Building an Encrypted, Distributed, and Searchable Key-value Store

Xingliang Yuan[†‡], Xinyu Wang[†‡], Cong Wang[†‡], Chen Qian[§], Jianxiong Lin[†]
[†] Department of Computer Science, City University of Hong Kong, China
[‡] City University of Hong Kong Shenzhen Research Institute, China
[§] Department of Computer Science, University of Kentucky, Unite States
{xl.y, j.lin}@my.cityu.edu.hk, {xinywang, congwang}@cityu.edu.hk, qian@cs.uky.edu

ABSTRACT

Modern distributed key-value stores are offering superior performance, incremental scalability, and fine availability for data-intensive computing and cloud-based applications. Among those distributed data stores, the designs that ensure the confidentiality of sensitive data, however, have not been fully explored yet. In this paper, we focus on designing and implementing an encrypted, distributed, and searchable key-value store. It achieves strong protection on data privacy while preserving all the above prominent features of plaintext systems. We first design a secure data partition algorithm that distributes encrypted data evenly across a cluster of nodes. Based on this algorithm, we propose a secure transformation layer that supports multiple data models in a privacy-preserving way, and implement two basic APIs for the proposed encrypted key-value store. To enable secure search queries for secondary attributes of data, we leverage searchable symmetric encryption to design the encrypted secondary indexes which consider security, efficiency, and data locality simultaneously, and further enable secure query processing in parallel. For completeness, we present formal security analysis to demonstrate the strong security strength of the proposed designs. We implement the system prototype and deploy it to a cluster at Microsoft Azure. Comprehensive performance evaluation is conducted in terms of Put/Get throughput, Put/Get latency under different workloads, system scaling cost, and secure query performance. The comparison with Redis shows that our prototype can function in a practical manner.

Keywords

Key-value Store; Searchable Encryption

1. INTRODUCTION

In order to manage the persistently growing amount of data, distributed key-value (KV) stores have become the backbone of many public cloud services [11, 16, 33]. Their well-understood advantages include high performance, linear scalability, continuous availability, and even great potentials of high-level support on rich queries and multiple data models, as seen in a number of proposals and implementations of recent KV stores [12, 14, 17, 27]. However, with the growing data breaches, privacy concerns in data outsourcing become even more pressing than before. Recent works from both cryptographic perspective, e.g., [5, 10, 15, 19] and database perspective, e.g., [29, 32, 37], provided solutions with trade-offs among security, efficiency, and query functionality. Yet, most of them focus on the setting of a centralized or logically centralized server. They do not specifically consider the features and the requirements in modern KV stores.

As known, KV stores are probably the simplest data storage system, which stores pairs of primary keys and data values, and allows to access data values when a primary key is given. To benefit a variety of data-driven applications, modern KV stores provide higher-level features. As one promising feature, multiple richer data models such as column-oriented, document and graph data are supported on top of one united KV store. This feature eases the operational complexity for applications that require more than one format of data [13, 14, 27]. For the other popular feature, many KV stores allow for the data access not just from primary keys, but also from other attributes of data via secondary indexes, so as to enable more efficient data access and rich queries [12, 16, 33]. Although promising, building an encrypted, distributed KV stores still face gaps and encounter challenges, especially for preserving the above salient features in a privacy-preserving manner.

Limitations of prior work. One straightforward approach is to directly store encrypted data along with the (possibly randomized) data identifier/label [30]. However, it only allows limited encrypted data retrieval by the identifier/label, preventing from all possible queries via other secondary attributes of data. Besides, this approach does not consider the support of multiple data models.

Another seemingly plausible approach is to treat KV stores as a blackbox dictionary, and to build an encrypted secondary index, like the one proposed by Cash et al. in [5]. But this direct combination, though slightly enhancing the first approach by limiting the query support to the encrypted index design, would inevitably suffer from secure queries with long latency. Because they treat the distributed KV store as a blackbox dictionary, the data locality in the encrypted query processing can hardly be preserved. In other words, the node where the index is accessed could be different from the node where the matched data are stored.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China

DOI: http://dx.doi.org/10.1145/2897845.2897852

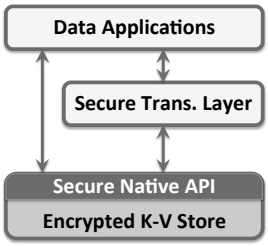


Figure 1: The proposed system framework

This approach will introduce extra inter-node communication overhead, undesirably increasing the query latency. And the issue will be further exacerbated in more complex query processing such as aggregation and range queries.

Design goals. In this work, we aim to build an encrypted and searchable KV store. For data confidentiality, all the data should remain in the encrypted form as long as they leave from the client of the data application. Meanwhile, it should also embrace a bunch of prominent features of non-encrypted distributed KV stores, such as low latency, load balancing, horizontal scalability, and fine availability. In addition, multiple data models should be supported without loss of data confidentiality. And developers can not only commit simple retrieval and update requests on a single encrypted data value but also execute secure queries via secondary attributes of data.

Design challenges. Achieving all the design goals simultaneously is a non-trivial task. As discussed, straightforwardly using existing KV implementations for the security counterpart is unsatisfying. Therefore, we propose to build and evaluate our system by carefully designing the security sub-components of our system whenever necessary, and cope with the following challenges. The first is to securely realize the data partition algorithm across distributed nodes for encrypted data, while achieving high performance lookup, load balancing, and linear scalability. Each encrypted data value should only be located by a specified token generated from an authorized client. Designing a customised secure data partition algorithm allows us to pave the way for securely preserving data locality and improving query efficiency.

The second challenge is to design an overlay that supports multiple data models on top of the encrypted KV store. It should be compatible with the secure data partition algorithm while hiding auxiliary information of ciphertext, e.g., the structural information in column-oriented stores, which is shown to be potentially vulnerable in terms of inference attacks [26]. The third challenge is to design a framework for encrypted and distributed indexes that enable secure queries on given secondary attributes over designated nodes. The index framework should address the data locality to avoid inter-node interactions, and provide us a platform to easily incorporate all the latest secure rich query designs into th encrypted and distributed KV store.

Contributions. Our system design considerations are illustrated below to tackle the above challenges. We will firstly use secure pseudo-random functions and standard symmetric encryption to build an encrypted KV store, shown in Figure 1. Essentially, each KV pair contains a pseudo-random label and an encrypted value: 1) it is simple yet secure, and inherently compatible with the off-the-shelf data partition algorithm (i.e., consistent hashing), achieving load balancing and incremental scalability; 2) it is a stepping stone such that many other data models can be flexibly built on top.

We then introduce a secure transformation layer between the encrypted KV store and data applications, depicted in Figure 1. It formulates an extensible abstraction, which maps the data formated from different data models to simple KV pairs in a privacy-preserving fashion, i.e., both data values and their inherent relationships are strongly protected. In this paper, we choose the column-oriented data model as our first instantiation, supported by wide column stores, e.g., Cassandra [23] and HBase [16]. Afterwards, we also detail the possible adaptation on graph data and document data. As a result, it separates data management and data manipulation from the storage back end, and accrues the benefits of the encrypted KV store.

In order to enable secure queries based on secondary attributes of data, we propose a framework for encrypted local indexes. This framework takes into consideration of distributed processing, KV store benefits, and the flexibility of instantiating various encrypted secondary indexes in the very beginning. By integrating our customised secure data partition algorithm, we can always ensure that the encrypted secondary index co-locates along with its own data on the same node. Namely, secure queries are conducted over distributed nodes in parallel without extra connections, interactions, or data movement. In this paper, we make our first attempt to support secure search queries that retrieve encrypted data with matched attributes. The proposed index construction leverages searchable symmetric encryption (SSE), a security framework for private keyword search, and follows one of the latest SSE constructions with sublinear time, asymptotically optimal space complexity, and provable security [5].

