of-the-art SSE constructions [12, 13, 16, 18, 40, 41, 62], I is essentially an encrypted *inverted index*, which allows efficient retrieval of the document ID list corresponding to the query keyword. The token t constitutes auxiliary information that allows the server to *partially decrypt* only the index components that lead to the retrieval of the result IDs. However, once these index

$\mathbf{Real}_{\mathsf{SSE},\mathcal{A}}(k)$	$\mathbf{Ideal}_{\mathsf{SSE},\mathcal{A},\mathcal{S}}(k)$
$k \leftarrow \mathsf{Setup}(1^\lambda)$	
$(D, st_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^\lambda)$	$(D, st_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^\lambda)$
$I \leftarrow \mathsf{BuildIndex}(k, D)$	$(I, st_{\mathcal{S}}) \leftarrow \mathcal{S}_0(\mathcal{L}_1(D))$
$(w_1, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(st_{\mathcal{A}}, I)$	$(w_1, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(st_{\mathcal{A}}, I)$
$t_1 \leftarrow \mathsf{Trpdr}(k, w_1)$	$(t_1, st_{\mathcal{S}}) \leftarrow \mathcal{S}_1(st_{\mathcal{S}}, \mathcal{L}_2(D, w_1))$
for $2 \leq i \leq q$	for $2 \leq i \leq q$
$(w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}_i(st_{\mathcal{A}}, I, t_1..t_{i-1})$	$(w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}_i(st_{\mathcal{A}}, I, t_1..t_{i-1})$
$t_i \leftarrow \mathsf{Trpdr}(k, w_i)$	$(t_i, st_{\mathcal{S}}) \leftarrow \mathcal{S}_i(st_{\mathcal{S}}, \mathcal{L}_2(D, w_1..w_i))$
let $\mathbf{t} = (t_1..t_q)$	let $\mathbf{t} = (t_1..t_q)$
output $v = (I, \mathbf{t})$ and $st_{\mathcal{A}}$	output $v = (I, \mathbf{t})$ and $st_{\mathcal{A}}$

Fig. 2. SSE ideal-real security game.

portions are decrypted, they become permanently known to the server. In other words, SSE inherently introduces certain information *leakage*.

An ad hoc way of defining security would be to outline a set of adversarial attacks, and prove that the scheme is robust against these attacks. This is dangerous, as we cannot anticipate the types of attacks an adversary is able to launch. A *rigorous* way to define security is to *formulate* the leakage, and *prove* that the adversary learns nothing more than this leakage. Curtmola et al. [18] introduced a framework for achieving this, following the seminal *ideal-real paradigm* by Goldreich [30]. In particular, after formulating leakage, we define two *games*. The *real* is essentially the execution of the actual SSE protocol. The *ideal* is a *simulation* of the real, i.e., an attempt to "fake" the real game, knowing only the formulated leakage. Finally, we prove that an adversary can *distinguish* the output of the first from that of the second with only *negligible* probability. Intuitively, this means that the adversary indeed does not learn anything more than the leakage, otherwise he would be able to distinguish the real from the ideal execution with non-negligible probability.

We focus on *semi-honest*, *adaptive* adversaries. "Semi-honest" means that the adversary is curious to infer information during the execution of the protocol, but does not deviate from the protocol. "Adaptive" means that the adversary attempts to learn information even in between query executions, and may adaptively select the next query based on the previous ones. A non-adaptive adversary submits all queries before starting to learn information. Clearly, adaptive adversaries are more realistic in database applications where the queries are not presented all at once to a system.

For completeness and to facilitate presentation in Section 3, in Figure 2 we present the SSE ideal-real games for (semi-honest) adaptive adversaries, as introduced in Reference [19]. In $\mathbf{Real}_{\mathsf{SSE},\mathcal{A}}$, an adversary $\mathcal{A}$ interacts with the actual SSE protocol, *choosing* the initial document set and (adaptively) the keyword queries. The adversary gets access only to BuildIndex and Trpdr, since it does not know the secret key k. $st_{\mathcal{A}}$ is some *state* maintained by the adversary. The final *view* of $\mathcal{A}$ is the encrypted index I, and the set of generated tokens $\mathbf{t}$ and $st_{\mathcal{A}}$. Now observe the line correspondence between $\mathbf{Real}_{\mathsf{SSE},\mathcal{A}}$ and $\mathbf{Ideal}_{\mathsf{SSE},\mathcal{A},\mathcal{S}}$. In the latter, a *simulator* $\mathcal{S}$ (maintaining state $st_{\mathcal{S}}$) is enforced with "faking" BuildIndex and Trpdr for the same D and query keywords, *only using* leakage functions $\mathcal{L}_1$ and $\mathcal{L}_2$ (explained below). Security boils down to returning $(I, \mathbf{t}, st_{\mathcal{A}})$ that is distinguishable with negligible probability from the output by the real game. The challenge lies in properly using leakages $\mathcal{L}_1$ and $\mathcal{L}_2$ to create I and $\mathbf{t}$, such that (i) they "look" like those produced by real and (ii) the Search algorithm in ideal is *consistent*, i.e., it functions similarly to that in real.

Table 2. Comparison of the Most Representative SSE Schemes

Scheme	Storage	Locality	Read Efficiency
Locality Non-aware			
Curtmola et al. [18] (SSE-1)	$O(n)$	$O(r)$	$O(1)$
Curtmola et al. [18] (SSE-2)	$O(nm)$	$O(r)$	$O(1)$
Liesdonk et al. [67]	$O(nm)$	$O(r)$	$O(1)$
Kamara et al. [41]	$O(n)$	$O(r)$	$O(1)$
Kamara et al. [40]	$O(nm)$	$O(r \log n)$	$O(n \log n)$
Cash et al. [13]	$O(n)$	$O(r)$	$O(1)$
Cash et al. [12]	$O(n)$	$O(r)$	$O(1)$
Stefanov et al. [62]	$O(n)$	$O(r \log^3 n)$	$O(\log^3 n)$
Locality Aware			
Cash et al. [14]	$O(n \log n)$	$O(\log n)$	$O(1)$
Asharov et al. [2] (Approach #1)	$O(n)$	$O(1)$	$O(\log n \ \log \log n)$
Asharov et al. [2] (Approach #2)	$O(n)$	$O(1)$	$O(\log \log n \ \log^2 \log \log n)$*
Asharov et al. [2] (Approach #3)	$O(n \log n)$	$O(1)$	$O(1)$
Demertzis and Papamanthou [23]	$O(n \cdot s)$	$O(n^{1/s}/L)$	$O(L)$
Demertzis et al. [21]	$O(n)$	$O(1)$	$O(\log^{0.67} n)$
Lower Bound			
Cash and Tessaro [14]	$\omega(n)$	$O(1)$	$O(1)$

*It is based on the assumption that all keyword lists in the dataset have size less than $n^{1-1/\log \log n}$.
n: dataset size, m: number of keywords, r: result size.

Although our schemes are independent of the underlying SSE construction, as an example, we describe the leakage functions $\mathcal{L}_1, \mathcal{L}_2$ assuming the SSE scheme by Reference [13]. $\mathcal{L}_1$ is associated with what is leaked from the index alone, whereas $\mathcal{L}_2$ accounts for the leakage from the queries.

— $\mathcal{L}_1(D) = n$, where n is the size of D.
— $\mathcal{L}_2(D, W) = \langle \alpha(W), \sigma(W) \rangle$,
where W is a set of keywords, $\alpha(W) = (id(w))_{w \in W}$ is the *access patterns*, i.e., the document Is returned by each keyword query, and $\sigma(W)$ is the *search patterns*, i.e., for every pair $w_i, w_j \in W$ such that $i \neq j$, it indicates whether $w_i = w_j$ or $w_i \neq w_j$.

Cash et al. [11] introduced a family of attacks known as the known-document attacks. The effectiveness of these attacks was further improved by Zhang et al. [69]. These attacks require that the untrusted adversary has the ability to inject files to the encrypted index. Then, by exploiting the $\mathcal{L}_2$ leakage (access and search patterns), the adversary can learn all the client's query keywords. However, these attacks are inapplicable to our setting, where only the trusted owner/client has permission to update the encrypted index (insert/modify/delete tuples).

SSE Constructions. There exist numerous SSE constructions in the literature, which are essentially variations of a *secure inverted index* for keyword search. In Table 2, we list the most representative ones, grouping them into two categories, which we call *locality non-aware* and *locality aware* (the meaning of *locality* will become clear soon).

Figure 3 facilitates explaining the difference between the two categories. In the first category (locality non-aware), the owner's (encrypted) documents are placed in *random* locations in the server's storage (main memory or disk). This is due to the way the secure index is constructed

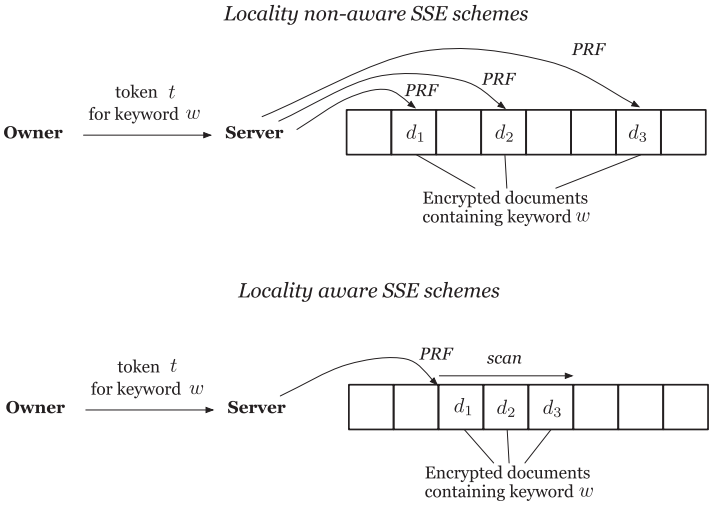


Fig. 3. Locality non-aware vs. locality-aware SSE schemes.

in each SSE scheme. The upper part of Figure 3 illustrates the way the documents that contain keyword w are randomly placed in the server's storage. Note that this random placement is important for providing provable security. Upon issuing query keyword w, the owner sends a token t to the server. The latter uses t to identify the random locations that store the results (through at least one PRF computation per result), and decrypt them before returning them to the owner. These approaches suffer from two drawbacks: (i) since the results of any keyword appear in random locations in the server's storage, retrieving them with random accesses may incur an excessive overhead (especially if the server uses a spinning disk for storage); (ii) if the query returns r documents, the server needs to perform at least r PRF computations, whose cost may become considerable for large r. The SSE methods in this category do not preserve any *locality* for the results of each keyword and, thus, we term them as *locality non-aware*.

The second category (locality aware) of SSE schemes targets at addressing the above-mentioned problem by providing result *locality*. Formally, locality is defined as the number of *look-ups* required to retrieve the result for any keyword. Optimal $O(1)$ locality is achieved when a single look-up is required to identify the storage location of the first result of keyword w, after which a single *scan* suffices for retrieving all the subsequent results. This is demonstrated in the lower part of Figure 3. Observe that (i) the results are all located in contiguous locations in the server's storage and (ii) the server performs a single PRF to locate the first result.

Unfortunately, optimal locality is very challenging to achieve while providing provable security. Specifically, to achieve optimal locality, a SSE scheme must either sacrifice storage space or read redundant information. The latter is formally defined as *read efficiency*, i.e., the expansion factor γ of the information to be retrieved when answering any keyword query (i.e., $\gamma \cdot r$ results are returned instead of the actual r). In other words, read efficiency quantifies the number of *false positives*. Optimal $O(1)$ read efficiency is achieved if $O(r)$ documents are returned when the result size is r. Cash and Tessaro [14] prove an interesting lower bound: a secure SSE scheme with optimal locality and read efficiency requires *superlinear* storage cost (this is shown at the bottom of Table 2).

Among the locality-aware SSE schemes, we are particularly interested in Approach #3 of Asharov et al. [2], which provides optimal locality and read efficiency at the cost of increased storage. We explain this technique using Figure 4. Recall that n is the size of the document collection

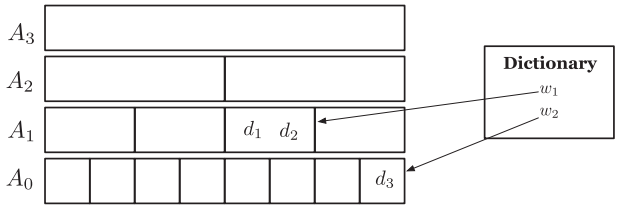


Fig. 4. Approach #3 of Reference [2].

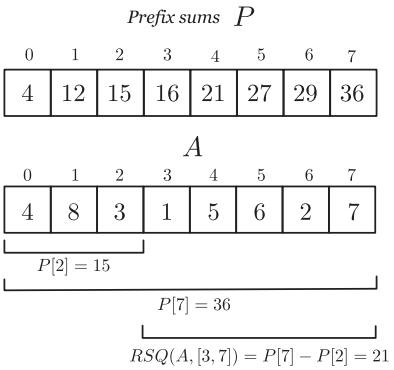


Fig. 5. RSQ example for range [3,7] on A.

(i.e., the number of all keywords in all documents, counting multiplicities across documents). The scheme creates $l = \log n + 1$ arrays $A_0, \ldots, A_l$, each of size n (the actual scheme uses hash tables, but here we use arrays for ease of demonstration). Array A_i consists of $n/2^i$ chunks (shown as rectangles in the figure), and each chunk stores a randomly chosen keyword result of size 2^i (appropriate padding is applied in case a result size is not a power of 2). An additional dictionary stores the chunk corresponding to each keyword. The total space required for this scheme is $O(n \log n)$.

In Figure 4, suppose that w_1 has results d_1, d_2, and w_2 has result d_3. The result of w_1 is randomly placed in a chunk of A_1, whereas the result of w_2 is randomly placed in a chunk of A_0. The owner's token for a keyword enables the server to look up the appropriate entry in the dictionary and retrieve the starting location of the chunk that contains the result. The server then scans the chunk to retrieve the query result. Since only a single look-up is involved and each chunk contains $O(r)$ items (where r is the actual result size), this scheme provides optimal locality and read efficiency, at the cost of extra storage.

Range-Sum Query (RSQ). Given an array A of n numbers (with indexing starting from 0) and a range $[i, j]$, the RSQ returns the sum of the elements in $A[i..j]$, i.e., $RSQ(A, [i, j]) = \sum_{k=i}^{j} A[k]$. A trivial way to answer this query is by accessing all elements in $A[i..j]$ and simply summing them up. The query time in this case is $O(r)$, where $r = j - i + 1$ is the range size.

A more efficient way to answer the RSQ query is with the use of *prefix sums* [4]. The prefix sums of an array A of n numbers is a n-element array P, where $P[i] = \sum_{j=0}^{i} A[j]$. In other words, element $P[i]$ stores the sum of element $A[i]$ and the elements that *precede* $A[i]$. In the example of Figure 5, $P[2] = A[0] + A[1] + A[2] = 15$. Observe that $P[i]$ can be calculated as $P[i] = P[i-1] + A[i]$. Therefore, P can be constructed in $O(n)$ time while consuming $O(n)$ space. Using the prefix sums,

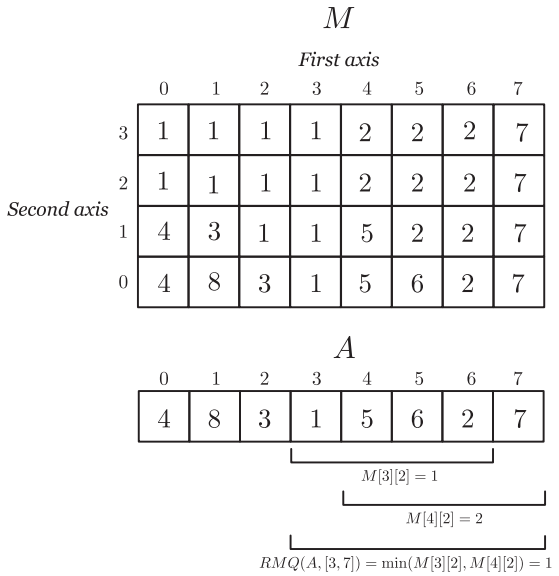


Fig. 6. RMQ example for range [3,7] on A.

RSQ can be answered in $O(1)$ time, simply as $RSQ(A, [i, j]) = P[j] - P[i-1]$ when $i \neq 0$, and $RSQ(A, [0, j]) = P[j]$. In Figure 5, $RSQ(A, [3, 7]) = P[7] - P[2] = 21$.

Range-Minimum Query (RMQ). Given an array A of comparable objects and a range $[i, j]$, the *Range-Minimum Query* (RMQ) returns the element from A with the minimum value within the subarray $A[i..j]$. For example, let $A = [4, 8, 3, 1, 5, 6, 2, 7]$ and assume that the array indexing starts from 0. Query $RMQ(A, [1, 5])$ returns $A[3] = 1$, which is the minimum element in subarray $A[1..5]$.

There exist several approaches to solving RMQ (e.g., see Reference [25] and the related work therein). We are particularly interested in the *sparse table* technique from Reference [3], which incurs $O(n \log n)$ precomputation time and space, and $O(1)$ query time, where n is the length of array A. This scheme precomputes $O(\log n)$ RMQ queries starting from every element of A whose range size is a power of 2. It stores these values in a 2D array M, where each element $M[i][l]$ contains $RMQ(A, [i, i + 2^l - 1])$, i.e., the minimum element in $A[i..i + 2^l - 1]$. For example, in Figure 6, $M[1][2] = RMQ(A, [1, 4]) = 1$, whereas $M[2][2] = RMQ(A, [2, 5]) = 1$. The total space of M is $O(n \log n)$. Observe that each $M[i][l] = RMQ(A, [i, i + 2^l - 1])$ can be computed as the minimum of $RMQ(A, [i, i + 2^{l-1} - 1])$ and $RMQ(A, [i + 2^{l-1}, i + 2^l - 1])$ in $O(1)$ time. In other words, if we compute $M[i][l - 1]$ first for every $i \in [0, n - 1]$ before we compute $M[i][l]$, then the precomputation time becomes $O(n \log n)$.

Array M can be used to answer an RMQ query for range $[i, j]$ by accessing *exactly two* elements as follows. First, we calculate $l = \lfloor \log(j - i + 1) \rfloor$ and observe that $[i, j]$ is covered by two overlapping ranges, namely $[i, i + 2^l - 1]$ and $[j - 2^l + 1, j]$. In Figure 6, $l = \lfloor \log(7 - 3 + 1) \rfloor = 2$ and range $[3, 7]$ is covered by ranges $[3, 3 + 2^2 - 1] = [3, 6]$ and $[7 - 2^2 + 1, 7] = [4, 7]$. Also we observe that $RMQ(A, [3, 7]) = \min(RMQ(A, [3, 6]), RMQ(A, [4, 7]))$, since the fact that the two ranges are overlapping does not affect the minimum result value in the union of the ranges. However, we have precalculated $RMQ(A, [3, 6])$ and $RMQ(A, [4, 7])$ in M as elements $M[3][2]$ and $M[4][2]$ (as shown in the figure), respectively, which we can access in constant time. Therefore, to conclude, with $O(n \log n)$ preprocessing time and space we can answer RMQ queries in $O(1)$ time.