In brief, our contributions are listed as follows:

- We present an encrypted, distributed, and searchable key-value store that ensures strong data protection. We propose a secure data partition algorithm and develop two basic APIs for retrieval and update on a single encrypted data value, i.e., Put/Get. When the system scales out, the affected encrypted data can be relocated without loss of confidentiality.

- We introduce a secure transformation layer that supports secure data modeling. It maps the encrypted data formated from different data models to encrypted KV pairs while hiding the structural information of data. This design is fully compatible with the proposed secure data partition algorithm.

- We propose a framework of encrypted local indexes that considers both data locality and incremental scalability, where each local index resides the data on the same node. Our design enables secure search queries over encrypted and distributed data in parallel. In addition, we conduct formal security analysis to demonstrate the strong security strength of this design.

- We develop the system prototype, and deploy it to Microsoft Azure. The experiments show comprehensive evaluation on Put/Get latency, throughput, index query performance, and system scaling costs. The comparisons with Redis on plaintext data show that

our system functions in a practical manner with little security overhead introduced.

Organization. Section 2 introduces the system architecture and the threat models. Section 3 elaborates on the system design. The security analysis is presented in Section 4. The implementation and the performance evaluation are given in Section 5. We discuss the related works in Section 6, and conclude in Section 7.

2. SYSTEM MODEL

2.1 System Architecture

Figure 2 illustrates the architecture of our proposed private KY store. It consists of three entities: the client of a data application, the dispatcher, and a number of clustered storage nodes, where the latter two entities are deployed at the public cloud. We envision that the data application would like to outsource a huge amount of data to a cloud-powered data store while ensuring the data confidentiality.In general, the dispatcher and the nodes are off-premises commodity servers or virtual machines. They are programmed to execute specific algorithms and operations.

Our system operates in a distributed framework that guarantees data privacy while preserving high performance and incremental scalability. It distributes the encrypted data to all the nodes evenly, which inherently handles heavy workloads without revealing the underlying values. The dispatcher deals with the secure Put/Get requests generated from the client for update and retrieval on a single encrypted data value. It routes the requests via a standard yet secure data partition protocol, i.e., following the algorithm of consistent hashing but over the encrypted domain. It is like a logically centralized hub, forwarding all the requests to the target nodes. To increase the throughput and avoid single-point failure, it can be physically distributed and synchronized between multiple nodes just like in HBase [16], or cached to the client and updated periodically just like in Dynamo [11]. After the request routing, the nodes respond the requests and send the encrypted values back to the client. For the data in the simple KV model, the client will directly generate secure Put/Get requests. While for the data in other models, the secure transformation layer at the client will transform the requests on encrypted structured/semi-structured data into the secure KV Put/Get requests.

In order to query the encrypted data through secondary attributes, the encrypted secondary index integrates the secure data partition algorithm so that each node can index its local data and process a given secure query in parallel. The proposed secure search query generates tokens for each of the nodes to provision them the search ability over distributed encrypted data. In addition, the general framework of encrypted local indexes can readily support known secure queries like counting, range, and aggregation. To handle heavy workloads, new nodes are incrementally added with moderate impact in our system. The affected encrypted data can be directly relocated via the secure data partition algorithm by the corresponding nodes, while the affected indexes are moved in a client-assisted manner, i.e., being rebuilt at the client for security considerations.

2.2 Threat Assumption

In this paper, we are targeting the threats from semi-honest adversaries. They are interested in the data, the query attributes and the query results. In Figure 2, the nodes are allocated in the off-premises public cloud. To meet the requirements of performance and availability, they have to be always online, and thus are vulnerable to security breaches and unauthorized disclosures [21, 34].

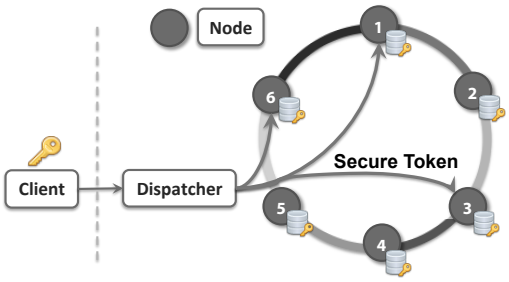


Figure 2: System architecture

On the one hand, outside unauthorized adversaries, who compromise some or all of the nodes, gain the privilege to access the hard disks and the physical memory. They attempt to learn the sensitive information about the data. On the other hand, cloud service providers may also be a threat. They could faithfully maintain the nodes, and execute the operations as required, but intentionally learn the data, or unintentionally mishandle the data to third parties. We assume that the client is in the trusted domain, and does not collude with those adversaries. Besides, the communication channels are assumed to be authenticated and encrypted against eavesdropping. Our security goal is to protect data confidentiality throughout their life-cycles. User authentication and data integrity are orthogonal to our focus.

2.3 Cryptographic Primitives

A symmetric encryption scheme $SE(KGen, Enc, Dec)$ contains three algorithms: The key generation algorithm $KGen$ takes a security parameter k to return a secret key K. The encryption algorithm Enc takes a key K and a value $V \in \{0,1\}^*$ to return a ciphertext $V^* \in \{0,1\}^*$; The decryption algorithm Dec takes K and V^* to return V. Define a family of pseudo-random functions $F : \mathcal{K} \times X \to R$, if for all probabilistic polynomial-time distinguishers Y, $|Pr[Y^{F(k,\cdot)} = 1 | k \leftarrow \mathcal{K}] - Pr[Y^g = 1 | g \leftarrow \{\mathsf{Func} : X \to R\}]| < negl(k)$, where $negl(k)$ is a negligible function in k.

3. THE PROPOSED SYSTEM

This section elaborates on our system design. We first introduce the secure data partition algorithm which can distribute the encrypted data evenly across the nodes without knowing the underlying value. Then we use the column-oriented data model as the instantiation to present how our system maps the data to encrypted KV pairs. Other data models like graph and document data are also supported. Subsequently, we implement two APIs for secure and fast access of the encrypted KV store. To enable the secure query on secondary attributes of data, we propose the construction of encrypted local indexes. We then show how the system scales out when new nodes are added.

3.1 Encrypted Key-value Store

We first introduce the construction of the proposed encrypted and distributed KV store, where each KV pair con-

sists of a pseudo-random label and an encrypted value. The proposed encrypted store is simple yet secure, efficient and scalable. The benefits are two-fold: 1) It is inherently compatible with the state-of-the-art data partition algorithm in distributed systems, i.e., consistent hashing [20]; 2) Many different high-level data models are also supported in the encryption domain, because the data with different structures and formats can be readily mapped to individual KV pairs in a privacy-preserving way.

3.1.1 Secure Data Partition Algorithm

For security, each KV pair (l, v) is protected, where the primary key[1] is kept safe by the secure pseudo-random function (PRF) denoted as l^* and the value is encrypted via symmetric encryption denoted as v^*.

$$\langle l^*, v^* \rangle = \langle P(K_a, l), Enc(K_v, v) \rangle$$

where K_a, K_v are the private keys, and P is PRF. Here, each protected pair can still be distributed to a cluster of nodes evenly by the known consistent hashing algorithm. In detail, the output range of a hash function wraps around to form a ring maintained at the dispatcher shown in Figure 2. Each node is assigned a random value as its position on the ring. The hash of l^* determines the location of v^*. Without loss of generality, the ring is clockwise checked, and the first node with a position label larger than the hash value will be the target location.

This algorithm allows the data to be stored across nodes with balanced distribution. It also enables distributed systems to scale incrementally. When a node is added, only its immediate neighbors are affected, and others remain unchanged. For the subsequent operations like request routing, data retrieval and data update, the security is guaranteed due to the pseudo-randomness of secure PRF and the semantic security of symmetric encryption. The correctness is also ensured, because PRF is a deterministic function, and each encrypted value is still associated with a unique pseudo-random label for identification.

3.1.2 Construction with Secure Data Modeling

Building on top of the encrypted KV store, our system needs to flexibly support multiple different data models. The goal of secure data modeling is to design a specific mapping of data from different formats to encrypted key-value pairs. Therefore, the data applications with richer data models will continue enjoying the benefits of high performance KV stores while ensuring strong data protection. As our first effort, the column-oriented data model is explored, which is widely used in NoSQL systems for semi-structured data, e.g., Cassandra [23]. In particular, the data values are stored in structured tables such that a row name associates with a set of column attributes, i.e., $(R, C_1, \cdots, C_n)$, where R is the row name, and C_i, $i = 1, \cdots, n$, are the column attributes.

At a high level, the data in a table can be mapped to individual KV pairs, where the label is derived from the row name and the column attribute. Here, we will give two constructions from two pragmatic settings of consistent hashing respectively. The former uses the protected row name as the input of consistent hashing to locate the target node, and then incorporates the protected column attribute to match the data, similar to the design in Cassandra [23]. The latter

[1]The term "key" here has a different meaning in cryptography, so we use "label" instead to avoid the ambiguity.

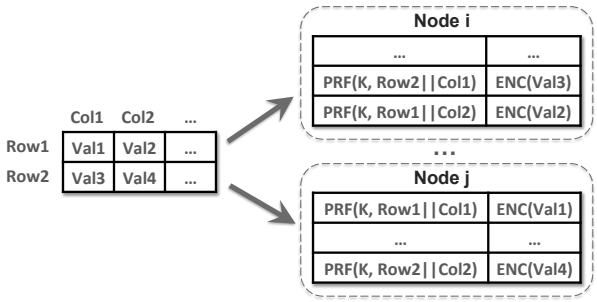


Figure 3: The construction of the encrypted KV store

constructs a composite label from a row name and a column attribute for both data partition and data retrieval, similar to the design in FoundationDB [13]. We note that the two constructions have their performance merit in different application scenarios, but we choose and implement the latter for security considerations, because it fully hides the structural information of encrypted data when the KV store is being accessed and scales out.

Construction I: We use secure PRF to protect the row name and determine the data partition. Each entry in the encrypted KV store is constructed as:

$$\langle G(P(K_a, R), P(K_C, C)), Enc(K_v, v) \rangle$$

The former component is a pseudo-random label, where P and G are PRF, K_a, K_C and K_v are private keys, and v is the data value. Note that $P(K_a, R)$ is the token used for request routing, and thus should be encrypted and separately stored for system scaling later. After locating the data, the target node computes $G(P(K_a, R), P(K_C, C))$ from another token $P(K_C, C)$ to find the encrypted value $Enc(K_v, v)$.

Construction II: In this construction, we propose to use a composite pseudo-random label for both data partition and identification. As shown in Figure 3, each entry is constructed as the format of:

$$\langle P(K_a, R||C), Enc(K_v, v) \rangle$$

where a pseudo-random label $P(K_a, R||C)$ is generated from the concatenation of the row name and the column attribute. As a result, this construction fully scrambles all the encrypted values in the table, but still preserves the structural information and the relationships of the underlying data. Besides, the one-to-one mapping property of PRF still guarantees a unique position for each data value on the consistent hash ring. Therefore, one encrypted data value is always addressed and retrieved by the same pseudo-random label.

Discussion on performance: We note that two constructions bring distinct performance merit. In Construction I, the data in the same row are stored in the same node. Although they are scrambled, the entire row can still be accessed efficiently without introducing traffic to other nodes. However, if some rows are frequently accessed, the corresponding node could suffer from heavy workloads. Construction II solves the issue by distributing the data more evenly. Yet, using a composite label introduces more inter-node traffic when accessing the data values in the same row, because these encrypted data could be stored in different nodes.

Comparison on security: Regrading security, the proposed two constructions both protect the data confidential-

Algorithm 1: Secure Put

Request: $Put(K, R, C, v)$
Data: Private key: K; Row name: R; Column attribute: C; Data value: v.
Result: $true$ or $false$.
begin

client:
1 $K_a \leftarrow H(K, \text{“}address''); $
2 $K_v \leftarrow H(K, C);$
3 $l^* \leftarrow P(K_a, R||C);$
4 $v^* \leftarrow Enc(K_v, v);$
5 $c \leftarrow 1;$
dispatcher:
6 $i \leftarrow route(l^*);$
7 **for** node $= i, \cdots, i+N:$ **do**
8 $b \leftarrow put(l^*, v^*);$
client:
9 **if** $b = true$ **then**
10 $c++;$
11 **if** $c = N_w$ **then**
12 return $true$;
13 return $false$;

Algorithm 2: Secure Get

Request: $Get(K, R, C)$
Data: Private key: K; Row name: R; Column attribute: C.
Result: Encrypted value: v^*.
begin

client:
1 $K_a \leftarrow H(K, \text{“}address''); $
2 $l^* \leftarrow P(K_a, R||C)$
3 $c \leftarrow 1;$
dispatcher:
4 $i \leftarrow route(l^*);$
5 **for** node $= i, \cdots, i+N:$ **do**
6 $v^* \leftarrow get(l^*);$
client:
7 **if** $v^* \neq null$ *and* $v^* = v^*_{i-1}$ **then**
8 $c++;$
9 **if** $c = N_r$ **then**
10 return v^*_i;
11 $v^* \leftarrow \perp;$
12 return $false$;

ity because all the data are encrypted after leaving from the client. Without the appropriate pseudo-random labels, the encrypted data will not be correctly accessed.

However, Construction I has additional leakage, which can be exploited for inference attacks [26]. As mentioned, the token $P(K_a, R)$ should be encrypted and stored to enable data relocation. Therefore, the data that belong to the same row are known. Besides, the data with the same column attribute are known as well if they are requested, because the same column will result in the same token $P(K_C, C)$. Then if the adversary has a reference database which is highly correlated to the encrypted one, he might be able to recover the data values via frequency analysis, i.e., recording histograms of frequent KV pairs in two databases and inducing the underlying values by matched or close histograms [26]. While in Construction II, the label $P(K_a, R||C)$ does not disclose the above information for both data access and data relocation. From the perspective of security, we adopt Construction II in our system prototype, and based on which, we present two basic APIs later for the secure access of the encrypted KV store.

3.1.3 Extensions to Other Data Models

In addition to column-oriented data, our system can also extend to other data models by mapping them to a set of KV pairs just like the recent development on NoSQL data stores called multi-model databases [13, 14, 17]. Here, we use two other common types of data as examples, i.e., graph data and document data.

On the support of graph data: For the graphs, they are essentially structured data. As an example, a social graph is used to represent the user connections in social network applications. In general, the social connections of each user are stored in the form of adjacency list: $adj(u) = \{u_1, \cdots, u_m\}$, where u is the user id, and $\{u_1, \cdots, u_m\}$ are the ids of the user's friends. To map $adj(u)$ to KV pairs, we resort to a self-incremental counter c. The KV pairs for the user u are constructed as:

$$\{\langle G(P(K_u, u), c), Enc(K_c, u_c)\rangle\}_m$$

K_u and K_c are the private keys, and c is increased from 1 to m. To fetch the adjacency list, the client generates $P(K_u, u)$ for the dispatcher. Then the dispatcher computes and routes $G(P(K, u), c)$ by increasing c incrementally until the label does not exist in the encrypted KV store.