3 PROBLEM DEFINITION

We define the problem of RSSE in a very similar manner to SSE. In fact, the security game of RSSE is identical to that of SSE in Figure 2, where each w_i stands for a range query rather than a keyword. Moreover, similar to SSE, RSSE captures *index-based* schemes in its game; this is different from OPE that formulates a secure encryption scheme that allows ordered comparisons, without the need of an index. We employ this definitional framework to devise solutions that reduce range query search to *multi-keyword search*. This further allows us to build secure solutions on top of an existing secure SSE construction.

More specifically, we assume that data owner possesses a dataset D of tuples. We focus on range queries on a single attribute with domain A.[2] We associate a pair (id, a) with each tuple $d \in D$, where id is a unique identifier for d and a is the value of d on A. We also write $d.id$ and $d.a$ to refer to the elements of this pair. We assume that the owner encrypts each $d \in D$ using a semantically secure encryption scheme, and sends the resulting ciphertext c along with $d.id$ to the server. The goal is to build a secure index I on the $d.id$ values, such that the server can perform range queries that retrieve the set of IDs of the tuples satisfying the query. Note that, for each returned result $d.id$, the owner can retrieve from the server the corresponding ciphertext c of d and decrypt it in a subsequent step, orthogonal to the search process. In a nutshell, our RSSE framework can be summarized as follows:

- Index creation: Break A into a set of (potentially overlapping) ranges, and attribute a unique *keyword* to every range. Regard each $d \in D$ as a *document*, and associate it with the keywords of the ranges that include $d.a$. We hereafter use the terms document and tuple interchangeably. Utilize a static SSE scheme to securely index D using as dictionary Δ the union of the range keywords of every $d \in D$.
- Query: Break the query range into sub ranges, map them to keywords, and generate tokens for searching the SSE index.[3]
- Security: Augment the leakage functions $\mathcal{L}_1$ and $\mathcal{L}_2$ of the underlying SSE scheme to capture the extra leakage stemming from the keyword mapping and index structure and we prove that an RSSE scheme is ($\mathcal{L}_1$,$\mathcal{L}_2$)-secure using the security definition/game for SSE schemes presented in Section 2.2. It is out of the scope of this article to perform a leakage analysis, since it is a very complex procedure that is based on all the possible attacks that leverage the proposed leakages. Instead, we design schemes with a goal to minimize the leaked information.
- Updates: Perform updates in batches. For every batch, create a separate index using new keys. Periodically, consolidate separate indexes into a single one (consolidation is performed hierarchically, similar to log-structured merge trees [48]). This requires the owner to download the involved indexes, and create a single (re-encrypted) index. The server must issue every range query on each "active" index, and return the separate result sets.

An *RSSE protocol* is specified by algorithms Setup, BuildIndex, Trpdr, and Search, which are defined identically to those described in Section 2.2 for static single-keyword SSE, where w now stands for the query range. Their instantiation in each of our proposed schemes varies, but builds upon the constructions of traditional SSE. Our contribution revolves around the proper assignment

[2] We assume that the values of A are *positive integers*. Note that we can always convert any real domain to a discrete positive one by proper scaling and translation.

[3] An RSSE scheme supports empty range queries; if the client sends a token to the server that does not match any keyword inside the index, the server returns a null value.

D

id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	2	2	2	2	2	2	2	2	2	2	4	5	5	6	6	7
B	8	5	1	0	9	10	0	6	6	0	0	0	0	10	10	0

Fig. 7. Dataset used as a running example.

of keywords to tuples in BuildIndex, the mapping of a range query to keywords/tokens in Trpdr, and potentially the adjustment of Search to function appropriately with the tokens of Trpdr.

We support updates using only static SSE schemes. Our goal is *forward privacy*, i.e., the server should not learn that a newly added item satisfies a query issued in the past. Our mechanism is generic capturing all schemes and, thus, is detailed separately in Section 9.

Our RSSE schemes can build upon any SSE scheme. To provide context, in Sections 4–6 we assume that our methods use the SSE scheme of Reference [13],[4] which is locality non-aware. In Section 7, we discuss in detail how to attribute locality to our RSSE solutions by utilizing a locality-aware SSE scheme such as Reference [2] instead.

In addition to range queries that return the IDs of the qualifying tuples, we also target at another query type, namely *range aggregates*. A range aggregate query returns an aggregate value (instead of the tuple IDs) over the tuples that satisfy a specified query range. For the sake of generality, we assume that the range predicate is applied on attribute A, whereas the aggregate value is computed over a potentially different attribute B. In SQL syntax, our targeted range aggregate query is defined as follows:

```
SELECT agg(B)
FROM D
WHERE A >= l AND A <= u
```

The aggregate functions agg we focus on are sum, count, avg, min, and max. In Section 10, we initially design secure schemes for sum and min, but subsequently show that these can be modified to support all the above aggregate functions, and even more complex aggregate queries such as top-k and bottom-k. We explain that a range aggregate scheme follows the same framework as RSSE, i.e., it entails a secure index construction, a token and search algorithm upon a query, a practically identical security game, and support for updates.

Throughout the rest of the article, we will be using the dataset D shown in Figure 7 in our examples (where A is the range query attribute, and B the aggregate query attribute).

4 QUADRATIC SCHEME

Our Quadratic scheme is a naive baseline whose sole purpose is to help in conveying the basics of our RSSE framework. Let A be the query attribute domain, and m its total size. There are $O(m^2)$ possible range queries that can be applied in this domain. We enumerate all these possible sub ranges of A and assign a unique keyword to each range. Observe that a domain value belongs to $O(m^2)$ sub ranges. We associate each $d \in D$ with the keywords corresponding to the $O(m^2)$ ranges covering $d.a \in A$.

[4]The SSE scheme proposed by Cash et al. [13] is not the most efficient SSE at this moment. In fact, using as a black-box any of the schemes presented in the recent article of Demertzis and Papamanthou [23] will drastically improve the efficiency of the proposed RSSE by at least 4 orders of magnitude—we refer the reader to the recent article of Reference [23] for a comprehensive comparison of the state-of-the-art SSE schemes.

We start by replicating each $d \in D$ into $d'_1, \ldots d'_v$, where v is the number of keywords d corresponds to, and include the replicated tuples in a new dataset D'. We then use any secure (single-keyword) SSE scheme *off-the-shelf* (i.e., without any changes), treating each $d' \in D'$ as a separate tuple. The Setup and BuildIndex algorithms are identical as in the SSE scheme, where the tuples in D' are now augmented with the keywords described above. Given a range query, Trpdr simply maps it to the *single* keyword associated with its range, and the rest of the algorithm is the same as in the SSE scheme. Finally, Search is also used as in SSE without changes, and returns exactly the tuples in D' containing the range keyword. By definition, the returned tuples are exactly those satisfying the range query (without replication).

Table 1 in Section 1 shows the costs of Quadratic. Each tuple is associated with at most $O(m^2)$ keywords and, hence, the index size is $O(nm^2)$, where n is the number of tuples in D. The search time is inherited from the SSE scheme, and assuming Reference [13], this is $O(r)$, where r is the number of results. The query size is $O(1)$ as it involves a single keyword/token.

In terms of security, this technique does not introduce any additional leakage to what SSE reveals for D', namely its size. However, this may disclose information about the distribution of the values of D on A; two datasets with different distributions (e.g., one where all tuples have the same $d.a$ value, versus one where they all have a different one) will result in different D' sizes. This can be easily tackled by *padding* (e.g., as in References [12, 18]); for any D', the mechanism takes as input the cardinality n of D and the domain size m, and always constructs a secure index corresponding to the maximum possible D' size. Hence, only n, m are leaked in Quadratic in $\mathcal{L}_1$ along with the $\mathcal{L}_2$ leakage of the underlying SSE scheme, which results in the highest security level for our setting. However, Quadratic clearly suffers from a prohibitive storage cost, which motivates our next solutions.

5 CONSTANT SCHEMES

In this section, we present our Constant-BRC and Constant-URC schemes, which lie on the other side of the spectrum as far as the storage cost is concerned. Specifically, these techniques introduce a constant asymptotic overhead on the index size with respect to the dataset size n. Before embarking on their description, we first explain a naive variant.

We assign to each tuple $d \in D$ a *single* keyword, which corresponds to its actual value on A, namely $d.a$. In other words, the dictionary Δ is the values in A. No replication is involved. We then index D with an SSE scheme,[5] yielding an index I of size $O(n)$. A query range of size R is simply mapped to R keywords, one for each value of A it covers. We trivially use these keywords as search tokens in the Trpdr and Search algorithms of the SSE scheme. The scheme will return the correct r results without false positives in $O(R + r)$ time. Disregarding security for now, the main drawback of this scheme is the potentially unacceptable query size $O(R)$ for very large ranges. This motivates our Constant-BRC and Constant-URC solutions, which take advantage of DPRFs (explained in Section 2.2) to reduce the query size to $O(\log R)$.

The instantiations of the RSSE algorithms for Constant-BRC and Constant-URC are the following (the two algorithms differ only in the range-covering technique used for generating the trapdoor – BRC or URC):

$k \leftarrow \mathsf{Setup}(1^\lambda)$. Output (k_1, k_2), where k_1 is a DPRF key and k_2 is the key of the underlying SSE scheme.

[5]Constant schemes can be built with SSE schemes that utilize a PRF as a cryptographic primitive to encrypt the index, compute the token, and perform the search.

$I \leftarrow$ BuildIndex(k, D). Associate each $d \in D$ with keyword $d.a$. Invoke the BuildIndex algorithm of the SSE scheme on D and its keywords, but instead of using a PRF to encrypt the IDs, use a DPRF instead. Each $d.id$ is decrypted only with token $f_k(d.a)$, where f_k is a DPRF function.

$t \leftarrow$ Trpdr(k, w). Invoke the token generation function (T) of the DPRF employing either BRC or URC, and retrieve the corresponding GGM values corresponding to range w from the GGM tree over A (see Section 2.2). Randomly permute these GGM values and output them as vector t. For each GGM value in t, provide the level of its respective node in the GGM tree as well.

$X \leftarrow$ Search(t, I). Derive the (leaf-level) DPRF values from the GGM values in t. Use these values as tokens in the Search algorithm of SSE and return the results.

We clarify the algorithm revisiting the example of Figure 1, where $A = \{0, \ldots, 7\}$. The owner first generates a DPRF key k_1 in Setup, and computes the DPRF values for the elements on A that appear in D, creating a GGM tree. Suppose that a $d \in D$ has $d.a = 6$. In BuildIndex, the owner assigns 6 as keyword to d. However, contrary to traditional SSE where this document can be decrypted by using as token a PRF value on keyword 6, Constant-BRC/URC use a DPRF value instead. Specifically, they invoke the same BuildIndex algorithm as SSE (using SSE key k_2 generated in Setup), but the token to decrypt d (or any other tuple d with $d.a = 6$) is $f_{k_1}(6) = G_0(G_1(G_1(k_1)))$. The algorithm proceeds similarly with every other $d \in D$.

Upon a query, Trpdr outputs as token t the GGM values corresponding to the nodes covering the range with BRC or URC. In our example in Figure 1, if BRC is used, then the output is $t = \langle (G_1(G_0(k_1)), 1), (G_1(k_1), 2) \rangle$, i.e., the GGM values of nodes $N_{2,3}$ and $N_{4,7}$ along with their levels, respectively. If URC is used, then t contains $(G_0(G_1(G_0(k_1))), 0)$ for node N_2, $(G_1(G_1(G_0(k_1))), 0)$ for N_3, $(G_0(G_1(k_1)), 1)$ for $N_{4,5}$, and $(G_1(G_1(k_1)), 1)$ for $N_{6,7}$. Note that the elements of t are *randomly permuted*.

In Search, the server first takes the GGM values of the non-leaf nodes and expands them to compute the DPRF values for the leaves. For instance, from $N_{2,3}$'s value $G_1(G_0(k_1))$, it generates DPRFs $G_0(G_1(G_0(k_1)))$ and $G_1(G_1(G_0(k_1)))$. It can do that because (i) G is public and (ii) it knows the level of $G_1(G_0(k_1))$, i.e., 1. It finally uses as tokens the DPRFs to retrieve the results, invoking SSE's Search algorithm.

The cost complexities of Constant-BRC and Constant-URC are identical and provided in Table 1. Each tuple is associated with a single keyword and, hence, the storage cost is $O(n)$. Due to the BRC/URC techniques, the query size is $O(\log R)$ for a range size R. The search time at the server entails expanding the $O(\log R)$ GGM values into R DPRFs, and retrieving the r results from SSE, yielding a total $O(R + r)$ time. Both solutions do not introduce false positives.

We next turn to security. Our Constant constructions cannot be proven secure against adaptive adversaries that are allowed to issue *intersecting* range queries. This is an inherent limitation of our underlying DPRFs, as shown in Reference [47]. Briefly stated, this is because the simulator must have *a priori* knowledge of the GGM sub-structure shared by the intersecting ranges to produce consistent tokens. However, this is not possible in the adaptive case. In the example of Figure 1, suppose that some query generated a token that includes the GGM value for node $N_{2,3}$. Then, let another query involve producing as token a GGM value for node N_2. Without the *a priori* knowledge that the second range intersects the first at N_2, the simulator could not have generated the GGM value for $N_{2,3}$, in a way that it can produce the GGM value for N_2.

Consequently, the Constant schemes limit the functionality by *not allowing query intersections*. Note that this constraint can be enforced at the application level. For instance, the owner's program may maintain the history of queries and abort when an intersecting query is seen, or may try to

answer the query from cached answers of previous queries that collectively encompass the new query range.

To prove security (under the constraint of non-intersecting queries), we define the two leakages as follows:

— $\mathcal{L}_1(D, A) = \langle n \rangle$
D is the dataset, A is the query attribute domain, n is the cardinality of D.
— $\mathcal{L}_2(D, A, W) = \langle \alpha(W), \sigma(W), ((\mu(N_i), \ell(N_i), id_{map}(N_i))_{N_i \in RC(w)})_{w \in W} \rangle$
$\alpha(W), \sigma(W)$ are the access and search patterns of the queries as defined for SSE. The extra leakage is as follows. For every query range $w \in W$, the leakage contains a tuple that consists of an alias $\mu(N_i)$ for every node N_i returned by the range covering $RC(w)$—where the RC function is either BRC or URC—along with the level $\ell(N_i)$ of N_i, and the exact mapping $id_{map}(N_i)$ of the tuple ids to the leaves of $\mu(N_i)$'s sub tree.

Observe that, contrary to traditional SSE, the two leakage functions take as input also the query attribute domain A. This is because our constructions build an index considering the entire span of A. We further explain the extra leakage in $\mathcal{L}_2(D, A, W)$ incurred by our schemes with an example using Figure 1. Let the first query be $w_1 : [0, 3]$, with results d_1, d_2, such that $d_1.a = 0$ and $d_2.a = 3$. Then, $\mathcal{L}_2$ over $W = \{w_1\}$ is an alias for node $N_{0,3}$, its level (2), and the information that d_1 maps to its left-most leaf, and d_2 to its right-most leaf. Now suppose that the second query is $w_2 = [5, 7]$. Then, $\mathcal{L}_2$ over $W = \{w_1, w_2\}$ is what is explained above, plus aliases for nodes $N_5, N_{6,7}$ (without disclosing their relative order) as well as the mappings of the qualifying tuples in these sub trees. Note that, neither the relative order of w_1, w_2 on A, nor the relative order of the sub trees of each query are revealed. We include the security theorem and proof of Constant BRC/URC in Appendix A.1.

Qualitative Comparison. The Constant schemes feature considerably better storage than Quadratic (at the expense of slightly increased query size and search time). However, they also introduce a significantly higher leakage, since now the structure of the subtree, the number of leaves, and the encrypted tuples of each leaf in the subtree are all leaked. Consequently, the aforementioned leakages further reveal information about the relative order of the encrypted tuples. Comparing the BRC and URC variants, URC offers slightly better privacy; the BRC coverage may exclude mapping certain ranges to the query, whereas URC covers all ranges of the same size in an indistinguishable manner.

6 LOGARITHMIC SCHEMES

Motivated by the high structural leakage of the Constant schemes, in this section we design solutions that trade off storage for privacy. This is achieved by replicating tuples similar to Quadratic, but with a significantly lower expansion factor. Specifically, we present five constructions that increase the storage complexity only by a logarithmic factor.

The constructions rely on the common idea of covering the query attribute domain with dyadic intervals of different lengths, which are conceptually organized in a hierarchy, and associating each tuple with all intervals that cover its attribute value. Queries are then executed by retrieving and combining a small number of intervals. The five constructions differ on how they construct these intervals and how they choose the intervals to be retrieved during query execution, leading to differences in their achieved levels of security and efficiency (expressed in terms of the total querying cost and the retrieved number of false positives). In particular:

— The Logarithmic-BRC and Logarithmic-URC schemes, described in Section 6.1, transform the query to the same (logarithmic in number) dyadic intervals as in Constant BRC/URC. We will explain that the added security stems from avoiding the use of DPRFs and from

associating each tuple with a logarithmic, instead of constant, number of keywords. Both schemes rely on a single round of communication between the owner and the server for the query execution and return no false positives.

—The Logarithmic-SRC scheme, introduced in Section 6.2, covers a query range of size R with a single interval of dyadic size $O(R)$, using a novel TDAG graph that we introduce. This scheme also relies on a single round of communication between the owner and the server for the query execution, but introduces *false positives* because the used interval may be larger than the query interval.

—To reduce the number of false positives of the Logarithmic-SRC scheme, we then present in Sections 6.3 and 6.4 the Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i$_2$ schemes, respectively. Both of these schemes are *interactive*, requiring *two* rounds of communication, utilize two indexes, and may return false positives. The data is stored at the TDAG$_2$ index after appropriate sorting and shuffling. The first index (TDAG$_1$ in Logarithmic-SRC-i$_1$, or an array in Logarithmic-SRC-i$_2$) is smaller and helps map each query range to a query range at TDAG$_2$ (thus, it is queried first). This query range is then transformed, by the owner, to another single dyadic range that covers it, and the second part of the query is issued to the server. The two schemes differ in their first index and the way that they are built and queried.

6.1 Logarithmic-BRC/URC

The Logarithmic-BRC and Logarithmic-URC schemes mitigate the structural leakage of Constant-BRC/URC by avoiding the use of DPRFs and associating each tuple with a logarithmic, instead of constant, number of keywords. The RSSE protocol for Logarithmic-BRC and Logarithmic-URC is as follows (the two schemes again differ in the range-covering technique used in Trpdr):

$k \leftarrow \mathsf{Setup}(1^\lambda)$. Same as in SSE.

$I \leftarrow \mathsf{BuildIndex}(k, D)$. Build a binary tree over domain A as described in Section 2.2, and assign a unique keyword at each node. For each tuple $d \in D$, find the nodes on the path from the tree root to $d.a$, and associate d with the node keywords. Replicate every d for each keyword it is associated with, and regard each replica as separate tuple as discussed in Quadratic. Let D' be the resulting dataset including all the replicas. Invoke the BuildIndex algorithm of the SSE scheme on D' and its keywords, after randomly permuting the documents that are associated with the same keyword.