On the support of document data: For the documents, they are objects associated with a unique document id F_{id}. We store them as id and document pairs, if the size of document does not exceed the size limit of data values, i.e., $(P(K_{id}, F_{id}, Enc(K_F, F))$, where K_{id} and K_F are the private keys, and F is the document. If the document is too large, it is required to be split into smaller chunks with moderate sizes. Given a document $F = \{f_1, \cdots, f_m\}$ with m chunks of the same size (the last one is padded if needed), the KV pairs to the document are constructed as:

$$\{\langle G(P(K_{id}, F_{id})), c), Enc(K_F, f_c)\rangle\}_m$$

K_{id} and K_F are the private keys, and c is increased from 1 to m. To retrieve the document, the client sends $P(K_{id}, F_{id})$ to the dispatcher. The encrypted chunks are fetched by computing $G(P(K_{id}, F_{id}), c)$ with the incremental c. After all the chunks are returned, they are decrypted and concatenated as a complete file at the client.

3.1.4 Basic Requests for Encrypted Key-value Stores

In our system, two basic APIs Put and Get are provided to support retrieval and update on a single encrypted data value. Implementing Put and Get has two requirements. One is to demand our system to keep data confidential during each request. The other is to require the requests to maintain consistency across data replicas if the nodes encounter accidental failure. The implementation of Put and Get is presented in Algorithm 1 and Algorithm 2 respectively. For

Algorithm 3: Build encrypted local indexes

Function: $Build(K, \mathbf{R}, \mathbf{C}, \mathbf{V})$
Data: Private key: K; Row name set: $\mathbf{R}$; Column attribute set: $\mathbf{C}$; Data values:$\mathbf{V}$.
Result: Encrypted index: $\mathbf{I}$.
begin

```
I : {I_1, ..., I_n} ← init(); // init index of n nodes
for i = 1 to n do
    K_C ← H(K, i);
for C = C_1, ..., C_|C| do
    init counters c for C : {c_1 ← 1, ..., c_n ← 1};
    for ∀v ∈ V associated with C do
        l* ← P(K_a, R||C), where R ∈ R;
        i ← route(l*);
        choose I_i ∈ I, c_i ∈ c for node i;
        t_1 ← F_1(K_C, 1||C), t_2 ← F_2(K_C, 2||C);
        α ← G_1(t_1, c_i);
        β ← G_2(t_2, c_i) ⊕ l*;
        insert(α, β);
        c_i ++;
```

a given request, the client generates the pseudo-random label $P(K_a, R||C)$, and then sends it to the dispatcher for request routing. With the label, the dispatcher locates the node from the consistent hash ring. By using the same label, the target node will find the encrypted value.

The reliability and availability in distributed systems are ensured by replication. In our prototype, we follow the general approach used by DynamoDB [11] such that the data replicas reside at the next r nodes clockwise to the first chosen node, where r is the number of replicas. To ensure the consistency between them, we adopt the quorum-based protocol [36] to implement the two APIs, which is widely used in distributed systems. The protocol complies with the rule such that $N_r + N_w > N$, where N_r is the number of nodes with replicas that perform Get successfully, and N_w is the number of nodes with replicas that perform Put successfully.

We note that the replicas are identical for each encrypted value in current design. In this paper, the adversary is assumed to be passive, who will not modify or delete the data maliciously. Thus, revealing the equality of replicas seems to be harmless to the data confidentiality. In future, we will explore secure replication schemes via secret sharing or encoding techniques [2, 24] to hide the equality of replicas and achieve stronger data protection.

3.2 Encrypted Local Indexes

To support secure and efficient queries over encrypted data, our system needs a general framework which can support various encrypted indexing techniques. Given encrypted secondary attributes, the matched encrypted data values should be pulled out from the index without harming the data confidentiality. As the instantiation for column-oriented data, we aim to support secure search queries on column attributes. In our context, the basic design considerations include space efficiency, query efficiency, and index scalability. Accordingly, we explore symmetric-key based indexes which are studied thoroughly in searchable symmetric encryption (SSE) [5, 6, 10, 19]. The SSE-based indexes achieve sublinear search time with controlled information leakage. Yet, most of them are neither space efficient nor scalable for very large scale datasets. Thus, we are interested in the design with optimal space complexity and good scalability [5].

As prior SSE-based indexes are not designed for distributed systems, they do not specifically consider the locality of the data and the index. Even if they can be applied, painful communication overhead will be introduced since the data and the index are accessed on different nodes. Besides, they hardly scale in an incremental way. When the volume of data exceeds the index size, it is inevitable to rebuild the entire index. Instead, we propose to design a framework that support known constructions of encrypted indexes. And the objective is to build encrypted local indexes for secure queries over distributed and encrypted data, which is already shown practical in NoSQL systems [11, 25].

Encrypted local index construction: For the column-oriented data model, our system indexes the data with same column attributes. In general, such column index can be treated as an invert index, where the attributes and the associated values are stored in a list. Considering the locality of data and index, we integrate the proposed secure data partition algorithm to design a group of fully distributed local indexes. Each node is enabled to query the encrypted local index for its own encrypted data. As mentioned, we adopt the SSE index construction in [5] for space efficiency and easy implementation. In particular, this encrypted index is essentially an encrypted dictionary, so the KV store can be directly treated as the underlying data structure for the index. The implementation overhead is minimized.

The building procedure of the encrypted index is presented in Algorithm 3. We also transform the column index into the encrypted KV pairs. Here, $\mathbf{R}$ and $\mathbf{C}$ are the sets of the row names and the column attributes respectively. the token α is generated via PRF on the inputs of a column attribute C and a local counter c_i for node i, i.e., $\alpha = G_1(t_1, c)$, where $t_1 = F_1(K_C, 1||C)$, and β is a masked label l^* of the encrypted data values, i.e., $\beta = G_2(t_2, c_i) \oplus l^*$, where $t_2 = F_2(K_C, 2||C)$. We note that the building process is suitable for such a case that when one party wants to import a large number of new data into our system. Alternatively, the client can compute α and β individually for each newly inserted data. From the construction, the index space complexity achieves optimal, i.e., $O(n)$, where n is the number of indexed data values.

Remarks: The proposed encrypted local indexes are readily built for various types of attributes in other data models. For example, the metadata in document data are treated as the attributes to access the data in specific fields of documents. Moreover, secure keyword search can be further enabled within the encrypted data with matched attributes. For reliability, the index can be periodically dumped into the KV store. Besides, it can be synchronized in background threads across replication nodes for availability and consistency just like the plaintext systems [23].