$t \leftarrow \mathsf{Trpdr}(k, w)$. Let w represent the query range. Find the nodes that cover w using BRC or URC. Create a token for each node keyword invoking the Trpdr algorithm of the SSE scheme and include it in t. Randomly permute t prior to returning it.

$X \leftarrow \mathsf{Search}(t, I)$. Invoke the Search algorithm of SSE for every element in t, and output the union of the results.

The protocol associates each tuple with the *dyadic intervals* that cover its attribute value. For instance, if $d.a = 3$ in Figure 1, then d is associated with keywords $N_{0,7}$, $N_{0,3}$, $N_{2,3}$, and N_3 (where each N_i is the label of a node). We then replicate each d and index the augmented dataset D' with traditional SSE. Given a range query, we compute its cover with BRC or URC, and issue the output node labels of the range-covering technique as keywords for traditional SSE search, i.e., we invoke the conventional Trpdr and Search algorithms of SSE for every node label and corresponding token, respectively. For example, for query $[2, 7]$ under BRC, Trpdr outputs an SSE token for $N_{2,3}$ and $N_{4,7}$ in random order, and the Search SSE algorithm is invoked for every such token separately.

The costs of Logarithmic-BRC and Logarithmic-URC are summarized in Table 1. The index size is $O(n \log m)$, where n is the number of tuples and m the domain size, since each tuple is associated with $O(\log m)$ keywords. The query size is $O(\log R)$, where R is the size of the query range, since

this is the number of nodes covering the querying range in BRC and URC. The search time is $O(\log R + r)$, where r the result size, because there are $\log R$ tokens issued to the underlying SSE scheme, each incurring no additional cost to the retrieval of its results. Finally, the protocol returns correct answers without false positives, since BRC and URC cover the query exactly, and a tuple in the result is certainly associated with the keyword of a node in the cover.

The only extra information leaked is a *partitioning* of the result IDs into groups. More formally:

- $\mathcal{L}_1(D, A) = \langle m, n \rangle$
  D is the dataset, A is the query attribute domain, n is the cardinality of D, and m is the size of A.
- $\mathcal{L}_2(D, A, W) = \langle \alpha(W), \sigma(W), ((\mu(N_i), id(N_i))_{N_i \in RC(w)})_{w \in W} \rangle$
  $\alpha(W), \sigma(W)$ are the access and search patterns of the queries as defined for SSE. For every query range $w \in W$, the leakage contains a tuple that consists of an alias $\mu(N_i)$ for every node N_i returned by the range covering $RC(w)$—where the RC function is either BRC or URC—along with the list of tuple ids $id(N_i)$ associated with keyword N_i.

We include the security theorem and proof of Logarithmic BRC/URC in Appendix A.2.

Qualitative Comparison. Logarithmic-BRC/URC feature increased storage cost compared to Constant-BRC/URC, but a better search time (since the server does not need to generate DPRFs from the tokens). The main benefit of Logarithmic-BRC/URC is in the substantially reduced leakage, since they hide both the distribution and the total order of the tuples in each subtree of the query range cover. The difference between the BRC and URC variants of Logarithmic is similar to that in Constant; URC does not disclose information about the position of the range over the domain. Nevertheless, what is still leaked in both variants is the *partitioning* of the result tuples into distinct groups (each corresponding to a subtree), which may further disclose ordering information about the groups. This motivates our solution in the next section.

6.2 Logarithmic-SRC

The extra "result partitioning" leakage of the Logarithmic-BRC/URC schemes was due to the fact that Trpdr produces *multiple* tokens, one for each subtree with which BRC or URC cover the query range. Logarithmic-SRC prevents this leakage by always covering the query range with a *single* range that is potentially a *superset* of the query range. In addition to enhanced privacy, this also leads to *constant* query size, but may introduce *false positives*.

We can naively realize Logarithmic-SRC building upon the same binary tree over A as in Logarithmic-BRC/URC as follows. We assign once again to each $d \in D$ the keywords corresponding to the nodes whose subtrees cover $d.a$, and replicate tuples to create an augmented dataset D'. However, instead of invoking BRC or URC in Trpdr to find the cover of the query range with tree nodes, we could simply select the node of the smallest subtree fully covering the query, and use its keyword for searching. In our example of Figure 1, we would cover query $[2, 7]$ with the tree root $N_{0,7}$. Unfortunately, this solution features an unacceptable worst-case complexity for the number of leaves (i.e., domain values) contained in the single subtree, which is $O(m)$ where m is the domain size, regardless of the query range size R. This would further lead to an unacceptable number of false positives. For example, in Figure 1, query $[3, 4]$ is covered by the tree root that encompasses the entire domain and, hence, dataset D. Motivated by the above, in Logarithmic-SRC we produce keywords for the tuples in D based on a novel *tree-like DAC*, henceforth referred to as *TDAG*, which ensures that *any* query range of size R is covered by a single subtree with size $O(R)$.

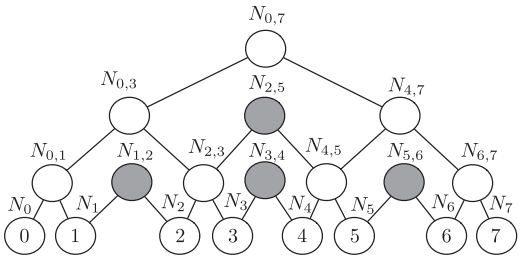


Fig. 8. TDAG example.

We explain the TDAG structure using Figure 8. We start by building a binary tree over domain A, similar to the case of the previous schemes. We then inject one extra node between every two nodes at every level of the tree (depicted in gray), and connect it with the two nodes directly below it in the next level (i.e., the right child of the node in its left, and the left child of the node in its right).

The following lemma is useful for our cost analysis of Logarithmic-SRC.

Lemma 6.1. *Given a TDAG constructed over a domain A and any range in A of size R, there is always a subtree of size $O(R)$ that can completely cover the range.*

Proof. For any integer $R > 0$, there is an integer $j \geq 0$ such that $2^j \leq R \leq 2^{j+1}$. Any range of size R can be covered by at most two dyadic ranges (i.e., subtrees in the binary tree over A) of size 2^{j+1}; either the range is fully contained in such a subtree, or it is split between two *consecutive* subtrees of size 2^{j+1}. These two subtrees are either children of the same parent in the binary tree or cousins. Recall that our TDAG structure essentially links every two cousins in each level with a new parent. Hence, there is always a node that covers the two subtrees each of size $2^{j+1} \leq 2R$. Thus, for any range of size R, there is always a subtree in TDAG with size at most $4R \in O(R)$. □

Given a range of size R, the SRC range-covering algorithm simply finds the *lowest common ancestor* of the lower and upper bound of the range, which can be performed in $O(\log R)$ time. In the example of Figure 8, SRC covers range $[2, 7]$ by $N_{0,7}$, and range $[3, 5]$ by $N_{2,5}$.

The RSSE algorithms for Logarithmic-SRC are the same as in Logarithmic-BRC/URC with the following differences: (i) in BuildIndex, each $d \in D$ is associated with the keywords/labels of the nodes of TDAG that cover $d.a$ (instead of the nodes of the binary tree) and (ii) Trpdr generates a single token for the node label output by the SRC covering technique (instead of BRC/URC).

Table 1 summarizes the costs of Logarithmic-SRC. The query size is constant, since the query is represented by a single token. The index has size $O(n \log m)$; for each tuple $d \in D$, there are $O(\log m)$ nodes in the path from the root to $d.a$, and each such node is connected to at most one injected node in the TDAG; the subtrees of all these $O(\log m)$ nodes cover $d.a$ and, thus, each d is associated with $O(\log m)$ keywords. The false positives depend on the dataset *distribution* over A. If the distribution is *uniform*, then the false positives are $O(R)$ due to Lemma 6.1. However, if the dataset is *skewed*, then the false positives could be up to $O(n)$. For example, if $[3, 5]$ is the query in Figure 8 and there is a single tuple that satisfies the range, but the rest of the dataset has value 2 on A, then Logarithmic-SRC will return the entire dataset due to the used keyword $N_{2,5}$. The search time is linear in the result size plus the false positives and, thus, it is $O(n)$ in the worst case where there is heavy skew.

In Logarithmic-SRC, any range query is reduced to a *single-keyword* query. Thus, Logarithmic-SRC degenerates to a traditional SSE scheme, inheriting its security (assuming the padding technique discussed in Quadratic). However, for technical reasons in our proofs, we must define an

extra subtle leakage, namely the fact that two different ranges may map to the same keyword (e.g., $[0, 2]$ and $[0, 3]$ in Figure 8). This can be modeled by extending the definition of search patterns. Nevertheless, from a practical point of view, this leakage is not observable by an adversary, since the mapping takes place at the owner. In particular, the adversary is unable to distinguish if the same token was produced twice for the same or for different range queries.

We describe the leakage of Logarithmic-SRC more formally:

- $\mathcal{L}_1(D, A) = \langle m, n \rangle$
  D is the dataset, A is the query attribute domain, n is the cardinality of D, and m is the size of A.
- $\mathcal{L}_2(D, A, W) =$
  $\langle \alpha(W), \sigma(W), (\mu(RC(w)), id(RC(w))$
  $\alpha(W), \sigma(W)\rangle$ are the access and search patterns of the queries as defined for SSE. For every query range $w \in W$, the leakage contains a tuple that consists of an alias $\mu(RC(w))$ for the node returned by the range covering $RC(w)$, along with the list of tuple ids $id(RC(w))$ associated with keyword $RC(w)$.

We include the security theorem and proof of Logarithmic-SRC in Appendix A.3.

Qualitative Comparison. Contrary to Logarithmic-BRC/ URC, in Logarithmic-SRC the adversary is unable to infer ordering information about the results, since each range is mapped to a *single* keyword and the tuples associated with this keyword are *randomly permuted.* Logarithmic-SRC also features optimal query size and the highest achievable privacy in our RSSE framework that builds upon single-keyword SSE. Similar to Logarithmic-BRC/URC, this comes at the cost of extra storage. Logarithmic-SRC is ideal for datasets with uniform distributions over the query attribute domain. However, it may feature an unacceptable number of false positives (and, thus, also search time) under heavy data skew. This is mitigated by our final solution described in the next section.

6.3 Logarithmic-SRC-i$_1$

The Logarithmic-SRC-i$_1$ construction aims at reducing the false positives of Logarithmic-SRC from $O(n)$ to $O(R + r)$, where R is the query range size and r is the result size. It achieves this by building a *double* index $I = (I_1, I_2)$, where I_2 indexes the tuples in D similar to the previous schemes, and I_1 is an *auxiliary* index that guides the search to I_2. This construction is *interactive* (hence the "i" in the name), i.e., it involves an extra round of communication between the owner and the server during the query; the owner first queries I_1, it receives the result from the server, and based on the result it queries I_2.

We illustrate the construction of I_1 and I_2 with the example of Figure 9. Suppose that $D = \{d_0, \ldots, d_{15}\}$. Assume also for simplicity that $d_0, \ldots, d_{15}$ are sorted on A, so that the subscript i of each d_i implies its position in the total order of the tuples on A. Also consider that $d_0.a = \ldots d_9.a = 2$, $d_{10}.a = 4$, $d_{11}.a = d_{12}.a = 5$, $d_{13}.a = d_{14}.a = 6$, and $d_{15}.a = 7$. Consider TDAG$_1$ in the upper part of the figure, which is built on domain $A = \{0, \ldots, 7\}$, and let $[3, 5]$ be the query range. Recall that Logarithmic-SRC answers this query with TDAG$_1$ by using keyword $N_{2,5}$. This returns as false positives $d_0, \ldots, d_9$ corresponding to domain value 2, which comprise more than half of D.

Instead of using TDAG$_1$ to index the tuples, Logarithmic-SRC-i$_1$ uses it to index the *ranges* of tuples corresponding to the domain values, where a tuple range accounts for a range of tuple subscripts. Specifically, each leaf is associated with a pair (domain value, tuple range), e.g., N_2 is associated with $(2, [0, 9])$, since $d_0, \ldots, d_9$ all have domain value 2. The non-leaf nodes are associated with the lists of (domain value, tuple range) pairs of the leaves in their subtree, e.g., $N_{2,5}$ corresponds to $\langle (2, [0, 9]), (4, [10, 10]), (5, [11, 12])\rangle$. I_1 is constructed using the traditional SSE

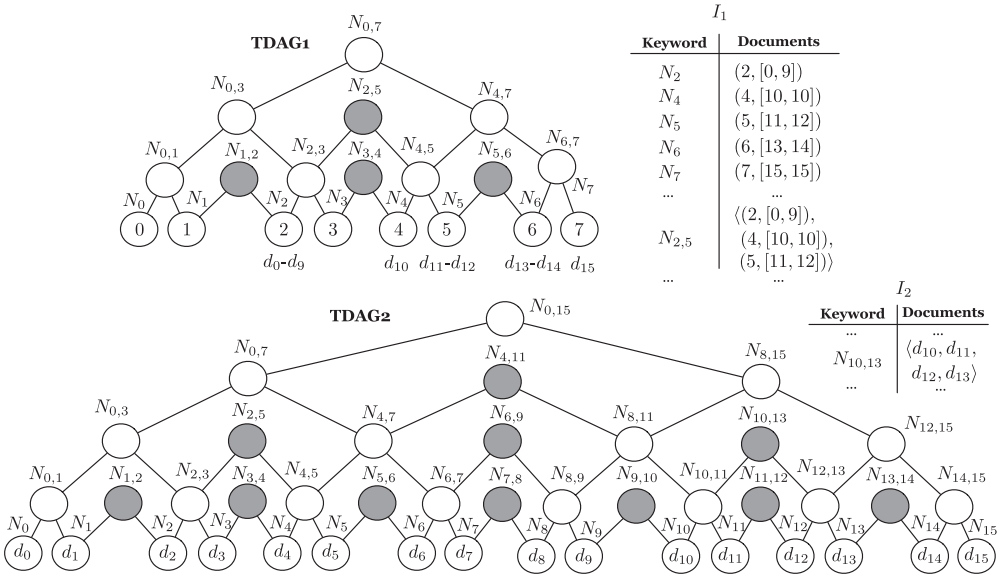


Fig. 9. Building the index in Logarithmic-SRC-i$_1$.

BuildIndex algorithm, where the documents are the pairs or lists of pairs described above, and the keywords are the TDAG$_1$ node labels.

The construction then builds a second TDAG, denoted by TDAG$_2$ in the lower part of the figure, which is built on the *tuples* sorted on A. Note that the order of the documents corresponding to the same keyword (i.e., to the same node in TDAG$_1$) does not affect the structure of TDAG$_2$. For instance, TDAG$_2$ would be equivalent if we swapped d_{11} with d_{12} in the leaf level (where d_{11} and d_{12} have the same keyword). Prior to constructing TDAG$_2$, we randomly shuffle the documents corresponding to the same keyword. Each node corresponds to the tuples covered by its subtree, e.g., $N_{10,13}$ is associated with d_{10}, d_{11}, d_{12}, and d_{13}. I_2 is constructed using the traditional SSE BuildIndex algorithm, where the documents are $d_0, \ldots, d_{15}$, and the keywords are the TDAG$_2$ node labels.

The scheme performs the query in two stages. In the first, it issues the query range to I_1. For instance, for $[3,5]$ it creates a token for $N_{2,5}$ (following always the SRC range-covering technique on the TDAG), and receives $\langle(2,[0,9]),(4,[10,10]),(5,[11,12])\rangle$. It then selects the pairs that satisfy the query, and creates a new, *single* query range on the document subscripts by merging the qualifying ranges. In our example, it merges $[10,10]$ and $[11,12]$ to create new query $[10,12]$ ([0,9] does not satisfy the original query). In the second stage, it issues $[10,12]$ to I_2, creating the token for node $N_{10,13}$ in TDAG$_2$, which fully covers the tuple range. The algorithm returns as result d_{10}, d_{11}, d_{12}, and d_{13}. Observe that now there is only a single false positive, d_{13}, whereas Logarithmic-SRC returned 10 false positives for the same query. The RSSE protocol for Logarithmic-SRC-i$_1$ is as follows:

$k \leftarrow \mathsf{Setup}(1^\lambda)$. Generate and output two SSE keys (k_1, k_2).

$I \leftarrow \mathsf{BuildIndex}(k, D)$. Build SSE index I_1 on the tuple ranges using TDAG$_1$ with key k_1, and index I_2 on the sorted tuples on A using TDAG$_2$ with key k_2. Output (I_1, I_2).

$t \leftarrow \mathsf{Trpdr}(k, w)$. This is an interactive algorithm. Parse $k = (k_1, k_2)$. Generate SSE token t_1 with k_1 for the SRC node on TDAG$_1$ that covers range w, and send it to the server. Decrypt

the answer to retrieve new range w'. Generate SSE token t_2 with k_2 for the SRC node on TDAG_2 that covers w', and output (t_1, t_2).

$X \leftarrow \text{Search}(t, I)$. This is an interactive algorithm. Parse $t = (t_1, t_2)$ and $I = (I_1, I_2)$. Retrieve t_1 from the owner, invoke the Search algorithm of SSE on I_1 and send the result to the owner. Retrieve t_2 from the owner, invoke the Search algorithm of SSE on I_2 and output the result X.

Table 1 illustrates the costs of Logarithmic-SRC-i_1. The number of documents indexed by I_1 is equal to the number of *distinct* domain values contained in the dataset, since we index the entire range of tuples on a specific domain value by a single (domain value, tuple range) document of constant size. Therefore, since I_1 is constructed using a TDAG similar to Logarithmic-SRC, its size is $O(n \log m)$. The number of documents indexed by I_2 is $O(n)$, i.e., the entire D. Contrary to I_1, the TDAG is built on n leaves and, thus, the size of I_2 is $O(n \log n)$. Assuming that m is typically larger than n, the total storage cost becomes $O(n \log m)$. The query size is $O(1)$, since there are only two tokens involved. The false positives are $O(R + r)$, considering also those of I_1. This is because the range size in I_1 is of size $O(R)$, whereas in I_2 is of size $O(r)$; due to Lemma 6.1, the number of false positives in each index is linear in the query range size and, thus, the total false positives are $O(R + r)$. The search time is dictated by the number of results plus the false positives and, thus, it is also $O(R + r)$.

Since I_1 and I_2 are built following the construction algorithm of the underlying SSE protocol and using two different keys, the leakage in each index is identical to that of the SSE scheme. Therefore, having the $\mathcal{L}_1$ and $\mathcal{L}_2$ leakages of SSE for *both* indexes and the mapping between the nodes of the first index to the nodes of the second index, we can prove the security of Logarithmic-SRC-i_1.

We describe the leakage of Logarithmic-SRC-i_1 more formally:

— $\mathcal{L}_1(D, A) = \langle m, n, n' \rangle$
D is the dataset, A is the query attribute domain, n is the cardinality of D, n' is the cardinality of unique values of D and m is the size of A.