3.2.1 Secure Query Operation

With the encrypted local indexes, the encrypted data values for a given attribute can efficiently be accessed. The secure query operation is presented in Algorithm 4. The client first generates secure tokens $\{t_1, t_2\}_n$ for total n nodes. After the tokens are received, the local index is enumerated, i.e., computing α via $G_1(t_1, c_i)$ to get β, and then unmasking β

Algorithm 4: Secure Query on a given attribute

Function: $Query(K, C)$
Data: Private key: K; Queried column attribute: C.
Result: Result encrypted values:$\mathbf{V}_r$.
begin
  client:
  for $i = 1$ *to* n **do**
    $K_C \leftarrow H(K, i)$;
    $t_1 \leftarrow F_1(K_C, 1||C)$, $t_2 \leftarrow F_2(K_C, 2||C)$;
  send $\{t_1, t_2\}_n$ to the dispatcher;
  for node $= 1, \cdots, n$ **do**
    $c_i \leftarrow 1$;
    $\alpha \leftarrow G_1(t_1, c_i)$;
    while *find* $(\alpha) \neq \perp$ **do**
      $\beta \leftarrow find(\alpha)$;
      $l^* \leftarrow \beta \oplus G_2(t_2, c_i)$;
      $v^* \leftarrow get(l^*)$;
      Add v^* to $\mathbf{V}_r$;
      c_i++;
      $\alpha \leftarrow G_1(t_1, c_i)$;

via XORing $G_2(t_2, c_i)$ to have the label l^*. Based on l^*, the encrypted value v^* is sent back to the client. By considering data locality, each local index is processed in parallel.

Next, we will show how our proposed framework for local indexes can further support secure rich queries such as count, range, and aggregation on the encrypted data. As the framework integrates the secure data partition algorithm, all the latest secure query designs will easily be incorporated to realize various encrypted local indexes with the ability of parallel processing.

On the support of equality check: For the queries on equality checking and counting, one can use deterministic encryption (DET) which preserves the equality of the underlying values. To improve the security, one can apply randomized encryption on the DET ciphertext [32]. But this approach requires the decryption of the queried column for equality check. As a result, the equality of DET ciphertext is revealed as long as the column is queried.

To achieve better security, we propose a token matching based approach, where the token securely encodes the indexed data value. Such design only discloses the equality of the underlying values for a given query value. Explicitly, we append another token $\gamma = G_3(t_3, c_i)$ to β, where $t_3 = F_3(K_C, 3||C||v)$. And the counter c_i ensures that γ is different for the data with same underlying values. As a result, another token t_3 will be generated for each node to conduct equality checking, i.e., computing $\gamma = G_3(t_3, c_i)$ for the check. Due to the deterministic property of PRF, only if the underlying values are equal, γ will be matched. The equality of other encrypted values is not disclosed.

On the support of other rich queries: To further support rich operators and functions, we can apply property-preserved encryption and homomorphic encryption, similar to prior private databases such as CryptDB [32]. We emphasize that our proposed local index framework can fully leverage the ability of parallel processing in distributed systems; that is, each node independently performs the computation on its local encrypted values.

For range queries, one may adopt order-preserving encryption [3] or order-preserving index techniques [31]. Here, we use the order-preserving encoding technique proposed in [31] as an example. In this design, a tree based index is proposed, where the order of the data value is encoded. This index can be represented as the order and ciphertext pairs to be stored in our KV store. The ciphertext here is a masked addressing label of an encrypted data value. Likewise, based on the secure data partition algorithm, each node can support secure range queries on its own encrypted data.

For the aggregation query on encrypted data with matched attributes, Paillier encryption [28] will be used for data value encryption. Then each node will leverage our proposed local index and conduct the query operation to enumerate and aggregate the encrypted values. If the total number of nodes is small, the intermediate results can be directly sent back to the client for finalization. Otherwise, the intermediate results can also be aggregated at some dedicated nodes for further reduction and finalization.

Remarks: The proposed framework of encrypted local indexes requires the client to generate tokens for each node. Thus, the number of tokens scales linearly with the number of the nodes. If we adopt the existing SSE-based index as an encrypted global index, only one token will be sent. But the global index will introduce extra interactions between different nodes because it does not consider the data locality. As shown in our experiment later, the encrypted local indexes outperform the global one, i.e., 3 times faster when querying the same number of encrypted data values. As a result, the overhead introduced by the transmission of tokens is much smaller than the overhead introduced by the interactions. Besides, the local indexes of individual nodes can be processed in parallel, so the benefits of incremental scalability are preserved.

3.3 Adding Nodes

When the workloads or the volumes of data rapidly increase, they will exceed the storage and processing power of the nodes in the cluster. Thus, our system should be able to accommodate the change by adding new node smoothly, which is a necessity for distributed systems. Technically, when the system scales out, the secure data partition algorithm updates the consistent hash ring accordingly. Then the new node will be assigned to a position on the ring. After that, only the affected node (i.e., its neighbor) relocates the encrypted data and the encrypted indexes.

3.3.1 Data Relocation

Because the proposed encrypted KV store is compatible with the consistent hashing algorithm, the affected nodes can directly move the data to the newly added ones from the update location. For example, assume that node A is assigned to store the data with addressing label $l^* \in [x, y)$, where x is the position of its preceding neighbor B, and y is the position of A in the ring. If a new node C is added between A and B with position z, the data will be moved to C if its label $l^* \in [x, z)$. During the procedure, the relocated data is still strongly protected.

3.3.2 Index Relocation

Recall that the encrypted indexes co-locate with the data on the same nodes. Therefore, when the data are relocated, the index should also be moved to the same target node. In

terms of security, the movement for the index is implemented in a client-assisted manner, i.e., rebuilding the affected index at the client. If we give the tokens to enable the cloud to partition the index, the structural information will be revealed; that is, the data associated with same attributes will be known without being queried. We note that the rebuilding operation is usually inevitable when SSE-based indexes are out of capacity [5, 10, 19]. But our design only requires to rebuild the affected local indexes rather than rebuilding all of them. To maintain the availability, the involved nodes can create auxiliary indexes for caching the update during the asynchronous rebuilding process. After the completion of movement, the entries in auxiliary indexes will be re-inserted to the new indexes.

3.4 Secure Transformation Layer

In order to fully leverage the encrypted KV store for security, usability, and functionality, we leverage a secure transformation layer to transform the plaintext requests and queries to the basic APIs for the access of the encrypted KV store. First, it encrypts the data values and the data attributes before outsourcing them to the nodes in the public cloud. It conducts secure data modeling at the client to hide the structural information of data formated from different models. Second, the transformation layer can also phase queries with different functionalities on the secondary attribute, and generate secure tokens for the computational operations over encrypted indexes.

We emphasize that our design fits with the trend of NoSQL development. A new class of database engine has emerged to begin supporting multiple data models on top of a single KV store [13, 14, 27, 35]. In that way, the KV stores erase the complexity of data operation and development, and benefit many different kinds of applications [17]. Meanwhile, most of the NoSQL systems [12, 25, 33] support the client to submit the queries on the secondary attributes. The transformation layer allows the client to use our system transparently.

4. SECURITY ANALYSIS

In this section, we demonstrate that our system guarantees the data confidentiality for both secure requests over the encrypted KV store and secure queries over the encrypted local indexes. Regarding the proposed encrypted data store, each entry is a pair of a pseudo-random label and an encrypted data value. Even if all the nodes are compromised or collude, it only gives the total number of data. The data confidentiality and the relationships between the underlying data are still protected. Even if the encrypted data are relocated, the security strength will not be compromised.