— $\mathcal{L}_2(D, A, W) =$
$\langle \alpha(W), \sigma(W), (\mu(RC(w))_{TDAG_1}, \mu(RC(w'))_{TDAG_2}, uqv(RC(w)), id(RC(w'))) \rangle$
$\alpha(W), \sigma(W)\rangle$ are the access and search patterns of the queries as defined for SSE. For every query range $w \in W$, the leakage contains a tuple that consists of an alias $\mu(RC(w))_{TDAG_1}$ for the node returned by the range covering $RC(w)$ in $TDAG_1$ (similarly for $TDAG_2$), along with the unique domain values stored in $TDAG_1$ and the list of tuple IDs $id(RC(w))$ associated with the keyword $RC(w')$. It is worth mentioning that the combinations of $(\mu(RC(w))_{TDAG_1}, \mu(RC(w'))_{TDAG_2})$ with $\sigma(W)$ also leak the relation between a node in $TDAG_1$, with a node in $TDAG_2$, i.e., more than one node from $TDAG_1$ can be associated with a node in $TDAG_2$ and vice versa.

We include the security theorem and proof of Logarithmic-SRC-i_1 in Appendix A.4.

Qualitative Comparison. Logarithmic-SRC-i_1 reduces the false positives as compared to Logarithmic-SRC, even in the case of heavy data skew, while retaining the optimal query size and storage cost $O(n \log m)$. As a downside, the usage of the auxiliary index leaks slightly more information than its counterpart. For instance, the size of I_1 (derived from $\mathcal{L}_1$ of I_1) leaks the number of distinct domain values covered by the dataset, whereas the size of a result from a query to I_1 (derived from $\mathcal{L}_2$ of I_1) reveals the number of distinct domain values covered by the result. Moreover, in *non-skewed datasets*, Logarithmic-SRC-i_1 inflicts higher search cost than Logarithmic-SRC. This is because the benefits of using the extra I_1 index in Logarithmic-SRC-i_1 to reduce the false positives diminish and, thus, the extra search overhead compared to Logarithmic-SRC becomes

evident. In that sense, Logarithmic-SRC-i$_1$ is better under data skew, whereas Logarithmic-SRC is preferable in non-skewed datasets.

6.4 Logarithmic-SRC-i$_2$

In the previous subsection, we showed that Logarithmic-SRC-i$_1$ improved upon Logarithmic-SRC by reducing the complexity of false positives from $O(n)$ to $O(R + r)$, where n is the dataset size, R is the query range size, and r is the actual result size. In this subsection, we present another scheme variant, called Logarithmic-SRC-i$_2$, which further reduces the false positives to $O(r)$. However, this comes with some extra cost on storage, which now becomes $O(m + n \log n)$ from $O(n \log m)$.

The main observation is that the R term in $O(R + r)$ comes only because of TDAG$_1$. This is because, in TDAG$_2$, the range query consists only of the actual ID range that satisfies the result and, thus, has size $R = r$. Due to Lemma 6.1, the false positives in TDAG$_2$ is $O(r)$. To better illustrate the fact that TDAG$_1$ indeed may generate up to $O(R)$ false positives, consider range $[1, 4]$ in Figure 9. This range is answered with node $N_{0,7}$ as the single keyword query. Regardless of whether nodes N_1-N_4 are empty, if nodes N_0 and N_5-N_7 are non-empty, their values are returned as results to the owner as false positives. Observe that the number of these values is $R/2$.

Motivated by the above, the key idea of Logarithmic-SRC-i$_2$ is to substitute TDAG$_1$ with a simple array, denoted by T$_1$. The scheme is still interactive but, contrary to TDAG$_1$, the purpose of T$_1$ is not to get all the ID ranges falling inside the query range. Instead, the goal is simply to get the IDs of the *first* and *last* document that satisfy the range query. Given this information, the owner can craft the next range query token for TDAG$_2$ in an identical manner to Logarithmic-SRC-i$_1$.

Figure 10 explains the index creation in Logarithmic-SRC-i$_2$ for the same example as that in Figure 9 for Logarithmic-SRC-i$_1$. Table T$_1$ in the top part of Figure 10 has one element has one element for each value of the domain, and element T$_1[i]$ simply lists the tuples with $d.a = i$. The mere purpose of this table is to construct the secure index I_1 as follows. For *every* domain value i we create two keywords, (i, L) and (i, R). (i, L) is associated with the ID of the *last* tuple with $d.a \leq i$, whereas (i, R) is associated with the ID of the *first* tuple with $d.a \geq i$. For example, keyword $(3, \mathrm{L})$ is associated with 9 since d_9 is the last tuple with search key smaller than 3, whereas keyword $(3, \mathrm{R})$ is associated with 10 since d_{10} is the first tuple with search key larger than 3 (since there is no tuple with search key equal to 3). The size of I_1 is clearly $O(m)$ since the domain has size m. Moreover, it can be constructed in time $O(m + n)$ given T$_1$, since T$_1$ can be populated in $O(n)$ time (it is a simple look-up table). Index I_2 is constructed identically to the case of Logarithmic-SRC-i$_1$ and, thus, has size $O(n \log n)$. The total space requirements of Logarithmic-SRC-i$_2$ is $O(m + n \log n)$.

We explain how a range query is performed using the example of Figure 10, assuming that $[3, 5]$ is the query range. Similar to Logarithmic-SRC-i$_1$, the query involves two rounds. In the first round, the owner issues two keywords, namely $(3, \mathrm{R})$ and $(5, \mathrm{L})$ on I_1. Observe that these queries will return the first and last tuple id qualifying as results, namely 10, 12, by definition of the I_1 construction. Subsequently, the owner issues range query $[10, 12]$ to the second index I_2 in the same manner as in Logarithmic-SRC-i$_1$, which returns results d_{10}, d_{11}, d_{12}, as well as d_{13} as the single false positive. The query size is $O(1)$.

The RSSE protocol for Logarithmic-SRC-i$_2$ is as follows:

$k \leftarrow \mathsf{Setup}(1^\lambda)$. Generate and output two SSE keys (k_1, k_2).

$I \leftarrow \mathsf{BuildIndex}(k, D)$. Build SSE index I_1 using T$_1$ (as explained in Figure 10) with key k_1, and index I_2 on the sorted tuples on A using TDAG$_2$ with key k_2. Output (I_1, I_2).

$t \leftarrow \mathsf{Trpdr}(k, w)$. This is an interactive algorithm. Parse $k = (k_1, k_2)$ and range $w = [i, j]$. Generate SSE token $t_1 = (t_{11}, t_{12})$ with k_1 for keywords (i, R) and (j, L). Decrypt the answer

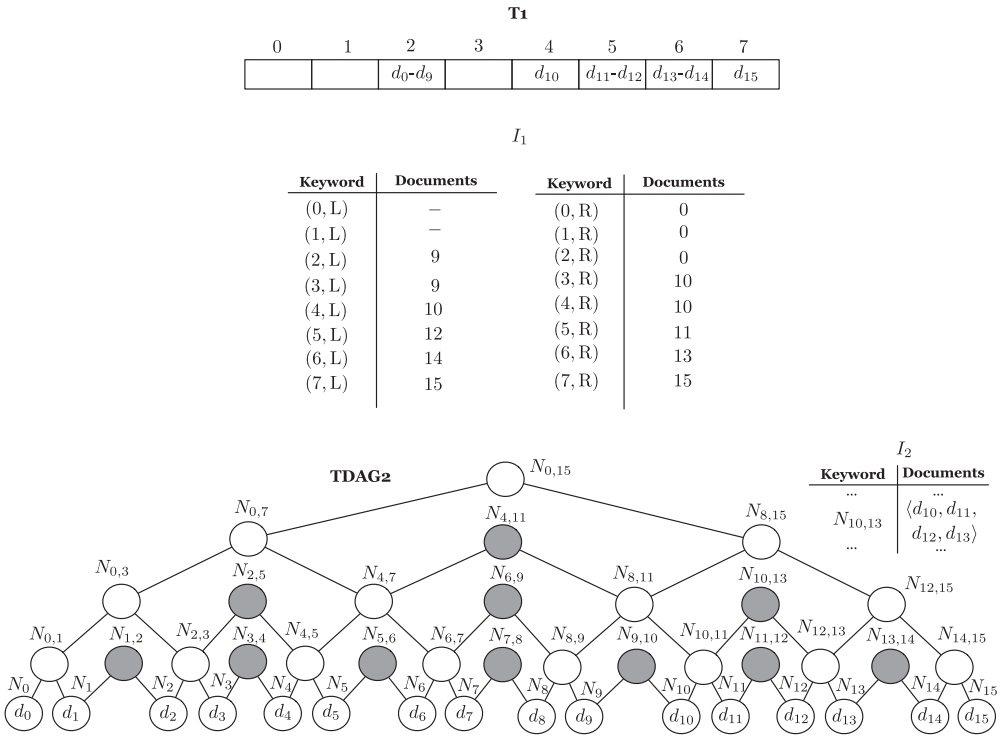


Fig. 10. Building the index in Logarithmic-SRC-i$_2$.

to retrieve new range w'. Generate SSE token t_2 with k_2 for the SRC node on TDAG$_2$ that covers w', and output (t_1, t_2).

$X \leftarrow$ Search(t, I). This is an interactive algorithm. Parse $t = (t_1, t_2)$, $t_1 = (t_{11}, t_{12})$ and $I = (I_1, I_2)$. Retrieve t_1 from the owner, invoke the Search algorithm of SSE for t_{11}, t_{12} on I_1 and send the results to the owner. Retrieve t_2 from the owner, invoke the Search algorithm of SSE on I_2 and output the result X.

We describe the leakage of Logarithmic-SRC-i$_2$ more formally:

— $\mathcal{L}_1(D, A) = \langle m, n\rangle$
D is the dataset, A is the query attribute domain, n is the cardinality of D, and m is the size of A.

— $\mathcal{L}_2(D, A, W) =$
$\langle \alpha(W), \sigma(W), (\mu(w_L), \mu(w_R), \mu(RC(w)), id(RC(w))\rangle$
$\alpha(W), \sigma(W)\rangle$ are the access and search patterns of the queries as defined for SSE. For every query range $w \in W$, the leakage contains a tuple that consists of an alias $\mu(w_L)$, an alias $\mu(w_R)$ for the tokens returned by the first encrypted index, an alias $\mu(RC(w))$ for the query in $TDAG_2$, and the list of tuple ids $id(RC(w))$ associated with the keyword $RC(w)$ in $TDAG_2$.

We include the security theorem and proof of Logarithmic-SRC-i$_2$ in Appendix A.5.

Qualitative Comparison. Compared to Logarithmic-SRC-i$_1$, Logarithmic-SRC-i$_2$ reduces both the false positives and search time to $O(r)$, but increases the storage cost to $O(m + n \log n)$. In addition,

it introduces slightly higher leakage in index I_1. Specifically, it leaks whether two ranges start or end at the same endpoint. For instance, for queries $[3, 5]$ and $[3, 6]$ that have the same start, the owner would issue the same token for keyword $(3, \mathrm{R})$ for both ranges.

7 ATTRIBUTING LOCALITY

So far, we have assumed that all presented RSSE schemes utilize the SSE construction from Reference [13]. This technique does not expand the storage required by each RSSE scheme, which we analyzed and summarized in Table 1. However, as we explained in Section 2.2, the SSE scheme of Reference [13] does not offer locality, which may lead to an excessive number of PRF computations and random accesses in the server's storage (equal to the tuples retrieved as result). In this section, we explore ways to use a locality-aware SSE scheme, while analyzing the caveats and opportunities for each RSSE scheme. In particular, in Section 7.1 we explain a generic way to attribute locality to all our RSSE schemes using a locality-aware SSE construction as a black box, and outline the impact on the various cost complexities with such an approach. In Section 7.2 we identify a unique opportunity for attributing locality specifically to Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i$_2$ without compromising any cost complexity, and devise a new locality-aware RSSE scheme, called *Logarithmic-SRC-i**.

7.1 A Generic Approach

A general approach to attributing locality to our RSSE schemes is to just substitute the locality non-aware SSE scheme of Reference [13] with a locality-aware one, such as Approach #3 from Reference [2], and use it as a *black box* in our constructions. This would allow every RSSE scheme to achieve optimal locality (as well as read efficiency). However, after a careful analysis, one can see that this would impact the storage complexity of all methods, as explained below.

Let us focus on Logarithmic-SRC for the sake of demonstration (similar arguments can be made for the other RSSE schemes as well). Recall that Logarithmic-SRC essentially creates a new dataset that contains keyword/document ID pairs (where the keyword is essentially the label of a TDAG node), whose size is $O(n \log m)$. Recall from Table 2 that the storage complexity of Approach #3 from Reference [2] is $O(n \log n)$, where n is the input dataset size. Consequently, substituting n for $n \log m$ (i.e., the input RSSE dataset size), we derive that the total space for Logarithmic-SRC when using Approach #3 from Reference [2] as the underlying SSE scheme is $O(n \log n \log m)$. Similarly, we can show that the storage of Logarithmic-SRC-i$_1$ is also $O(n \log n \log m)$, whereas that of Logarithmic-SRC-i$_2$ becomes $O(n \log^2 n)$.

Although the above is a viable adaptation of locality-aware SSE schemes to our RSSE solutions, the extra $\log n$ factor in the storage requirements of the SRC schemes translates to potentially significantly larger indices as compared to the case of utilizing locality non-aware SSE schemes instead. In the next subsection, we demonstrate that, specifically for Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i$_2$, we can still attribute locality while eliminating the extra $\log n$ factor from the space complexity.

7.2 Logarithmic-SRC-i*

We make an important observation: all our SRC schemes have *linearithmic storage* requirements, similar to the locality-aware SSE scheme of Approach #3 from Reference [2]. The question that arises is the following: *Instead of using this SSE scheme as black box, is there any way to utilize its internal mechanics (described in Section 2.2) and organize the contents of the SRC structures in a particular way that does not incur extra storage penalty?*

We first explore Logarithmic-SRC, focusing on an extreme case where all the n tuples of the dataset have the same search attribute value, say 0. Revisiting Figure 8, this means that all the

tuples are stored in node N_0 and, by the definition of the TDAG, also in nodes $N_{0,1}$, $N_{0,3}$, and $N_{0,7}$. Recall that we treat each node as a keyword. Also recall from Figure 4 that Reference [2] would store the results of all these keywords (each with size n) in $\log m$ chunks in the same array $A_{\log n}$. Therefore, the size of $A_{\log n}$ would be $O(n \log m)$. By definition, all A_i arrays in this SSE scheme must have the same size and, thus, the total size of the SSE structure becomes $O(n \log n \log m)$, i.e., the same as in the case where the SSE scheme is used as a black box.

Now we turn to Logarithmic-SRC-i$_1$, and focus initially on TDAG$_1$. Observe that each leaf node can have up to a single value (by the definition of TDAG$_1$). In addition, a node at level i can only have up to 2^i values. Moreover, a node in TDAG$_1$ is non-empty only if its subtree has some non-empty leaf, whereas the number of non-empty leaves is $O(n)$. Let us suppose that we force a non-empty node at level i to have exactly 2^i values (even if we need to randomly pad this node). Then, treating each node as a keyword for the SSE scheme, we can derive that each keyword at level i has 2^i results, and that there are $O(n)$ such keywords at any level.

We can utilize the SSE construction from Reference [2] (Figure 4) as follows. Based on the above discussion and by the definition of the SSE scheme, the results of each keyword from level i of TDAG$_1$ are stored in array A_i, randomly in $O(n/2^i)$ chunks of size $O(2^i)$ each. There are in total $\log m$ levels and, thus, $\log m$ A_i arrays. Therefore, the total storage cost is $O(n \log m)$. A similar construction and analysis applies for TDAG$_2$, which results in a total storage cost of $O(n \log n)$.

Observe that the final storage cost of the RSSE solution remains the same as in the case where the locality non-aware SSE scheme of Reference [13] was used. The reason is twofold. First, we practically *interleaved* the construction of the TDAG structures with that of the A_i arrays of the SSE scheme from Reference [2]. This enabled us to avoid inflating the storage overhead. Second, we exploited the property of each TDAG to store a *single* value per leaf node. This enabled each inner node at level i to store up to 2^i values (instead of up to n in the case of Logarithmic-SRC), thus achieving the desirable storage complexity.

The above modifications allow us to define a new *locality-aware* RSSE scheme, called *Logarithmic-SRC-i**, which is formalized as follows:

- $k \leftarrow \mathsf{Setup}(1^\lambda)$. Generate and output two SSE keys (k_1, k_2).
- $I \leftarrow \mathsf{BuildIndex}(k, D)$. Build index I_1 on the tuple ranges using TDAG$_1$ with key k_1, and index I_2 on the sorted tuples on A using TDAG$_2$ with key k_2, constructing appropriately the arrays $A_0, \ldots, A_{\log n}$ and dictionary of the locality-aware SSE Approach #3 of Reference [2] (as described above). Output (I_1, I_2).
- $t \leftarrow \mathsf{Trpdr}(k, w)$. This is an interactive algorithm. Parse $k = (k_1, k_2)$. Generate SSE token t_1 with k_1 for the SRC node on TDAG$_1$ that covers range w, and send it to the server. Decrypt the answer to retrieve new range w'. Generate SSE token t_2 with k_2 for the SRC node on TDAG$_2$ that covers w', and output (t_1, t_2).
- $X \leftarrow \mathsf{Search}(t, I)$. This is an interactive algorithm. Parse $t = (t_1, t_2)$ and $I = (I_1, I_2)$. Retrieve t_1 from the owner, invoke the Search algorithm of SSE on I_1 and send the result to the owner. Retrieve t_2 from the owner, invoke the Search algorithm of SSE on I_2 and output the result X.

Similar modifications apply to Logarithmic-SRC-i$_2$ as well. However, note that these account only for the TDAG$_2$ structure; in index I_1 constructed with the help of T$_1$, each keyword has $O(1)$ results and, thus, optimal locality and read efficiency are trivially achieved.

We include the security theorem and proof of Logarithmic-SRC-i* in Appendix A.6.

Qualitative Analysis. Logarithmic-SRC-i* achieves optimal locality and read efficiency, whereas the rest of our RSSE schemes do not. Most interestingly, it does so in a way that does not compromise

its storage cost complexity. This is in contrast to the generic approach that is applicable to all RSSE schemes that increases the storage overhead by a multiplicative factor $O(\log n)$. The leakages of Logarithmic-SRC-i* are identical with the Logarithmic-SRC-i$_1$ or Logarithmic-SRC-i$_2$.

8 TACKLING GENERIC ATTACKS

The past few years, several attacks against private range protocols have been proposed, both by the crypto and the database communities [20, 38, 44]. Dautrich et al. [20] introduce a novel attack for *precise query protocols*, i.e., protocols and schemes that leak the query result size and the access patterns. Islam et al. [38] attempt to improve the aforementioned work by using auxiliary information, while Reference [44] accelerates the performance of the same attack from $O(m^2 \log m)$ to $O(m^2) \cdot \omega(1)$, where m is the size of the domain. They achieve this without the use of auxiliary information. Also, they only assume that the database is dense and the queries are drawn from a uniform distribution. This attack is known as the *access pattern* attack. Furthermore, Kellaris et al. [44] proposes an even more powerful attack, called *communication volume* attack, that only exploits the result size of each query, formally referred to as the size of the access pattern. The communication volume attack comes with the requirement that the adversary first observes $O(m^4)$ uniformly drawn queries. Below, we focus on the state-of-the-art attacks considered in Reference [44], since they outperform the prior attacks introduced in References [20, 38].