Regarding the security of encrypted index, our design is built from the framework of SSE. Without querying, no information of index is known. And each query and the result will not give any information beyond the current query. We define the leakage function in our system, follow the simulation-based security definition [10, 19], and present security proof against adaptive chosen-keyword attacks. Explicitly, the leakage function of Build is:

$$L_1(\mathbf{C}) = (\{m_i | i \in [1,n]\}_n, |l^*|)$$

where $\mathbf{C}$ is the set of indexed attributes, m_i is the size of local index I_i, n is the number of nodes, and $|l^*|$ is the length of pseudo-random label. The leakage function for a given Query operation is given:

$$L_2(C) = (\{t_1^i, t_2^i\}_n, \{\{l^*, v^*\}_{c_i}\}_n, i \in [1,n])$$

For an attribute C, $\{t_1^i, t_2^i\}_n$ are the tokens for total n nodes respectively. Each Query reveals the number c_i of (l^*, v^*) pairs at each node. For q number of Query queries, the query pattern is defined:

$$L_3(\mathbf{Q}) = (M_{q\times q}, \{\{c_i\}_n\}_q)$$

$\mathbf{Q}$ is q number of adaptive queries. $M_{q\times q}$ is a symmetric bit matrix to trace the queries performed on the same attributes. $M_{i,j}$ and $M_{j,i}$ are equal to 1 if $t_1^i = t_1^j$ for $i, j \in [1,q]$. Otherwise, they are equal to 0. $\{c_i\}_n$ is a set of counters that record the number of values for the query attribute on each node. Accordingly, we present the security definition as below:

Definition 1. *Let $\Phi = (KGen, Build, Query)$ be the encrypted and distributed index construction. Given leakage L_1, L_2 and L_3, and an adversary $\mathcal{A}$ and a simulator $\mathcal{S}$, define the following experiments.*

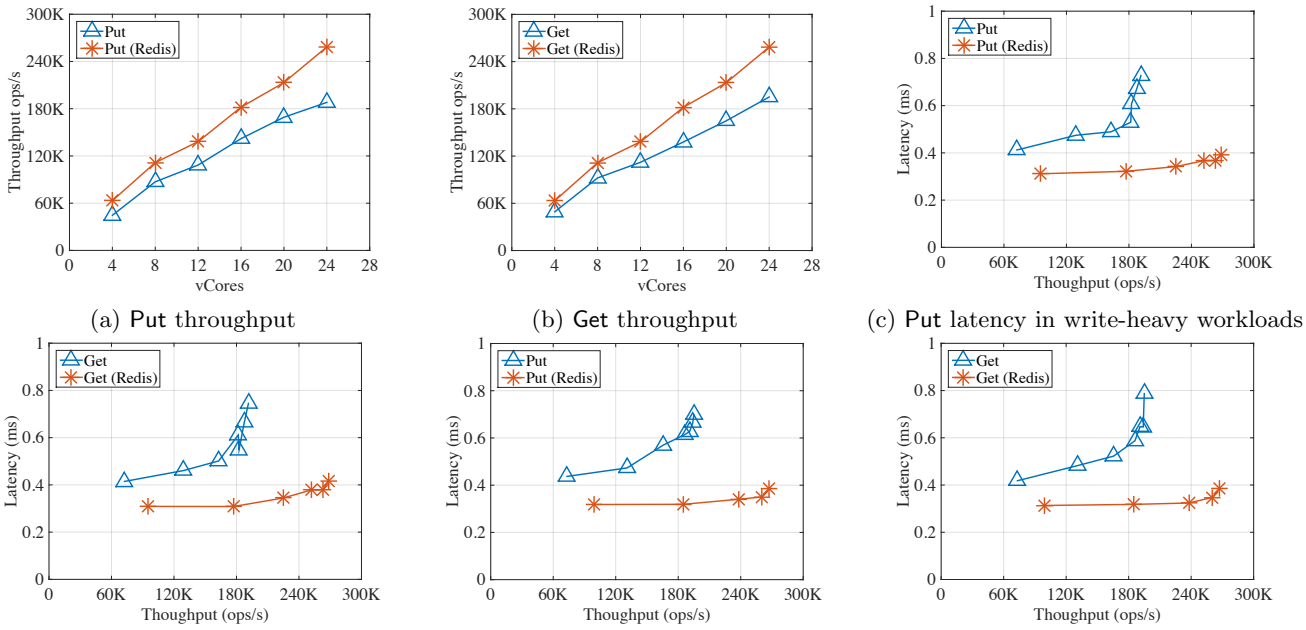
Real$_{\mathcal{A}}(k)$*: The client calls $KGen(1^k)$ to output a private key K. $\mathcal{A}$ selects a dataset $\mathbf{D}$ and asks the client to build $\mathbf{I}$ via Build. Then $\mathcal{A}$ performs a polynomial number of q queries, and asks the client for tokens and ciphertext. Finally, $\mathcal{A}$ returns a bit as the output.*

Ideal$_{\mathcal{A},\mathcal{S}}(k)$*: $\mathcal{A}$ selects $\mathbf{D}$. Then $\mathcal{S}$ generates $\mathbf{I}$ for $\mathcal{A}$ based on L_1. $\mathcal{A}$ performs a polynomial number of adaptive q queries. From L_2 and L_3, $\mathcal{S}$ returns the simulated ciphertext and tokens. Finally, $\mathcal{A}$ returns a bit as the output.*

Φ is (L_1, L_2, L_3)-secure against adaptive chosen-keyword attacks if for all probabilistic polynomial time adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $Pr[\mathbf{Real}_{\mathcal{A}}(k) = 1] - Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k) = 1] \leq negl(k)$, where $negl(k)$ is a negligible function in k.

Theorem 1. *Φ is (L_1, L_2, L_3)-secure against adaptive chosen-keyword attacks if SE is CPA-secure, and H, P, F_1, F_2, G_1, G_2 are secure PRF.*

Proof. Given L_1, the simulator $\mathcal{S}$ can generate the simulated index I' at each node, which is indistinguishable from the real index I. The number of the entries is identical. The size of the real entry and the simulated one is the same. But $\mathcal{S}$ generates random strings for each entry. From L_2, $\mathcal{S}$ can simulate the first query and the result. For the simulated local index of each node, $\mathcal{S}$ randomly selects the same number c_i of entries as the query on the real index. Here, the entry is selected by a random string t_1'. Then the token can be simulated such as: $t_2' = l^{*'} \oplus \beta'$, where $l^{*'}$ is identical to l^*, and mapped to the simulated entry. For the subsequent queries, if L_3 indicates the query appearing before, $\mathcal{S}$ will select exactly the same entries and use the same tokens generated before. Otherwise, $\mathcal{S}$ simulates the query and the result by following the procedure of the first query based on L_2. Due to the pseudo-randomness of PRF, the adversary cannot distinguish between the real index and the simulated one for each node, and the query tokens and the results from real indexes and the simulated ones. □



(a) Put throughput (b) Get throughput (c) Put latency in write-heavy workloads

(d) Get latency in write-heavy workloads (e) Put latency in read-heavy workloads (f) Get latency in read-heavy workloads

Figure 4: Put and Get performance evaluation

5. EXPERIMENTAL EVALUATION

5.1 System Implementation

We implement the system prototype and deploy it to Microsoft Azure. We create a cluster that consists of 6 `Standard_D12` instances as the nodes of the encrypted KV store and 9 `Standard_A4` instances as the clients of data applications. Each `Standard_D12` instance is assigned with 4 vCores, 28GB RAM and 200GB SSD, and each `Standard_A4` instance is assigned with 8 vCores, 14GB RAM, and 200GB SSD. They are installed with Ubuntu Server 14.04. Each instance runs with up to 100 threads to generate the workload for performance and scalability evaluation. Because the clients in our system have to perform cryptographic operations, we use more instances with more vCores to avoid the client bottleneck. In current prototype, we choose to cache the consistent hashing ring at the client for request routing. Thus, we do not need to select a specific node as the dispatcher. This mechanism is also used in DynamoDB [11] to save communication costs with the dispatcher and reduce the request latency. The ring can be updated periodically to keep the data partition fresh.

We set up Redis 3.0.5 at one node and create an Azure image to duplicate the Redis environment to other nodes. The secure transformation layer is currently deployed at the client via C++. The operations on the nodes are also implemented via C++. The cryptographic building blocks are implemented via OpenSSL. PRF is implemented via HMAC-SHA2, and symmetric encryption is implemented via AES/CBC-256. Besides, we preload totally $20,000,000$ data values (10 bytes for each) to our encrypted KV store before starting the experiment.

5.2 Performance Evaluation

The evaluation of our proposed KV store targets on request and query performance, system scalability, and security overhead. We will measure the Put and Get throughput, the Put and Get latency under different workloads, the cost of data relocation and index relocation when the system scales out, and the cost of the secure query over the encrypted local indexes. Specifically, we compare our Put and Get performance with directly using Redis to access plaintext data. And we also compare the query performance of the encrypted local indexes and the encrypted global index, where the latter treats the KV store as the black box and directly adopts the design in [5].

5.2.1 *Secure Put and Get*

To evaluate the scalability of our system, we first report the throughput for Put and Get respectively. To do so, the client threads are continuously increased to generate the workloads till the throughput stops increasing. By using different number of nodes, we capture the total number of handled requests for a duration of 100s to obtain the throughput when each of the nodes is fully loaded. From Figure 4-(a) and Figure 4-(b), the throughput of the encrypted KV store scales linearly along with the number of vCores, and achieves up to 1.9×10^6 Put/s and up to 2.0×10^6 Get/s. Comparing to non-encrypted Redis, the throughput of Put and Get have 27% and 28% loss respectively. The security overhead comes from the costs of HMAC-SHA2 and AES encryption. And the size of ciphertext (i.e., 32 bytes) is larger than the plaintext (i.e., 10 bytes), which degrades the performance.

Next, we measure the latency for Put and Get under different kinds of workloads to gain deeper understanding on the performance of the proposed encrypted KV store. Two typical workloads for data-intensive applications are simulated [9]. One is write-heavy with 50% Put and 50% Get requests. The other is read-heavy with 5% Put and 95% Get requests. The latency is measured by adding client threads until the throughput of each of 6 nodes stops increasing.

As shown in Figure 4-(c) and Figure 4-(d), Put and Get

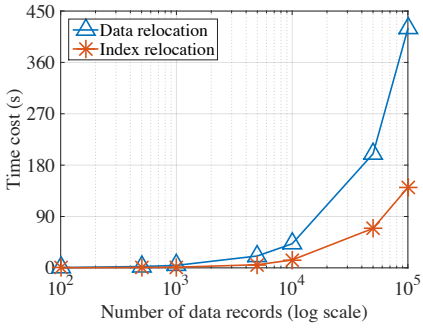


Figure 5: Time cost of adding 1 node

achieve millisecond latency for write-heavy workloads, less than 1ms per Put and per Get, when the average throughput on each node reaches around 31,912 ops/s. In Figure 4-(e) and Figure 4-(f), Put and Get also achieve low latency for read-heavy workloads, less than 1ms per Put and per Get, when the average throughput on each node reaches around 32,500 ops/s. The comparison to non-encrypted Redis indicates that the Get and Put latency is comparable. But we observe that the latency will increase gradually when the workloads reach 80% of maximum throughput. While for the non-encrypted Redis, the latency is still stable at the same scale of the workloads. We note that this impact on system performance is inevitable due to the strong protection of data confidentiality, especially when the size of plaintext data value is less than the size of block cipher. To achieve comparable performance in heavy workloads, more storage nodes will be required.

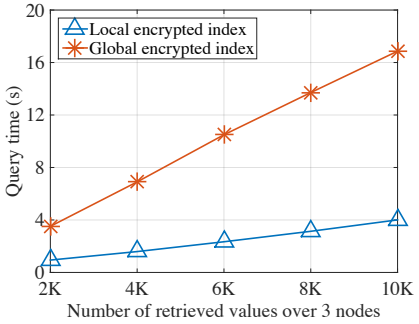
5.2.2 Scaling Out

When the growing workloads or the increasing amount of inserted data exceed the workload or the capacity of the node, our system can add new nodes smoothly, i.e., supporting incremental scaling over encrypted data and encrypted local indexes. In this experiment, the incremental scalability is evaluated by the time cost of adding one node into our system. Figure 5 depicts the cost for data relocation and index relocation respectively. Both data and index relocation costs increase linearly along with the number of affected data on the corresponding nodes. The time cost of data relocation is higher than the time cost of index relocation, e.g., 419s and 141s for 100,000 data values, respectively.

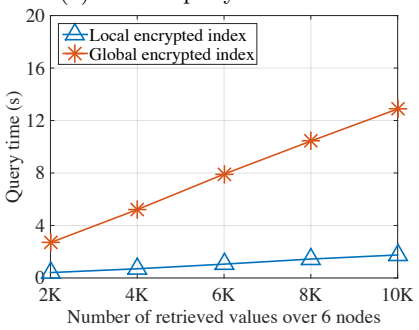
In this experiment, the affected indexes are directly built by executing Put requests, while the old one is set invalid and will be evicted later. This treatment optimizes the rebuilding process for index relocation. While for data relocation, the node will first scan the KV pairs and then execute Put requests for the data movement, so it takes longer time.

5.2.3 Secure Query Performance

To show the efficiency and the scalability of the proposed local index framework, we evaluate the secure query performance over encrypted local indexes. Specifically, we also compare our results with an encrypted global index, which is implemented from the encrypted dictionary design in [5]. We conduct a query for a given attribute with up to 10,000 data values. In Figure 6, we can see that the time cost increases linearly from 2,000 to 10,000 data values for the same number of nodes. But the query on the global index



(a) Secure query on 3 nodes



(b) Secure query on 6 nodes

Figure 6: Query performance over encrypted local indexes

is more than 3× slower than the local indexes. The reason is that the global index does not consider the data locality. To process a secure query, the node first locates where the matched data is stored, and it will establish another connection to the target node if the data is not at the current node. In that case, the target node will also establish a connection to the client to return the data. We note that even the number of tokens scales linearly with the number of the nodes in our design, while the global index design only requires the client to generate one token, the overhead introduced by the token transmission is much smaller than the overhead introduced by extra interactions across different nodes.

From the results in Figure 6, the secure query can be effectively handled in parallel, the time cost with 6 nodes is roughly half of the time cost with 3 nodes when querying the same number of encrypted data. That is because the data in a column are fully scrambled via PRF, they are evenly distributed across different nodes, and each node will process roughly the same number of entries in their local index. When more nodes are used, the time of retrieving the same number of data will be reduced. As being illustrated, it takes around 2s and 4s to retrieve 10,000 encrypted data values from 6 nodes and 3 nodes respectively. We believe that this advantages will be more apparently for other secure rich queries like counting, range, and aggregation.

On the contrary, the encrypted global index does not enjoy the benefits of the scalability. The query time does not decrease linearly when the number of nodes increases. As seen in Figure 6, it takes 13s and 17s to retrieve 10,000 encrypted data values from 6 nodes and 3 nodes respectively. For one reason, the encrypted global index needs to be processed in a serial way. For the other reason, the costs introduced by inter-node connections dominate the cost of secure query processing. In summary, the proposed encrypted lo-

cal indexes can support very efficient search queries over encrypted data, and outperform the encrypted global index if prior encrypted index designs are directly applied.

6. RELATED WORKS

Private DBMS systems: Practical encrypted DBMS systems recently are proposed and implemented to support rich queries without compromising the data confidentiality. Among which, one well-known system is CryptDB designed by Popa et al. [32]. It proposes to use onion encryption that encrypts the data in one or more layers for queries with different functionalities. Meanwhile, the underlying structure of DBMS keeps unchanged so as to support most of SQL queries. After that, Tu et al. develop MONOMI to improve the performance of CryptDB and allow analytical queries over encrypted data [37]. It is essentially built on CryptDB but includes a dedicated query planner to decide the optimized partitions of complex analytic queries, i.e., part of sub-queries or computation are conducted at the server, while the rest are done at the client.