We first explain how these attacks affect our schemes. Our Constant-BRC/URC schemes create for each range a vector of $O(\log R)$ randomly permuted tokens, and for each token they allow the server to derive the leaf-level DPRF values. Each token forms a subtree, and the entire structure of this subtree, including the order of the leaf tuples, is leaked to the server. However, the exact order relation between different tokens is not directly leaked, due to the tokens' random permutation within the vector. Note that the access pattern and the communication volume attacks cannot be used in these cases, as Constant-BRC/URC schemes do not allow overlapping queries. Moreover, recall that our Logarithmic-BRC/URC schemes are similar to the Constant ones, and even though they require increasing the storage by a logarithmic factor, they do not require the server to traverse/reconstruct the subtree defined for each token. The tuples inside a node are randomly permuted, thus preventing the server from learning order-related information about these tuples. If we disable the functionality of performing overlapping queries (as we do in the Constant schemes), then we prevent the attacks from accessing any order information. Yet, permitting overlapping queries can lead to the recovery of the entire tree structure, including the order information of the tuples in the leafs. Finally, in our Logarithmic-SRC family, the access pattern attack cannot be used for recovering any order information, since the Logarithmic-SRC schemes do not leak the access pattern itself, but only an upper-bound of each query result size. It is not clear if the communication volume attack can be used against the Logarithmic-SRC family, because the attacker cannot observe $O(m^4)$ uniformly chosen range queries; all range queries are transformed into single-keyword queries and there are only $O(m)$ possible queries that can be executed and observed by the server. Additionally, all the proposed attacks presume that the server solely observes the exact query result, while in the Logarithmic-SRC family, the results include false positives. None of the proposed attacks can be directly used against the Logarithmic-SRC family. We conjecture that a new, more specialized attack could be found to attack Logarithmic-SRC schemes, but even then the server would be unable to fully recover the order of all the tuples. Nevertheless, since our approach allows $O(m)$ queries it is possible for the server to retrieve coarse-grained order information about large groups of tuples.

A possible generic defense against these attacks that can be adopted by our Logarithmic schemes (URC/BRC and SRC) is to use fresh keys to re-encrypt the encrypted indexes after a certain number of queries, which clearly enforces the server to lose track of any order-related information on the

previous queries. The aforementioned solution creates a trade-off between the frequency of re-encryptions and the leaked information. Under certain circumstances, this solution can achieve $O(1)$ amortized query overhead, while certain de-amortization techniques [57] can be used to bound the worst case.

We will now describe a solution that can provably protect our Logarithmic-SRC/SRC-i$_1$/SRC-i$_2$/SRC-i* schemes from the above attacks. Kellaris et al. [45] propose new schemes that use ORAM to protect against access pattern attacks. However, they state that even these schemes remain vulnerable to the communication volume attack. To overcome this problem, they propose a private range scheme based on two main techniques, ORAM and differential privacy. The use of the former protects their scheme from the access pattern attack, while the latter introduces noise to the results that makes their scheme safe against the communication volume attack. It is out of the scope of our work to reproduce the results of this article, but Reference [45] approaches can be used orthogonally to our Logarithmic-SRC/SRC-i$_1$/SRC-i$_2$/SRC-i* schemes. Note that this enhancement only applies to the Logarithmic-SRC/SRC-i$_1$/SRC-i$_2$/SRC-i* schemes for two reasons. The first reason is that these schemes are not vulnerable to access pattern attacks, since the deterministic trapdoors do not leak information about the order of the results, as happens in Logarithmic-BRC/URC. The second reason lies in the database organization of these schemes, which resembles a tree-like structure. This structure has a logarithmic space expansion of the database, but bounds the communication additive overhead to be logarithmic (see Theorem 4.3 of Reference [45]). In this solution, we add a logarithmic number of noise tuples for each query result, which affects the total search time, respectively. It is important to mention that this solution cannot be efficiently applied to approaches with $O(n)$ space, since, in that case, the noise introduced by differential privacy will also have to be $O(n)$. The last point shows the superiority of our Logarithmic-SRC/SRC-i$_1$/SRC-i$_2$/SRC-i* schemes that can achieve optimal locality, while being provably secure from recently proposed attacks without requiring the use of expensive ORAM schemes. We refer the reader to the original paper [45] for more details.

9 UPDATES

Recall from Section 2.1 that most dynamic SSE (DSSE) schemes [12, 40, 41, 62] create a dynamic index that introduces the least possible leakage and provides *forward privacy*, i.e., it does not reveal that a new update satisfies a previous query. Miers and Mohassel [52] propose a DSSE scheme partially based on ORAMs; i.e., every single tuple access requires an extra poly-logarithmic overhead for accessing more tuples. Furthermore, Bost [10] presents a scheme that requires $O(m \log N)$ available storage, where m denotes the size of the domain, and Stefanov et al. [62] use ideas mainly inspired by ORAMs and expensive cryptographic tools, such as oblivious sorting. We did not use the recent works of References [10, 52, 62], because these do not seem to meet our efficiency requirements.

We follow an alternative methodology adopting a *bulk-loading* technique from commercial databases (e.g., Vertica [48]), which is simple from a security point of view, but (i) builds upon static SSE schemes that are faster and easier to implement than their dynamic counterparts, (ii) enables easy formulation of leakage, and (iii) captures forward privacy. It is also *generic*; it applies to *all* our solutions and to *any* future static RSSE scheme. We need to highlight that our dynamic solution has the exact same leakage profile as the aforementioned dynamic SSE schemes; categorized as a dynamic scheme with state-of-the-art leakage for updates.

We assume that updates come in batches, which need not be of the same size, and each batch i is treated as a separate dataset D_i. The updates can be insertions of new tuples, or modifications/deletions of old tuples. Every update is treated as an insertion in the new dataset; deletions carry a one-bit flag indicating that the tuple must be removed. For each new dataset D_i, the owner creates

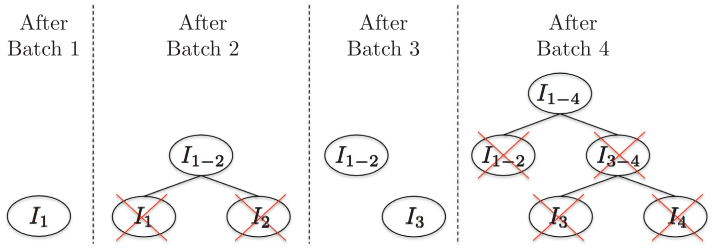


Fig. 11. Demonstrating the update process for $s = 2$. I_{X-Y} denotes the index corresponding to all batches from X to Y. When the second update batch arrives, the owner downloads I_1, decrypts it, merges it with the updates, constructs I_{1-2}, and uploads it to the server. I_2 is never created (included in the figure only for demonstration purposes). Similarly, at the fourth batch, I_{1-2} and I_3 are downloaded and then index I_{1-4} is computed and sent to the server. The indexes I_4 and I_{3-4} are never created.

a new index I_i (note that I_i may correspond to more than one internal indexes that the SSE scheme creates, as in the case of Logarithmic-SRC-i$_2$, where each I_i contains two indexes) with a *fresh* key k_i following the BuildIndex of the utilized construction, and sends the encrypted data to the server. Recall that both inserted and deleted tuples may exist in each index I_i.

Suppose that the owner has uploaded b such indexes. Upon a query, the owner creates b separate tokens, one for each index, with the corresponding key. The server processes them separately on the b indexes, and returns the results. The final refinement of the results occurs at the owner, who filters out the deleted tuples and appropriately performs the potential modifications.

Clearly, the number of keys, query size, storage cost, search time, and result size increase with b (linearly, if the batches are of about the same size) and, thus, the number of indexes should not increase indefinitely. Therefore, we adopt the approach of Vertica, which essentially organizes the datasets/batches into a *log-structured merge tree*. Specifically, the owner sets a parameter called *consolidation step*, denoted by s, which determines how frequently the indexes must be merged. After creating s new indexes, the owner downloads them, merges their tuples into a single index, consolidating deleted tuples with their inserted counterparts (i.e., an inserted tuple is removed if we meet an identical tuple that has its delete flag set), re-encrypts the index, and sends it to the server. This happens *hierarchically*; after consolidating s indexes s times, the s merged indexes are further consolidated into a new one. An example for $s = 2$ is depicted in Figure 11. Conceptually, this process is like organizing the indexes as leaves of a full s-ary tree created bottom-up, such that when s nodes are created in a level, they get consolidated creating a parent node at the next higher level. This leads to an *amortized logarithmic* merge cost in the number of batches [48]. Although this incurs extra periodic cost at the owner for the consolidation and re-encryption, it retains $O(s \log_s b)$ indexes at the server (and keys at the owner) at all times, instead of b. Finally, note that s should be tuned based on the application at hand. For instance, if the application expects frequent deletions, it is beneficial to set s to a small value, to perform merge operations more frequently, thus eliminating the extra cost for storing the deletions as insertions.

The leakage of this methodology is essentially the entire history of the $\mathcal{L}_1, \mathcal{L}_2$ leakages of every index that was once "active" at the server. More formally, at the time of processing batch i, the number of independent indexes is c (c is upper-bounded by $O(s \log_s b)$), and the total leakage of our dynamic scheme is defined as the union of all $\mathcal{HIST}^\kappa$, for $1 \leq \kappa \leq i$, where $\mathcal{HIST}$ is defined as the union of leakages $(\mathcal{L}_1, \mathcal{L}_2)^1, \ldots, (\mathcal{L}_1, \mathcal{L}_2)^c$. For instance, from this leakage, one could derive the number of deletions that occurred in one of the batches. Also observe that our technique satisfies forward privacy; every index is encrypted with a fresh key and, thus, a token

Prefix sums P

0	1	2	3	4	5	6	7
0	0	45	45	45	45	65	65

I

keyword	value
0	0
1	0
2	45
3	45
4	45
5	45
6	65
7	65

Fig. 12. Constructing a secure index I for answering RSQ.

created for one index in the past cannot be used to decrypt index components produced in the future.

10 RANGE AGGREGATE QUERIES

In this section, we focus on range aggregate queries (defined in Section 3). Observe that there is a straightforward way to answer this type of query using our Constant-BRC/URC and Logarithmic-BRC/URC RSSE schemes. For instance, for the RSQ, we can simply store the sum of all tuples corresponding to each node of the tree structures of each scheme, instead of the tuples themselves. Then, instead of sending the tuples associated with the nodes/keywords corresponding to the query range, the server sends only the (encrypted) sums stored in those nodes. The owner can then simply add all results, since each result corresponds to a single subtree covering only part of the range. However, in addition to the inherited leakage of the underlying RSSE schemes, potentially unnecessary storage is sacrificed, whereas the query size is non-constant. On the other hand, Logarithmic-SRC and its variants cannot be used at all, since the false positives alter the actual result. Calculating an exact result involves sending all the tuples stored in the queried node as in the original SRC schemes, and forcing the owner to *manually* filter the false positives and calculate the aggregate locally. This clearly incurs considerable cost to both the server and the owner.

Our goal is to design secure schemes that can answer range aggregate queries with *constant query and result size*, while retaining the storage overhead as low as possible. This motivates new constructions for this problem. In Section 10.1, we describe our RSQ scheme (along with its adaptations for answering also count, average, and variance queries), whereas in Section 10.2 we present our RMQ technique (along with its adaptations for solving also max, top-k, and bottom-k queries).

10.1 Range Sum Query (RSQ)

We introduce a novel, provably secure scheme for the RSQ based on *prefix sums* (see Section 2.2). Figure 12 shows the prefix sums array P and secure index I constructed in this scheme. We essentially create a keyword for every domain value i, and store as result the corresponding value of

$P[i]$. Let $[3, 7]$ be the query range. Then the owner sends tokens for keywords 2 and 7 to the server, receives and decrypts results $P[2]$ and $P[7]$, and computes the result as $P[7] - P[2] = 65 - 45 = 20$.

This technique is formalized as follows:

$k \leftarrow \mathsf{Setup}(1^\lambda)$. Generate and output a SSE key k.
$I \leftarrow \mathsf{BuildIndex}(k, D)$. Create the prefix sums P of the dataset on the aggregate attribute B for the domain of attribute A. Build SSE index I using as keywords each domain value i and corresponding result value $P[i]$. Output I.
$t \leftarrow \mathsf{Trpdr}(k, w)$. Parse w as query range $[l, u]$. Generate SSE tokens for keywords $l - 1$ and u and output them as $t = (t_1, t_2)$.
$X \leftarrow \mathsf{Search}(t, I)$. Parse $t = (t_1, t_2)$ and invoke the Search algorithm of SSE on I for t_1 and t_2. Output the two results as X.

The owner decrypts the result X to get $P[l - 1]$ and $P[u]$, and calculates the final aggregate result as $P[u] - P[l - 1]$. The scheme requires $O(m)$ space, where m is the domain size, and $O(1)$ query size and search time. As leakage, it reveals whether two ranges share an endpoint.

We describe the leakage of the RSQ scheme more formally:

— $\mathcal{L}_1(D, A) = \langle m \rangle$
D is the dataset, A denotes the query attribute domain and m is the size of A.
— $\mathcal{L}_2(D, A, W) = \langle \alpha(W), \sigma(W), (\mu(w_L), \mu(w_R)) \rangle$
$\alpha(W), \sigma(W)\rangle$ are the access and search patterns of the queries as defined for SSE schemes. For every range query $w \in W$, the leakage contains a tuple that consists of an alias $\mu(w_L)$, an alias $\mu(w_R)$ where $\mu(w_L)$ refers to a $l - 1$ correspondence and $\mu(w_R)$ to u (of the range $[l, u]$).

We include the security theorem and proof of RSQ in Appendix A.7.

With minor modifications of the above scheme, we can answer count, average, and variance queries. For instance, to support count queries, we construct the prefix sums by considering that every tuple has B attribute value equal to 1. For average, we construct prefix sums for both counts and sums and return both values per query token. The owner can then retrieve the exact sum and count separately, and then divide these two values to derive the average. Finally, variance requires constructing prefix sums for counts, sums, and sums of squares, such that we can derive the average and the average of squares. The exact variance is easily computed by subtracting the square of the average value from the average of squares. In all the above cases, the storage, query size, and search time are all expanded by a small constant factor.

Finally, note that we can easily handle updates (insertions and deletions), following the update methodology discussed in Section 9. In particular, we perform updates in batches, creating a *new* secure index as described above for each batch. In the case of deletions, we make sure we *reverse the sign* of the aggregate attribute value for each deleted tuple (each deleted tuple with an aggregate attribute value of X is thus treated as an inserted tuple with an aggregate attribute value of $-X$) when creating the prefix sums. The owner queries each index *independently* as presented above deriving a separate result, and simply adds all these results together. This increases the query and search time by a factor equal to the number of separate indices stored at the server (which, as mentioned in Section 9, can be compacted periodically if their number exceeds some threshold).

10.2 Range Min Query (RMQ)

In this subsection, we design two novel and provably secure RMQ schemes that feature constant query size and search time. The first requires $O(m \log m)$ space, whereas the second $O(m + n \log n)$

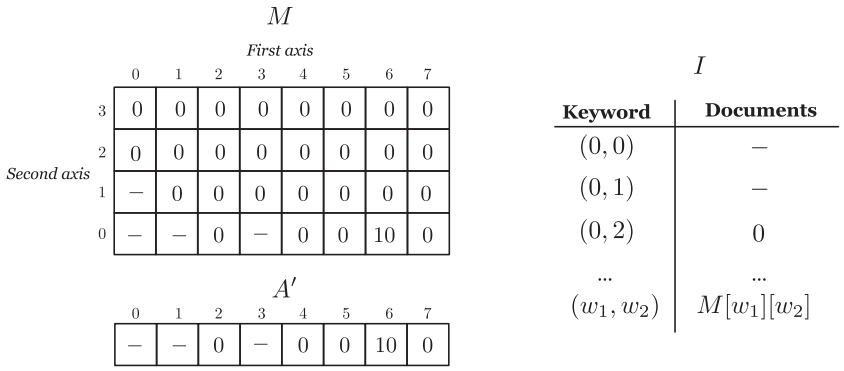


Fig. 13. Example of secure RMQ Approach #1.

space. They are both based on the sparse table approach for solving the traditional RMQ query described in Section 2.2. Their main difference is that the first applies the sparse table technique on the range attribute domain (and, hence, the resulting complexity), whereas the second maps the range query process from the range attribute (A) domain to the *id attribute domain*, involving an extra look-up to retrieve auxiliary information about the range endpoints. The second scheme is preferable in the presence of great *skew* in the distribution of the A values, i.e., when there are very few distinct A values. However, the space savings come with the cost of an extra round of communication between the owner and the server. In other words, in contrast to the first approach, the second is *interactive*.

Approach #1: ($\mathbf{O(m}$ *log* $\mathbf{m)}$ *space).* Considering the sparse table solution to the traditional RMQ problem described in Section 2.2, we can construct a secure RMQ scheme with the targeted space bounds in a simple manner. Specifically, the owner first constructs a vector A' on range attribute A storing at element $A[i]$ the *minimum* aggregate attribute (B) value for each tuple with range attribute value (A) equal to i. Figure 13 shows an example of this vector for the dataset of Figure 7. For instance, there are 10 tuples with $A = 2$, but the minimum value is 0, thus $A'[2] = 0$. On the other hand, there is no tuple with $A = 0$ and, thus, we store "$-$" in $A'[0]$ instead. The size of vector A' is m, i.e., equal to the size of domain A.

Next, the owner constructs the M matrix on A' in the same manner as we described in Section 2.2. Note that, by definition, M stores $\log m$ values for *every* element of A', even for those that store $-$. The secure index I then consists of $m \log m$ keywords of the form $w = (w_1, w_2)$, each associated with a document that stores $M[w_1][w_2]$. As in all our previous solutions, the secure index is constructed with some underlying SSE scheme. Observe that M contains $m \log m$ elements and, thus, the total space complexity of this scheme is $O(m \log m)$.

Given a query range of the form $[i, j]$, the owner computes a token for keyword (i, l) and one for keyword $(j - 2^l + 1, l)$, where $l = \lfloor \log(j - i + 1) \rfloor$ and sends them to the server. Using the secure index I, the server can retrieve and return as result the encrypted values of $M[i][l]$ and $M[j - 2^l + 1][l]$. According to the discussion in Section 2.2, the owner can calculate the final results as $\min(M[i][l], M[j - 2^l + 1][l])$. In Figure 13, for query range $[3, 7]$, the server sends $M[3][2] = 0$ and $M[4][2] = 0$, and the owner computes the final result as 0. Note that, if $M[w_1][w_2] = -$, then this is ignored, whereas if both returned M entries are equal to $-$, then the owner knows that there are no tuples in the query range. Observe that the query size and search time are constant.

We omit the detailed construction and proof of this approach, as they are simpler versions of the ones we provide for our second secure RMQ scheme, presented below.

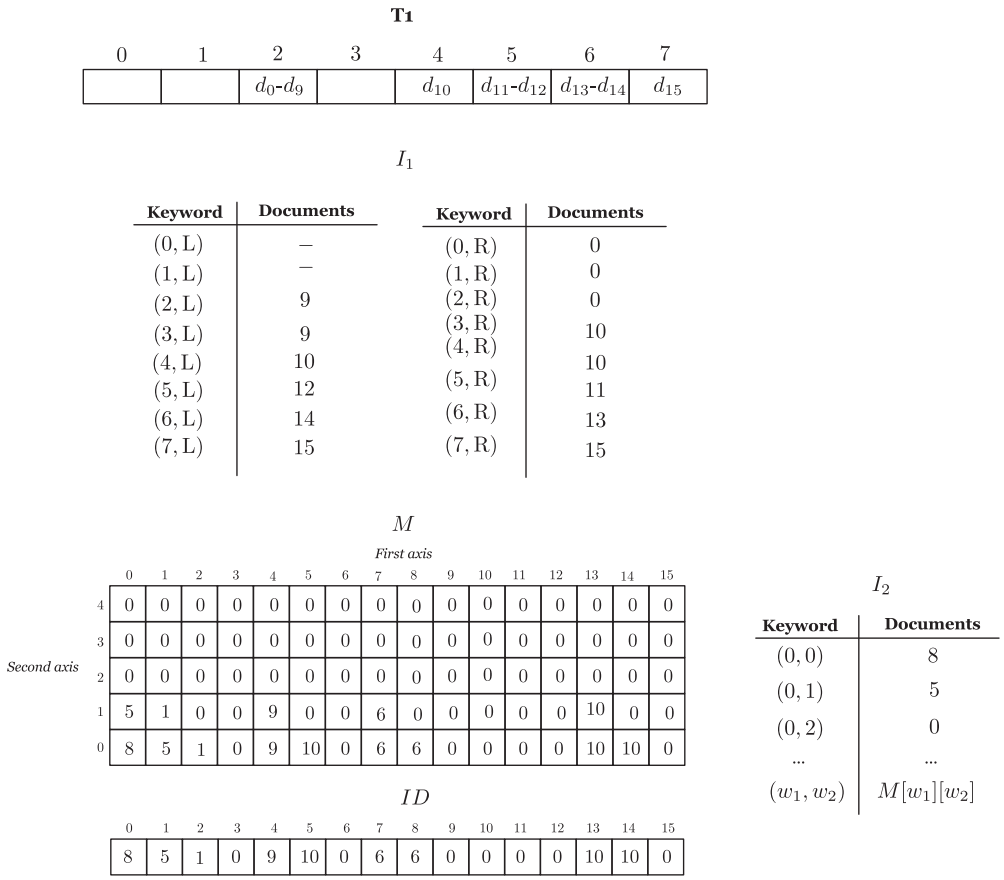


Fig. 14. Example of secure RMQ Approach #2.

Approach #2: *(***O(m + n** *log* **n)** *space)*. Observe that the first approach wastes a lot of space when there is great skew in the distribution of the range attribute (A) values. This is because vector A' ends up storing many redundant $-$ values, which in turn may appear in a considerable number of elements in M. Ideally, we would like the vector on which M is computed to be *dense*, i.e., to have no $-$ (i.e., empty) values, such that M becomes dense as well. The main idea of our second approach is to map the range search from the range attribute (A) domain to the ID attribute domain. We discuss the construction in detail below, using the example in Figure 14 for the dataset of Figure 7.

The owner first constructs an n-element vector ID as follows. It *sorts* the tuples based on the range attribute A, and stores at $ID[i]$ the aggregated attribute B of the i-th tuple in the sorted order. For simplicity, in the dataset of Figure 7 the tuples are already sorted on A and, thus, the i-th tuple has id i (we stress though that this may not be the case in general). Please note that, in our example, the smallest range attribute value is 2, thus the 10 tuples with this range attribute value are mapped to the first 10 entries of vector ID, while at $ID[10]$ we store the aggregate attribute of the next tuple in the sorted order (i.e., the tuple having a range attribute equal to 4). The owner then builds the M matrix on ID as we discussed in Section 2.2 (observe that M now has five levels and that its lowest level is identical to the ID vector). Similar to our first RMQ approach, the owner builds a secure

index on keywords (w_1, w_2) that are associated with a document that stores $M[w_1][w_2]$. We call this index I_2 in the second approach.

Given a range on A, for example, $[3, 7]$, observe that there is no way to utilize M, since the latter is constructed on the tuple IDs. However, recall that the tuples were *sorted* on A when we constructed ID. Therefore, if we can locate the *first* and *last ids* of the tuples that satisfy the range on A, we can *map* the range on A to a range on ID and, thus, utilize M to answer it. Towards this end, we construct a second secure index I_1 as follows. Recall that we constructed an index for locating the first and last tuple ID in a query range on A in Logarithmic-SRC-i$_2$ (Section 6.4), namely I_1 on a simple vector T_1. We build such an index in the same manner, which is shown in the upper part of Figure 14. Note that it is important for T_1 to store the tuple IDs *after sorting* the tuples on A.

The query algorithm is *interactive*, which we illustrate with an example for query range $[3, 7]$ in Figure 14. The owner first issues keywords $(3, \mathrm{R})$ and $(7, \mathrm{L})$ on I_1, and receives tuple IDs 10 and 15, respectively. Using these IDs, she forms a new query range $[10, 15]$ for vector ID. She then creates the appropriate tokens for the corresponding M matrix that will be used on secure index I_2. Based on previous discussions, these will be $(10, 2)$ and $(11, 2)$ to get $M[10][2] = 0$ and $M[11][2] = 0$, respectively. The owner finally derives that the result is 0. The formal construction is provided below:

$k \leftarrow \mathsf{Setup}(1^\lambda)$. Generate and output two SSE keys (k_1, k_2).

$I \leftarrow \mathsf{BuildIndex}(k, D)$. Build T_1, ID and M as explained in Figure 14. Construct SSE index I_1 using T_1 with key k_1, and SSE index I_2 on M using key k_2. Output (I_1, I_2).

$t \leftarrow \mathsf{Trpdr}(k, w)$. This is an interactive algorithm. Parse $k = (k_1, k_2)$ and range $w = [i, j]$. Generate SSE token $t_1 = (t_{11}, t_{12})$ with k_1 for keywords (i, R) and (j, L). Decrypt the answer to retrieve new range $[i', j']$ for ID. Compute $l = \lfloor \log(j' - i' + 1) \rfloor$. Generate SSE token $t_2 = ((i', l), (j' - 2^l + 1, l))$ with k_2 for I_2, and output (t_1, t_2).

$X \leftarrow \mathsf{Search}(t, I)$. This is an interactive algorithm. Parse $t = (t_1, t_2)$, $t_1 = (t_{11}, t_{12})$, $t_2 = (t_{21}, t_{22})$ and $I = (I_1, I_2)$. Retrieve t_1 from the owner, invoke the Search algorithm of SSE for t_{11}, t_{12} on I_1 and send the results to the owner. Retrieve t_2 from the owner, invoke the Search algorithm of SSE t_{21}, t_{22} on I_2 and output the result X.

Observe that the second approach needs $O(m)$ space for I_1 and $O(n \log n)$ space for I_2, i.e., the total space requirements are $O(m + n \log n)$. Similar to the first approach, the query size and search time are constant.

The leakage is the union of the SSE leakages for the two keyword queries on I_1 and the two keyword queries on I_2. Practically, the adversary learns if two queries have same starting range endpoint or ending range endpoint (from I_1), and whether two queries are answered by the same element of M (from I_2). The latter leakage reveals that the two ranges may have similar sizes, since the same power of two range sizes helped in answering both the queries.

We describe the leakage of RMQ-Approach #2 more formally:

— $\mathcal{L}_1(D, A) = \langle m, n \rangle$
D is the dataset, A is the query attribute domain, n is the cardinality of D and m is the size of A.

— $\mathcal{L}_2(D, A, W) = \langle \alpha(W), \sigma(W), (\mu(w_{1,L}), \mu(w_{1,R}), (\mu(w_{2,L}), \mu(w_{2,R})) \rangle$
$\alpha(W), \sigma(W)\rangle$ are the access and search patterns of the queries as defined for SSE schemes. For every query range $w \in W$, the leakage contains a tuple $(\mu(w_{1,L}), \mu(w_{1,R})$ of aliases for the two tokens of I_1 and a tuple $(\mu(w_{2,L}), \mu(w_{2,R})$ of aliases for the two tokens of I_2. As in

Logarithmic-SRC-i$_1$, ($\mu(w_{1,L})$, $\mu(w_{1,R})$), ($\mu(w_{2,L})$, $\mu(w_{2,R})$ and $\sigma(W)$ correspond to the leaked relationships between these tokens.

We omit the proof of this approach, as it is similar with the one of Logarithmic-SRC-i$_2$ (it has four different tokens and for each of them we use $\sigma(W)$ to identify if a token has appeared before or we will choose a new random token).

Again, the random values are of appropriate length and format. Now, the simulator uses $\mathcal{L}_2$ to simulate the queries. For simplicity, we omit all the combinations between the four different tokens; the high level idea is similar to the one of Logarithmic-SRC-i$_1$. In particular, we use $\sigma(W)$ to identify if a token (one of the four above) has appeared before. This allows us to specify if the simulator will return a previous token or a new one.

With simple modifications, we can support more aggregate queries in a similar way. For instance, we can support *range max* by simply recording the *maximum* value on B instead of the minimum in all the discussed structures. In addition, we can handle slightly more complicated queries like *range bottom-k* and *range top-k*, which return the k minimum and the k maximum values (with multiplicities) in the specified range, respectively. Range min (max) is then a special case of range bottom-k (top-k) for $k = 1$. Below we describe the modifications only for bottom-k, since the case of top-k is very similar.

We can construct a secure scheme for range bottom-k queries by expanding the storage space and result size of the underlying RMQ scheme by a factor of k. We build upon our second RMQ approach and, thus, the total storage space required is $O(m + kn \log n)$. When constructing the secure index, the only modification required is on the preparation of M and the eventual construction of I_2. Instead of storing only the minimum value in each element of M, we store the k smallest values in the corresponding range. In addition, for each such value, we explicitly store the *id* of the tuple that contains it. In other words, we store k pairs $(d.B, d.id)$ in each element of M (if the number of tuples is less than k in some ranges, then we store – instead). Index I_2 then stores these k pairs as the contents of the documents for each keyword (w_1, w_2) (with the sort order being not solely on $d.B$ but on the entire $(d.B, d.id)$ pair—this is required to ensure correctness in the case of deletions), instead of just the minimum as in the RMQ scheme. Given a query, the owner follows the same algorithm as we discussed for RMQ, but receives $2k$ $(d.B, d.id)$ pairs in the second round of communication. Observe that the owner can simply *sort* these pairs in ascending order of $(d.B, d.id)$, *eliminate all duplicate pairs*, and choose the first k pairs as the bottom-k result. The query size is constant, but the result size becomes $O(k)$.

Finally, we are also able to answer the RMQ query and its variations in the presence of updates as follows. We focus again on bottom-k (which also covers range min), since top-k is similar. The key idea in the solution is the fact that we include the *ids* of the bottom-k tuples in addition to their values on aggregate attribute B in matrix M (this implies that we should include them also when $k = 1$ for range min). In the case of insertions, recall that we need to create independent structures for every update batch and query each separately. After getting the results from each batch, we just need to sort the *union* of the received pairs on B, remove the duplicates, and select the first k as the bottom-k result, exactly as we explained above.

Deletions require a more subtle treatment. First, we need to store deletions separately from insertions, but the secure structures we build on them are identical to those of insertions discussed above. Second, we query all structures independently as explained above, but maintain *two* sorted and deduplicated lists; one for insertions and one for deletions, let L_I and L_D, respectively. Finally, we perform the *set difference* $L_I - L_D$, and return the first k pairs in the resulting list as the final result. The problem that arises is that, if a deleted tuple used to be a bottom-k result for the query range, we may end up with fewer than k elements in $L_I - L_D$. In that case, we need to consolidate

the update batches and construct new indices. A heuristic solution to this problem is to keep $k' > k$ pairs in matrix M, slightly increasing all the involved costs. This reduces the chance that a set of deletions causes the size of $L_I - L_D$ to drop below k.

11 EXPERIMENTAL EVALUATION

The performance of the proposed schemes was also evaluated experimentally using real datasets. From the experiments, we excluded the Quadratic scheme, which features a prohibitive storage cost. Logarithmic-SRC-i* was implemented using Logarithmic-SRC-i_1 as the base scheme. Based on our discussion in Section 2.1 about the three schemes of Li et al. [50], we also include a comparison to the basic scheme of Li et al., hereafter referred to as PB, *recalling though that the latter offers weaker security than our schemes.* Our experiments were set to evaluate the performance and scalability of the proposed algorithms for range search queries and range aggregate queries. More experiments, which focus on the performance of the algorithms in presence of updates and on the query execution cost for the owner are included in the Appendix.

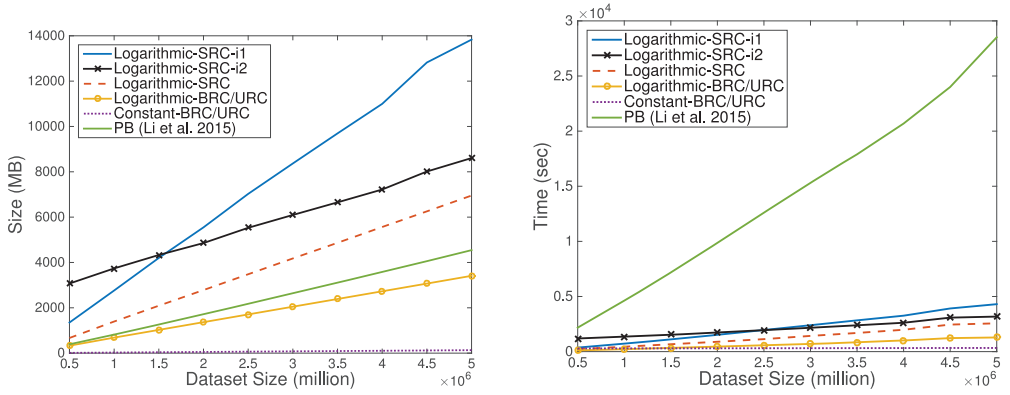
Setup. We implemented our solutions in Java, and conducted our experiments on a 64-bit machine with an Intel® Core™ i7-2720QM CPU at 2.2GHz and 16GB RAM, running Linux Ubuntu 14.10. We utilized the JavaX.crypto library for the entailed cryptographic operations. Specifically, we implemented PRF and GGM evaluations with HMAC-SHA-512, and hash computations with SHA-1. We also used AES128-CBC for encryption. We chose the construction by Cash et al. [13] as our underlying SSE scheme, setting its parameters to the values recommended in [13] for space-efficiency ($S = 6000$, $K = 1.1$).

We experimented with two real datasets. The first is from the Gowalla geo-social network (`snap.stanford.edu/data/loc-gowalla.html`), hereafter called Gowalla. This dataset consists of 6,442,890 user location check-ins in a period between February 2009 and October 2010 and it has size 34.3MB. We used as query attribute the date/time of the check-ins converted to 32-bit integers and translated such that the domain is $A = \{0, \ldots, 103017913\}$. The second dataset is from the U.S. Postal Service (`www.app.com`), called USPS, and contains 389,032 employee records and it has size 389MB. We used as query attribute the annual salary field with domain $A = \{0, \ldots, 276840\}$. Note that Gowalla is relatively uniform on A (95% distinct values on A), whereas USPS is heavily skewed (5% distinct values on A). All datasets and respective indexes fit in memory.

We first compare our proposed algorithms and experimentally evaluate them based on their index construction cost, the number of false positives, and the search time.

Index Costs. In the first set of our experiments, we assess the index size and construction time in Gowalla when varying the dataset size n, and demonstrate the results in Figures 15(a) and (b), respectively. The construction time involves also the I/O cost for reading the dataset from the disk into main memory. As in Reference [50], to retrieve the various datasets, we simply partition the initial dataset (sorted on A) into 10 sets of 500K tuples each, chosen uniformly at random from the entire dataset, start with one partition, and gradually add the rest of the partitions. Note that the BRC and URC variants of the same scheme feature identical costs. The index size entails only the replicated tuple IDs and their associated keywords. The curves in both figures have the same trends, since the index size dictates the construction time. Moreover, both the index size and construction time scale linearly with n, since even the logarithmic factors essentially add a constant factor to the overall size.

As expected, the Constant schemes achieve the smallest index size (12.63-131MB), as well as the lowest construction time (288–332s). The costs in Logarithmic-BRC/URC increase faster due to the logarithmic factor in the index size. Logarithmic-SRC incurs about twice the size and time compared to Logarithmic-BRC/URC, due to the nodes injected to form the TDAG. Logarithmic-SRC-i_1



(a) Index size (b) Construction time

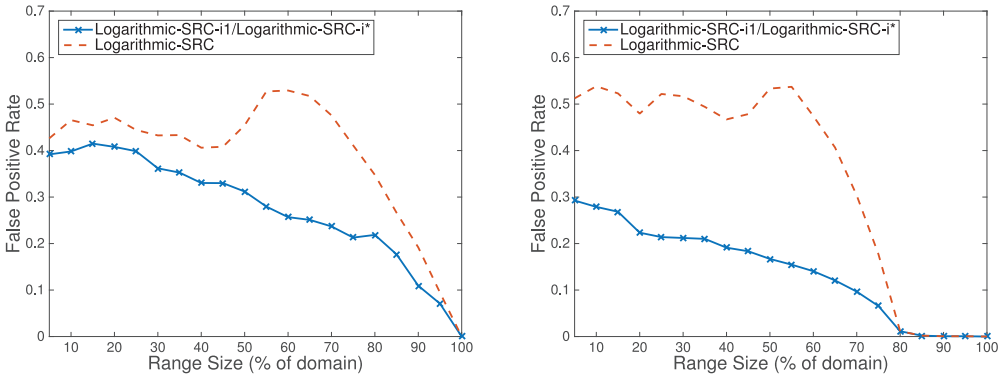
Fig. 15. Index costs (Gowalla).

Table 3. Index Costs (USPS)

Scheme	Index size (MB)	Constr. time (s)
Constant-BRC/URC	10.30	2.853
Logarithmic-BRC/URC	195.7	54.967
Logarithmic-SRC	391.4	106.970
Logarithmic-SRC-i$_1$	419.14	119.662
Logarithmic-SRC-i$_2$	398.69	112.32
PB [50]	299.06	2374

requires double the size and time compared to Logarithmic-SRC. Recall that Logarithmic-SRC-i$_1$ entails building a second index on top of the one in Logarithmic-SRC, whose size depends on the distinct values on A covered by the dataset. Since 95% of tuples in Gowalla have distinct keys, the size of the extra index is almost as large as the basic one, doubling the overall space requirements. Logarithmic-SRC-i$_2$, on the other hand, is initially more expensive than Logarithmic-SRC-i$_1$, but it becomes more compact after the dataset outgrows 1.5 million tuples. This coincides with the theoretical cost analysis of the two methods. Notice that, in this dataset, domain size m is a bit over 100 million. In other examples, domain sizes may easily stretch to several billions, e.g., the full range of 32-bit integer keys will already cover domain sizes over 4.2 billion, whereas IPv6 ip addresses will have $m = 2^{128}$. In these cases, Logarithmic-SRC-i$_1$ will outperform Logarithmic-SRC-i$_2$ for much larger datasets. Logarithmic-SRC-i* has similar storage demands with Logarithmic-SRC-i$_1$, but since it uses padding such that all result sizes are a power of 2, its overall size is at most twice the size of Logarithmic-SRC-i$_1$. Finally, observe that Constant-BRC/URC and Logarithmic-BRC/URC outperform PB [50] in terms of index size, whereas the construction cost of all our schemes is significantly lower than PB.

Table 3 includes the index size and construction time for the full USPS dataset. Similar to the case of Gowalla, the Constant schemes feature the smallest overheads, Logarithmic-BRC/URC add a constant factor to the costs of Constant, and Logarithmic-SRC incurs almost twice as high costs as Logarithmic-BRC/URC. However, the most interesting observation here is that, contrary to the case of Gowalla, Logarithmic-SRC-i$_1$ adds minimal overheads to those of Logarithmic-SRC.



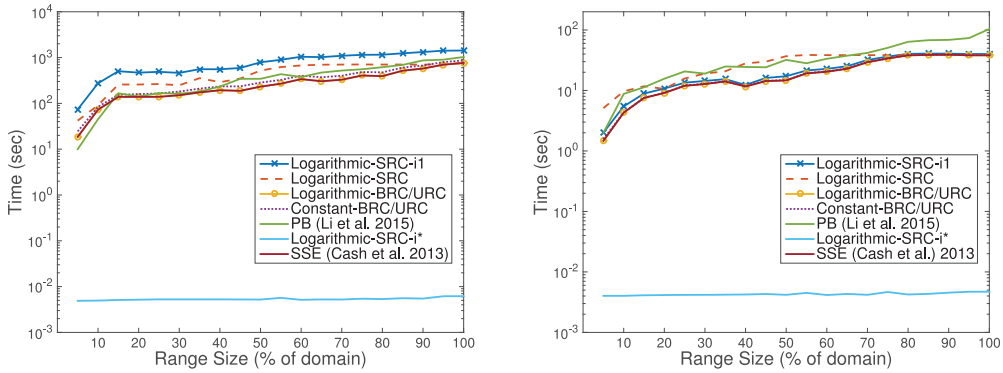
(a) Gowalla (b) USPS

Fig. 16. False positives.

This is because in USPS there are only 5% distinct values on A, which makes the extra index in Logarithmic-SRC-i$_1$ very compact. Once again, Constant-BRC/URC and Logarithmic-BRC/URC feature a smaller index size than PB, whereas all our schemes have orders of magnitude faster construction time than PB.

False Positives. Figure 16 plots the average false positive rate (i.e., the average ratio of false positives over the total result size) as a function of the query range size, computed over the results of 200K random queries on each domain. The Constant and the Logarithmic-BRC/URC schemes are not included in this experiment since they do not induce false positives. Since Logarithmic-SRC-i* was built using Logarithmic-SRC-i$_1$ as the reference scheme, both Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i* have almost identical false positives, and are therefore displayed with a single curve. We see that the false positive rate decreases almost linearly with the range size, since more tuples previously marked as false positives are contained in the range. In both datasets, Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i* incur fewer false positives, outperforming Logarithmic-SRC by up to 27% in Gowalla and 38% in USPS. In USPS, the rate drops more steeply in Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i*, and the performance margin becomes wider, since USPS is more skewed than Gowalla, offering stronger opportunities to the auxiliary I_1 index in these two methods to eliminate false positives. Overall, the false positives in Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i* do not exceed 40% of the entire answer. Note that PB introduces a very small number of false positives for all range sizes. However, our Constant-BRC/URC and Logarithmic-BRC/URC, whose performance has dominated that of PB in the investigated metrics so far, introduce no false positives at all.

Search Cost. We evaluate the wall-clock time for executing Search at the server. In Figures 17(a) and (b), we report the average CPU cost in Gowalla and USPS, respectively, for the same 200K queries used in the previous experiment. Notice that the Y axis is in logarithmic scale for illustration purposes, and the displayed time is inclusive of the unavoidable time of retrieving the actual results through the underlying SSE scheme. The search time in all our methods, except Logarithmic-SRC-i*, is dominated by the PRF/DPRF evaluations entailed in Reference [13] for each retrieved tuple. Consequently, Logarithmic-BRC (resp. Constant-BRC) and Logarithmic-URC (resp. Constant-URC) have negligible performance difference and are grouped together. Finally, as a worst-case scenario for the absolute running time benefits of Logarithmic-SRC-i* compared to all other schemes, the queries were performed on in-memory



(a) Gowalla (b) USPS

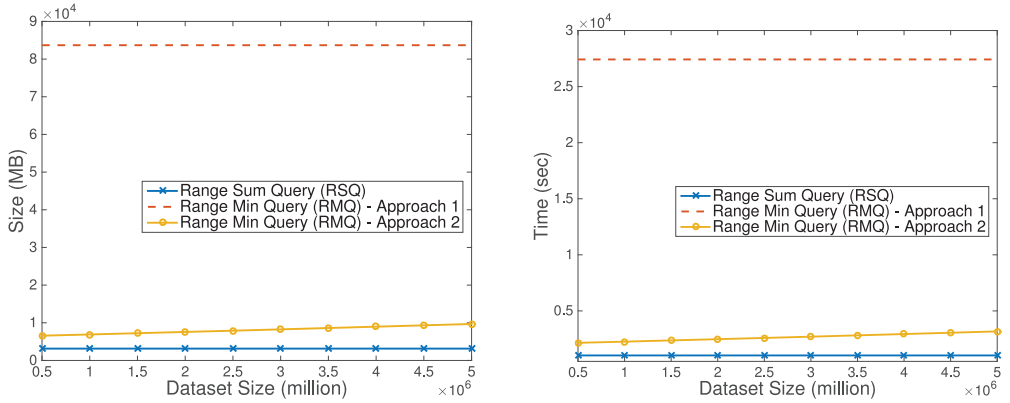
Fig. 17. Search time.

data. Please recall that Logarithmic-SRC-i* is the only scheme that optimizes the read efficiency, thus, for disk resident data, the absolute running time difference is expected to be even larger.

Notice that, choosing a single-keyword SSE protocol other than Reference [13] as a basis of our methods could entail different costs for our methods, since these exploit SSE as a black box (except Logarithmic-SRC-i*), and thus depend on its performance. In other words, a more efficient SSE scheme will immediately improve the performance of our methods. However, evaluating and comparing the effect of single-keyword SSE schemes is out of the focus of this work.

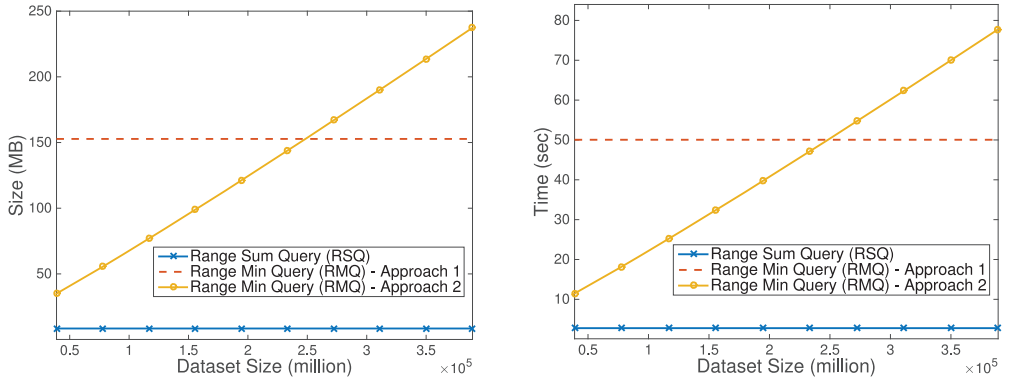
Clearly, Logarithmic-SRC-i* outperforms all other algorithms by four orders of magnitude (its worst-case time is around 5 milliseconds). The results are consistent in both datasets. This stark performance improvement compared to our other algorithms is attributed to the optimal locality and read efficiency of Logarithmic-SRC-i*, which enables it to execute a query with a single PRF evaluation, as well as a single and contiguous memory lookup (recall that PRF/DPRF evaluations constitute the dominant time bottleneck for all our other methods). Also, performance of Logarithmic-SRC-i* stays almost unaffected by an increase in the query range, since additional sequential reads add negligible overhead. We also see that the performance of the Logarithmic-BRC/URC methods coincides with that of pure SSE. This is expected, since the search complexity of the Logarithmic-BRC/URC is $O(\log R + r)$, and $\log R$ adds negligible overhead. The Constant schemes are slightly more expensive, due to the extra expansion of the GGM tokens into $O(R)$ DPRFs. Observe that this additional cost is more pronounced in Gowalla, due to its significantly larger domain (and, thus, query range sizes) than USPS. The SRC-based schemes are more costly, due to the false positives they introduce. In Gowalla, Logarithmic-SRC-i_1 is more expensive than Logarithmic-SRC, due to the extra searches in its auxiliary index. Nevertheless, in USPS, Logarithmic-SRC-i_1 outperforms Logarithmic-SRC, since its savings in false positives outweigh the extra index cost. PB features comparable search cost with that of Constant-BRC/URC and Logarithmic-BRC/URC in the Gowalla dataset, but higher search cost in the case of USPS. In overall, the Constant-BRC/URC and Logarithmic-BRC/URC schemes subsume PB also in terms of performance.

Range Aggregate Queries. We examine the performance of our techniques for aggregate queries on datasets of different sizes. The datasets were constructed in a similar way to the above experiments, by selecting subsets of the Gowalla and USPS datasets.



(a) Index size (b) Indexing time

Fig. 18. Index costs—Range Aggregate Queries (Gowalla).



(a) Index size (b) Indexing time

Fig. 19. Index costs—Range Aggregate Queries (USPS).

We first consider the index size for the three algorithms. As shown in Figures 18(a) and 19(a), the index size for the RSQ and the first RMQ algorithm (RMQ Approach #1) is orthogonal to the dataset size; this coincides with the theoretical results discussed in Section 10, which show that index size for these algorithms depends only on the domain size m. On the contrary, for the case of the second RMQ algorithm, the index size for Gowalla grows just a little bit more than linearly with the dataset size, substantially outperforming the former approach. This happens because the second RMQ approach manages to exploit the high sparsity of the Gowalla dataset (n/m is below 0.05) to reduce space requirements. On the other hand, for the USPS dataset, we see that the second RMQ algorithm soon becomes more expensive than the former, since the dataset is denser (n/m ends up around 1.4 for the full dataset).

In terms of indexing time, we see again in Figures 18(b) and 19(b) that the performance of the RSQ algorithm and the first RMQ algorithm (RMQ Approach #1) is almost orthogonal to the dataset size. As before, there exists a miniscule, not visually detectable increase in the construction times of all indexes due to the initial preparatory steps of the two algorithms (i.e., constructing the prefix sum for RSM or mapping of values for RMQ—cf. Figures 12 and 13), but this is overshadowed by

Table 4. Search Times for Aggregate Queries. Difference was Insignificant between the Two Datasets

Query	Time (msec)
Range Sum Query	0.032
Range Min Query—Approach 1	0.032
Range Min Query—Approach 2	0.065

the remaining steps of the two algorithms and the encryption cost. The second approach for RMQ again offers substantially higher benefits for Gowalla, compared to the denser USPS dataset.

In terms of search time (Table 4), all algorithms are extremely efficient, taking less than 0.1 millisecond in both datasets. In fact, no statistically significant difference is observed in terms of search time for the two datasets. This is expected, since the three algorithms execute a constant number of index queries, i.e., they need to fetch and decrypt a constant number of small answers (two for the RSQ and the first approach of Range Min Query, and four for the second approach of the Range Min Query). The length of the query range and the dataset size do not have an effect on the performance, since both the number of queries and the cost of each query remains constant at all three methods.

12 CONCLUSIONS AND FUTURE WORK

In this article, we revisited the problem of range search over data outsourced to an untrusted server. Prior techniques are either secure but exhibit prohibitive performance cost, or efficient but with unacceptable privacy leakages. We presented the first concrete framework for practical private range search building upon the definitional framework of SSE. We introduced a variety of schemes with realistic security/efficiency tradeoffs. Our constructions utilize any secure (existing or future) SSE scheme as a black box, and appropriately convert range search to multi-keyword search. They can also be generically or tightly integrated with SSE schemes that provide a locality-aware methodology for retrieving results from contiguous locations at the server's storage, and defend against powerful generic attacks on private range search schemes. In addition, we introduced novel constructions for handling range aggregate queries with non-trivial asymptotic complexities. We formally defined the security of all proposed algorithms formulating the leakages and sketching all proofs, and experimentally demonstrated their practicality.

In our future work, we plan to focus on the more challenging setting of *multi-dimensional* (i.e., *multi-attribute*) range queries. Note that our proposed solutions can be readily tailored to handle multi-dimensional data as well. A typical trick, borrowed from spatial database applications, is to map any multi-dimensional point to a single-dimensional space-filling curve, and handle any multi-dimensional range query as a *set* of single-dimensional queries [53]. Therefore, we can build any of our schemes on the 1D domain of the space-filling curve, and use them to answer each single-dimensional query. The result to a multi-dimensional query is then the *union* of the results of all the single-dimensional queries. Similarly, the leakage of a multi-dimensional query is the *union* of the leakages of the single-dimensional queries as well. While solutions to multi-dimensional range search can be reached using our schemes, we recognize that the the problem is still open for more efficient solutions and merits a more thorough study that we plan to undertake as future work.

APPENDIXES

A.1 SECURITY OF CONSTANT-BRC/URC

THEOREM 12.1. *Given* Π*, an adaptively secure SSE scheme, and* H*, a collision-resistant hash function, the Constant-BRC/URC schemes are* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure in the random oracle model, where* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *are the leakage functions presented in Section 5 and query intersections are not allowed.*

PROOF SKETCH. It is important to highlight that this proof requires for the Constant schemes to not allow query intersections. The simulator takes as an input the $\mathcal{L}_1$ leakage to output the encrypted index I. In particular, the simulator invokes the simulator-algorithm of the underlying SSE scheme to initialize the encrypted index I with n random values of appropriate format and length; this may vary across different SSE schemes but only depends on n. Now, the simulator uses $\mathcal{L}_2$ to simulate the queries. First, she uses the search pattern ($\sigma(W)$) to determine if the query has appeared before or if it is a new query. If the query has appeared before, the simulator uses her state $st_{\mathcal{S}}$ to return the same tokens and results as she did in the previous query. For new queries, $\mathcal{L}_2$ includes $RC(w)$, i.e., the range covering of the query w, which denotes a set of nodes that cover the requested range. For each node N_i, $\ell(N_i)$ contains the level of the node in the DPRF tree and $id_{map}(N_i)$, which is the exact mapping of tuple IDs to the leaves of N_i's subtree. For each node, the simulator outputs a token, which is produced by picking a random seed r. This seed is used as a token in the DPRF sub-tree of this node. Then, this token produces all the DPRF leafs using the level of the node $\ell(N_i)$. For each leaf x, the simulator programs the random oracle[6] using the information of id_{map} to store that $H(x)$ will be the search keyword of the value x. Finally, for each $H(x)$, the simulator invokes the simulation-algorithm of the underlying SSE scheme to return the correct result for each value x; this simulation-algorithm requires $id_{map}(x)$ as an input. The simulator stores in her state all decisions she made. In case the adversary outputs a query that intersects with a previous query, the simulator aborts. □

A.2 SECURITY OF LOGARITHMIC-BRC/URC

THEOREM 12.2. *Given* Π*, an adaptively secure SSE scheme the Logarithmic-BRC/URC schemes are* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure, where* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *are the leakage functions presented in Section 6.1.*

PROOF SKETCH. It is important to highlight that this scheme also supports overlapping queries. The simulator takes as an input the $\mathcal{L}_1$ leakage to output the encrypted index I. In particular, the simulator creates D', which contains $n \log m$ records and then invokes the simulator-algorithm of the underlying SSE scheme to initialize the encrypted index I with $n \log m$ random values of appropriate length and format. Now, the simulator uses $\mathcal{L}_2$ to simulate the queries. In particular, she uses the search pattern to determine if the given range has appeared before or if it intersects with any previously queried node; if the latter case is true, then this implies that the requested range contains at least one node that has been retrieved before. She uses the $RC(w)$ function, where RC is either BRC or URC, and for each node $\mu(N_i)$ she uses $\sigma(W)$ to determine if this node has been previously queried. If node $\mu(N_i)$ has appeared before, the simulator uses her state $st_{\mathcal{S}}$ to return the same token and result as she did the previous time for node $\mu(N_i)$. Otherwise, if the node is new, the simulator invokes the simulation-algorithm of the underlying SSE scheme to retrieve the result for this node; this simulation-algorithm requires id_{N_i} as input. The above procedure is repeated by the simulator for all $\mu(N_i)$. Logarithmic-BRC/URC schemes leak all the overlapping nodes induced by overlapping ranges, and the simulator uses $\sigma(W)$ and $RC(w)$ to

[6]The random oracle model is also used in References [2, 13] and we refer the reader to these works and to Reference [43] for further details.

correctly simulate the results of each sub-range in the requested range query. $RC(w)$ also contains information regarding the order of the nodes. The simulator stores in her state all the decisions she made. □

A.3 SECURITY OF LOGARITHMIC-SRC

Theorem 12.3. *Given* Π*, an adaptively secure SSE scheme, the Logarithmic-SRC scheme is* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure, where* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *are the leakage functions presented in Section 6.2.*

Proof Sketch. The simulator takes as an input the $\mathcal{L}_1$ leakage to output the encrypted index I. In particular, the simulator creates D' which contains $2n \log m$ records and then invokes the simulator-algorithm of the underlying SSE scheme to initialize the encrypted index I with $2n \log m$ random values of appropriate length and format. The factor 2 is important due to the extra nodes that we injected in TDAG. Now, the simulator uses $\mathcal{L}_2$ to simulate the queries. In particular, she uses the search pattern ($\sigma(W)$) to determine if $RC(w)$ has appeared before, or if it is new. It is important to mention that $RC(w)$ contains only one node, which means that we do not have any additional leakage as in the previous cases (i.e. we do not leak order information). If the node $\mu(RC(w))$ has appeared before, then the simulator uses her state $st_{\mathcal{S}}$ to return the same token and result as she did in the previous query for node $\mu(RC(w))$. Otherwise, if the node is new, then the simulator invokes the simulation-algorithm of the underlying SSE scheme to retrieve the result for this node; this simulation-algorithm requires $id(RC(w))$ as input. The simulator stores in her state all decisions she made. □

A.4 SECURITY OF LOGARITHMIC-SRC-i$_1$

Theorem 12.4. *Given* Π, Π' *adaptively secure SSE schemes, the Logarithmic-SRC-i*$_1$ *scheme is* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure, where* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *are the leakage functions presented in Section 6.3.*

Proof Sketch. The simulator takes as an input the $\mathcal{L}_1$ leakage to output the encrypted index I. In particular, the simulator creates D'_1, which contains $2n' \log m$ records and then invokes the simulator-algorithm of the underlying SSE scheme to initialize the encrypted index I_1 with $2n' \log m$ random values and D'_2 for I_2 with $2n \log n$ random values. Again, the random values are of appropriate length and format. The factor 2 is important due to the extra nodes that we injected in TDAG. Now, the simulator uses $\mathcal{L}_2$ to simulate the queries. In particular, she uses the search pattern to determine if $RC(w)$ has appeared before or if it is new. In case that the queried range has appeared before, the simulator uses her state $st_{\mathcal{S}}$ to return the same token and result as she did in a previous query. Otherwise, if the node is new, the simulator uses $\sigma(W)$ to determine if $\mu(RC(w))_{TDAG_1}$ or $\mu(RC(w'))_{TDAG_2}$ have been previously queried. If $\mu(RC(w))_{TDAG_1}$ has been queried before, she returns the same results as the previous time. Otherwise, she invokes the simulation-algorithm of the underlying SSE scheme to retrieve the result for this node; this simulation-algorithm requires $unq_{RC(w)}$ as an input. The procedure is similar for $\mu(RC(w'))_{TDAG_2}$, only now the simulation-algorithm of the underlying SSE scheme takes as input $id(RC(w'))$. The simulator stores in her state all decisions she made. □

A.5 SECURITY OF LOGARITHMIC-SRC-i$_2$

Theorem 12.5. *Given* Π, Π' *adaptively secure SSE schemes, the Logarithmic-SRC-i*$_2$ *scheme is* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure where* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *are the leakage functions presented in Section 6.4.*

Proof Sketch. The simulator takes as an input the $\mathcal{L}_1$ leakage to output the encrypted index I. In particular, the simulator creates D'_1, which contains $2 * m$ random records and then invokes

the simulator-algorithm of the underlying SSE scheme to initialize the encrypted index I_1 and D_2' for I_2 with $2n \log n$ random values of appropriate length and format. Now, the simulator uses $\mathcal{L}_2$ to simulate the queries. In particular, she uses the search pattern to determine if $\mu(w_L)$, $\mu(w_R)$, $\mu(RC(w))$ have appeared before or if they are new. In case they have been queried before, the simulator uses her state $st_{\mathcal{S}}$ to return the same tokens and results. Otherwise, she uses $\sigma(W)$ and $id(RC(w))$ to return the new results and tokens as she did in the previous proofs. The simulator stores in her state all decisions she made. □

A.6 SECURITY OF LOGARITHMIC-SRC-i*

THEOREM 12.6. *The Logarithmic-SRC-i* scheme is $(\mathcal{L}_1, \mathcal{L}_2)$-secure where $\mathcal{L}_1$ and $\mathcal{L}_2$ are the leakage functions presented in Logarithmic-SRC-i$_1$ and Logarithmic-SRC-i$_2$, respectively.*

PROOF SKETCH. We describe the simulation for $TDAG_2$, and, similarly, we can simulate $TDAG_1$ and T_1. The simulator takes as an input the $\mathcal{L}_1$ leakage to output the encrypted index I_2. In particular, the simulator creates arrays $A_0, \ldots, A_{\log n}$ of size $2n$ entries and initializes each entry with random values of appropriate length and format. Additionally, she creates a dictionary with $2n$ entries, where each entry contains two random values (r_1, r_2). Now, the simulator uses $\mathcal{L}_2$ to simulate the queries. In particular, she uses the search pattern to determine if $RC(w)$ has appeared before, or if it is new. If $RC(w)$ has appeared before, the simulator uses her state $st_{\mathcal{S}}$ to return the same token and result as she did the previous time. Otherwise, she chooses a random entry from the dictionary, marks in her state this entry as "used" and returns r_1, r_2 as a token. She uses $\ell = \log |id(RC(w))|$ to choose one random bucket from the array A_ℓ and programs the random oracle $H(r_2)$ to point to this bucket. Then she marks in her state that this bucket is "used." Additionally, she programs $H(\dot{})$ to output the correct tuple-IDs using $id(RC(w))$. The simulator succeeds, because for any new query $RC(w)$, which includes results in arrays $A_0, \ldots, A_{\log n}$, she can pick uniformly at random an "un-used" bucket. If an "un-used" bucket does not exist, this means that the client has asked for all possible queries in that level. Intuitively, the important property that assures the aforementioned uniform choices is that from each array A_i we return results of equal size. □

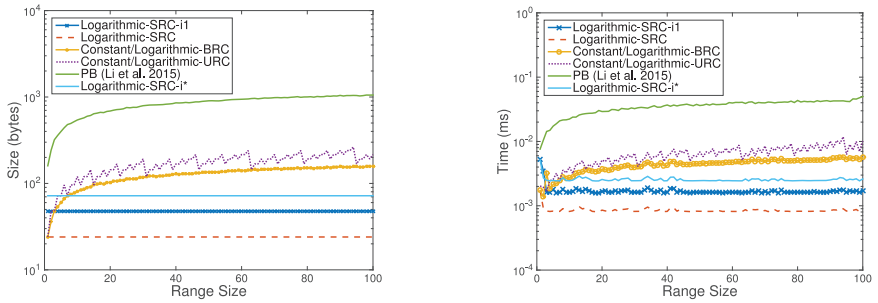
A.7 SECURITY OF RSQ

THEOREM 12.7. *Given Π, adaptively secure SSE schemes, the RSQ scheme is $(\mathcal{L}_1, \mathcal{L}_2)$-secure where $\mathcal{L}_1$ and $\mathcal{L}_2$ are the leakage functions presented in Section 10.1.*

PROOF SKETCH. The simulator takes as an input the $\mathcal{L}_1$ leakage to output the encrypted index I. In particular, the simulator creates D_1, which contains m random records and then invokes the simulator-algorithm of the underlying SSE scheme to initialize the encrypted index I. Now, the simulator uses $\mathcal{L}_2$ to simulate the queries. In particular, she uses the search pattern $\sigma(W)$ to determine if $\mu(w_L)$ and $\mu(w_R)$ have appeared before, or if they are new. For the tokens that have been queried before, the simulator uses her state $st_{\mathcal{S}}$ to return the same tokens. Otherwise, she return the new tokens. The simulator stores in her state all decisions that she makes. □

B.1 EXPERIMENTS: QUERY COSTS AT OWNER

To verify the practicality of the proposed approaches, we also evaluated experimentally the costs incurred at the owner during query execution. In particular, we generated random queries for ranges of size 1 to 100 over the domain $A = \{0, \ldots, 2^{20}\}$, and measured the size of each query (in bytes) and the time required for generating it. Figures 20(a) and (b) present the average results over 1,000 executions for each query range size. For clarity, the curves corresponding to Constant and Logarithmic for the same range-covering technique (i.e., BRC or URC) are grouped together



(a) Query size (b) Query generation time

Fig. 20. Query costs at owner.

in both figures, since (i) the same range-covering technique leads to the same query size in both Constant and Logarithmic and (ii) the difference in query generation time between Constant and Logarithmic is negligible, as token generation times of Constant are only slightly higher than those of Logarithmic (due to the GGM value expansion starting from the root).

As expected, Logarithmic-SRC, Logarithmic-SRC-i$_1$, and Logarithmic-SRC-i* feature the smallest query sizes, requiring one, two, and three tokens per query, respectively, with each token taking 24 bytes. On the other hand, the query size in both BRC- and URC-based schemes scales logarithmically with the range size. The saw-like trend of URC is due to the worst-case decomposition, whose size oscillates with the range size regardless of the randomly selected query position. In contrast, in BRC, different query positions for the same range size lead to different numbers of tokens, which are smoothed by the averaging.

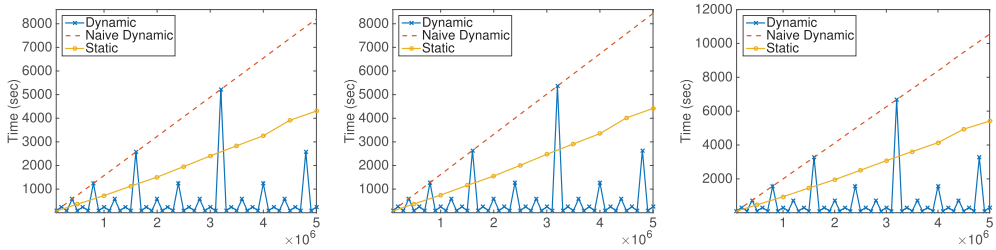
Figure 20(b) depicts the wall-clock time required in Trpdr to compute the tokens for all nodes that are produced by the range-covering technique. We see that the curves follow a similar trend to those of Figure 20(a), which is expected since the total time is dominated by the PRF evaluations (one for each covering node/keyword). Observe that these operations are lightweight, and are typically carried out in less than 0.01 millisecond in all our schemes. PB, on the other hand, incurs larger query sizes and higher query generation times than all our schemes, mainly due to the excessive number of cryptographic hash functions involved in its Bloom filter approach.

It is important to note that the reported costs of our methods do not depend on the datasets, or on the domain; they only depend on the *position* of the range in the binary tree over the domain. As such, the presented results are representative of all possible domains and datasets.

B.2 EXPERIMENTS: HANDLING OF UPDATES

In this set of experiments, we consider the case of dynamic datasets, handled as described in Section 9. For these experiments, we fix the parameter s to 2. This means that after every two new indexes, we initiate a consolidation phase that merges one or more indexes to construct a new one. In these experiments, the batch size is set to 100,000 updates.

Figure 21(a) plots the time required for dynamic Logarithmic-SRC-i$_1$ (labeled as "dynamic" in the figure) to maintain the index, when ignoring the time spent on network operations. That is, the shown results account only for the time required for decrypting, reconstructing, and re-encrypting the indexes, as dictated by the dynamic algorithm. As a reference, the plot also includes (a) the cost of the static Logarithmic-SRC-i$_1$ to build the same index, *assuming that the whole dataset is made available at once*, labeled as "static," and (b) the cost of a naive dynamic version, which simply downloads all data after every batch of updates, decrypts the data, and reconstructs the updated



(a) Cost when ignoring the network cost (b) Cost over a 1 Gbps link (c) Cost over a 100 Mbps link

Fig. 21. Update costs over the size of data included in the index (x-axis).

indexes from scratch, labeled as "naive dynamic." Notice that static Logarithmic-SRC-i$_1$ is not a viable algorithm for dynamic datasets, since it requires that all data is available *a priori*; it is only presented here as a lower bound, to demonstrate the overhead that is added by making our algorithm dynamic. This overhead includes the potential download of indexes from the server and the decryption of the data stored in these indexes.

As expected, dynamic Logarithmic-SRC-i$_1$ substantially outperforms the naive dynamic counterpart, since it typically needs to download and re-encrypt only a small part of the index. In the worst case, the cost of dynamic Logarithmic-SRC-i$_1$ becomes approximately the same as the cost of its naive dynamic counterpart. This worst case happens at multiplies of the batch size (100,000 in this experiment) with a power of $s = 2$, e.g., at 3.2 million updates in this plot. During these worst cases, dynamic Logarithmic-SRC-i$_1$ needs to download all indexes from the server to merge them and to form a new, consolidated index.

However, an attractive property of dynamic Logarithmic-SRC-i$_1$ is that the frequency of these peaks that dynamic Logarithmic-SRC-i$_1$ needs to download and consolidate all indexes is reduced as the dataset is growing. This property is already visible in the plots; the peak initially occurs at 1×10^5 updates, and then repeats after 1×10^5, 2×10^5, 4×10^5, 8×10^5, and, finally, after 1.6×10^6 updates. That is, the distance of the peaks grows exponentially as the dataset grows.

We also see that dynamic Logarithmic-SRC-i$_1$ does not introduce substantial overhead compared to the static counterpart that requires the full data to be made available *a priori*. The maximum overhead happens when all data needs to be downloaded and re-uploaded from scratch. In this case, the cost of dynamic Logarithmic-SRC-i$_1$ is about twice the cost of static. As discussed, the frequency of these worst-case points is reduced exponentially as the data set grows.

Our results up to now did not consider the network-induced delays that are caused by uploading and downloading the indexes to the remote server. These delays are clearly unavoidable, and depend only on the network bandwidth and the size of the indexes to be uploaded. To examine the practicality of the proposed algorithm, we simulated the performance of the compared algorithms assuming configurations with fixed network bandwidths. Figures 21(b) and (c) plot the total time required for maintaining the index (including the network delays) for network links of bandwidths 100Mbps and 1Gbps. As expected, the cost of all algorithms scales by a small constant factor, compared to the results without network delays [Figure 21(a)]. This factor is linearly related to the bandwidth of the link, and is almost equal in the dynamic and the naive dynamic algorithms. For the static algorithm, the increase is slightly smaller, since the algorithm includes one less network interaction, i.e., data does not need to be downloaded first from the remote server. However, even under very restrictive bandwidth assumptions compared to today's industry norms (100Mbps), we see that the dynamic algorithm achieves good performance, comparable to the case that the network cost is ignored.

REFERENCES

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order preserving encryption for numeric data. In *SIGMOD*.

[2] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. 2016. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *STOC*.

[3] Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. 2005. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms* (2005).

[4] Guy E. Blelloch. 1990. *Prefix Sums and Their Applications*. Technical Report CMU-CS-90-190.

[5] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. of the ACM* (1970).

[6] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. Order-preserving symmetric encryption. In *EUROCRYPT*.

[7] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*.

[8] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. 2015. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*.

[9] Dan Boneh and Brent Waters. 2007. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography*. Springer, 535–554.

[10] Raphael Bost. 2016. Sofos: Forward secure searchable encryption. In *CCS*.

[11] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *CCS*.

[12] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, M. Rosu, and Michael Steiner. 2014. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS*.

[13] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2013. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*.

[14] David Cash and Stefano Tessaro. 2014. The locality of searchable symmetric encryption. In *EUROCRYPT*.

[15] Yan-Cheng Chang and Michael Mitzenmacher. 2005. Privacy preserving keyword searches on remote encrypted data. In *ACNS*.

[16] Melissa Chase and Seny Kamara. 2010. Structured encryption and controlled disclosure. In *ASIACRYPT*.

[17] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. 2016. Practical order-revealing encryption with limited leakage. In *IACR-FSE*.

[18] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: Improved definitions and efficient constructions. In *CCS*.

[19] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2011. Searchable symmetric encryption: Improved definitions and efficient constructions. *J. Comput. Sec.* (2011).

[20] Jonathan L. Dautrich Jr. and Chinya V. Ravishankar. 2013. Compromising privacy in precise query protocols. In *EDBT*.

[21] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. *arXiv Preprint* (2017).

[22] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. 2016. Practical private range search revisited. In *SIGMOD*.

[23] Ioannis Demertzis and Charalampos Papamanthou. 2017. Fast searchable encryption with tunable locality. In *SIGMOD*.

[24] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. 2015. Rich queries on encrypted data: Beyond exact matches. In *ESORICS*.

[25] Johannes Fischer and Volker Heun. 2006. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *CPM*.

[26] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2017. HardIDX: Practical and secure index with SGX. *arXiv Preprint* (2017).

[27] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph.D. Dissertation. Stanford University.

[28] Craig Gentry. 2010. Computing arbitrary functions of encrypted data. *Commun. of the ACM* (2010).

[29] Eu-Jin Goh et al. 2003. Secure indexes. *IACR Cryptology ePrint Archive* (2003).

[30] Oded Goldreich. 2006. *Foundations of Cryptography, Vol. 1*. Cambridge University Press.

[31] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to construct random functions. *J. ACM* 33, 4 (1986), 792–807.

[32] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.

[33] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*.
[34] Florian Hahn and Florian Kerschbaum. 2016. Poly-logarithmic range queries on encrypted data with small leakage. In *CCS Workshop*.
[35] Bijit Hore, Sharad Mehrotra, Mustafa Canim, and Murat Kantarcioglu. 2012. Secure multidimensional range queries over outsourced data. *VLDB J.* (2012).
[36] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. 2004. A privacy-preserving index for range queries. In *VLDB*.
[37] Caleb Horst, Ryo Kikuchi, and Keita Xagawa. 2017. Cryptanalysis of comparable encryption in SIGMOD'16. In *SIGMOD*.
[38] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2014. Inference attack against encrypted range queries on outsourced databases. In *CODASPY*.
[39] Seny Kamara and Tarik Moataz. 2017. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *EUROCRYPT*.
[40] Seny Kamara and Charalampos Papamanthou. 2013. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography*.
[41] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *CCS*.
[42] Panagiotis Karras, Artyom Nikitin, Muhammad Saad, Rudrika Bhatt, Denis Antyukhov, and Stratos Idreos. 2016. Adaptive indexing over encrypted numeric data. In *SIGMOD*.
[43] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography*. CRC press.
[44] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic attacks on secure outsourced databases. In *CCS*.
[45] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2017. Accessing data while preserving privacy. *arXiv Preprint* (2017).
[46] Florian Kerschbaum and Axel Schroepfer. 2014. Optimal average-complexity ideal-security order-preserving encryption. In *CCS*.
[47] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. 2013. Delegatable pseudorandom functions and applications. In *CCS*.
[48] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *PVLDB* (2012).
[49] Kevin Lewi and David J. Wu. 2016. Order-revealing encryption: New constructions, applications, and lower bounds. In *SIGSAC*.
[50] Rui Li, Alex X. Liu, Ann L. Wang, and Bezawada Bruhadeshwar. 2014. Fast range query processing with strong privacy protection for cloud computing. *PVLDB* (2014).
[51] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. 2015. Modular order-preserving encryption, revisited. In *SIGMOD*.
[52] Ian Miers and Payman Mohassel. 2017. IO-DSSE: Scaling dynamic searchable encryption to millions of indexes by improving locality. In *NDSS*.
[53] Bongki Moon, H. v. Jagadish, Christos Faloutsos, and Joel H. Saltz. 2001. Analysis of the clustering properties of the hilbert space-filling curve. *TKDE* (2001).
[54] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference attacks on property-preserving encrypted databases. In *CCS*.
[55] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. 2014. Dynamic searchable encryption via blind storage. In *SP*.
[56] Rafail Ostrovsky. 1990. Efficient computation on oblivious RAMs. In *STOC*.
[57] Mark H. Overmars. 1983. *The Design of Dynamic Data Structures*. Springer Science & Business Media.
[58] Raluca A. Popa, Frank H. Li, and Nickolai Zeldovich. 2013. An ideal-security protocol for order-preserving encoding. In *SP*.
[59] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*.
[60] Elaine Shi, John Bethencourt, T-HH Chan, Dawn Song, and Adrian Perrig. 2007. Multi-dimensional range query over encrypted data. In *SP*.
[61] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *SP*.
[62] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical dynamic searchable encryption with small leakage. In *NDSS*.
[63] Emil Stefanov and Elaine Shi. 2013. ObliviStore: High performance oblivious cloud storage. In *SP*.

[64] Emil Stefanov, Elaine Shi, and Dawn Song. 2012. Towards practical oblivious RAM. *NDSS* (2012).
[65] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An extremely simple oblivious RAM protocol. In *CCS*.
[66] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. In *PVLDB*.
[67] Peter Van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter Hartel, and Willem Jonker. 2010. Computationally efficient searchable symmetric encryption. In *SDM*.
[68] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In *CCS*.
[69] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*.
[70] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*.

Received January 2017; revised August 2017; accepted November 2017