Very recently, Pappas et al. present BlindSeer to achieve better query privacy and support arbitrary boolean queries [29]. It utilizes an encrypted Bloom filter tree as the back end storage, and the query is embedded into a Bloom filter. For each query, BlindSeer performs secure function evaluation via garbled circuits and oblivious transfer for the tree traversal. We note that all the above encrypted DMBSs focus on executing rich queries over encrypted data in the centralized DBMS systems, which are not designed for the scale and performance needs for modern data-intensive applications. They are a different line of work compared to our system.

Another kind of mechanism for protecting data and query privacy is to apply fragmentation and encryption in databases [1, 8]. Aggarwal et al. suppose data to be stored at two non-collude servers. By fragmenting the data, the sensitive associations are protected [1]. Under the same assumption, Ciriani et al. model the privacy constraints to represent the sensitivity of attributes and the associations, and improve the performance by minimizing the number of fragmentations [8]. On the other hand, Chow et al. propose a two-party computation model for privacy-preserving query on distributed databases [7]. In short, the above designs make a weak assumption such that the involved servers should not collude. While in our design, even if all the nodes collude, the data confidentiality will be still guaranteed.

Search over encrypted data: Our system design is also related to another line of works [5, 6, 10, 19] (to list a few) called searchable symmetric encryption (SSE), i.e., secure and efficient search schemes over encrypted data. Curtmola et al. improve the security definitions of SSE, and introduce new constructions with sublinear search time [10]. Then Kamara et al. propose a dynamic SSE scheme that supports adding and deleting files, and precisely capture the leakage of dynamic operations [19]. On the other hand, several attacks on SSE are proposed [4, 18], which exploit the search pattern and the access pattern to recover the queries and the document set. Yet, those attacks are all based on the assumption that the adversary knows partial information about the document and the queries.

We note that Chase et al. design a SSE scheme for arbitrarily structured data [6]. They introduce the notion structured encryption, and propose a construction that enables lookup queries on encrypted matrix-structured data with controlled disclosure. In particular, the data in a matrix (i.e., table) is encrypted and permuted in a pseudo-random fashion. But this construction is hardly updated and not scalable. Very recently, Cash et al. design and implement an efficient dynamic SSE to handle huge amount of data [5]. When a huge index stores in an external memory, the proposed hybrid packing approach addresses the locality of documents with same keywords, and improves I/O parallelism. Kuzu et al. propose an encrypted and distributed index for secure keyword search [22]. They build an encrypted inverted index, and then partition it into different regions.

Unfortunately, all the above designs cannot incrementally be scaled; that is, the rebuilding of the entire index is required when new nodes are added. Besides, even if they can be applied by treating the non-encrypted key-value store as the black box, a large number of connections and interactions would be introduced between the data node and the index node, because those designs do not consider the data and index locality specifically.

7. CONCLUSION

This paper presents a scalable, private, and searchable key-value store, which allows a client application to outsource a growing amount of data to public clouds with strong privacy assurance. The proposed underlying storage is an encrypted key-value store. It is secure and highly scalable. Data values are distributed evenly through a standard consistent hashing algorithm. Two basic APIs are accordingly provided for secure and fast data retrieval and update on single encrypted data value. To support search query over encrypted and distributed data, we then design the encrypted local indexes with the consideration on data and index locality, and give rigorous security analysis. We implement the system prototype, and deploy it to Microsoft Azure, and evaluate it comprehensively according to the performance metrics for distributed database systems. The results show that our system is practical, which introduces little security overhead compared to plaintext systems.

Acknowledgment

This work was supported in part by the Research Grants Council of Hong Kong (Project No. CityU 138513), National Natural Science Foundation of China (Project No. 61572412), US National Science Foundation under grant CNS-1464335, and a Microsoft Azure grant for research.

8. REFERENCES

[1] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret: A distributed architecture for secure database services. In *Proc. of CIDR*, 2005.

[2] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM TOS*, 9(4):12, 2013.

[3] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Proc. of CRYPTO*. Springer, 2011.

[4] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proc. of ACM CCS*, 2015.

[5] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very large databases: Data structures and implementation. In *Proc. of NDSS*, 2014.

[6] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Proc. of ASIACRYPT*, 2010.

[7] S. Chow, J.-H. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *Proce. of NDSS*, 2009.

[8] V. Ciriani, S. D. C. D. Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Combining fragmentation and encryption to protect privacy in data storage. *ACM TISSEC*, 13(3):22, 2010.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of the 1st ACM symposium on Cloud computing*, 2010.

[10] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proc. of ACM SOSP*, 2007.

[12] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. In *Proc. of ACM SIGCOMM*, 2012.

[13] FoundationDB. Foundationdb: Data modeling. Online at http://www.odbms.org/wp-content/uploads/2013/11/data-modeling.pdf, 2013.

[14] FoundationDB. A rock-solid, high performance database that provides nosql and sql access. Online at https://foundationdb.com/, 2015.

[15] F. Hahn and F. Kerschbaum. Searchable encryption with secure and efficient updates. In *Proc. of ACM CCS*, 2014.

[16] HBase. The hadoop database, a distributed, scalable, big data store. Online at http://hbase.apache.org, 2010.

[17] InfoWorld. The rise of the multimodel database. Online at http://www.infoworld.com/article/2861579/database/the-rise-of-the-multimodel-database.html, 2015.

[18] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proc. of NDSS*, 2012.

[19] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proc. of ACM CCS*, 2012.

[20] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, 1997.

[21] V. Kher and Y. Kim. Securing distributed storage: challenges, techniques, and systems. In *Proc. of the ACM workshop on Storage security and survivability*, 2005.

[22] M. Kuzu, M. S. Islam, and M. Kantarcioglu. Distributed search over encrypted big data. In *Proc. of ACM CODASPY*, 2015.

[23] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[24] M. Li, C. Qin, and P. P. Lee. Cdstore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proc. of USENIX ATC*, 2015.

[25] MongoDB. A cross-platform document-oriented database. Online at https://www.mongodb.com/, 2015.

[26] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proc. of ACM CCS*, 2015.

[27] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The ramcloud storage system. *ACM TOCS*, 33(3):7, 2015.

[28] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. of EUROCRYPT*, 1999.

[29] V. Pappas, B. Vo, F. Krell, S. Choi, V. Kolesnikov, A. Keromytis, and T. Malkin. Blind Seer: A Scalable Private DBMS. In *Proc. of IEEE S&P*, 2014.

[30] E. Pattuk, M. Kantarcioglu, V. Khadilkar, H. Ulusoy, and S. Mehrotra. BigSecret: A secure data management framework for key-value stores. In *Proc. of IEEE Int'l Conference on Cloud Computing*, 2013.

[31] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proc. of IEEE S& P*, 2013.

[32] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proc. of ACM SOSP*. ACM, 2011.

[33] Redis. An advanced key-value cache and store. Online at http://redis.io/, 2015.

[34] K. Ren, C. Wang, Q. Wang, et al. Security challenges for the public cloud. *IEEE Internet Computing*, 16(1):69–73, 2012.

[35] R. Schumacher. Datastax, graph, and the move to a multi-model database platform. Online at http://www.datastax.com/2015/02/datastax-graph-and-the-move-to-a-multi-model-database-platform, 2015.

[36] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[37] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proc. of the VLDB Endowment*, volume 6, pages 289–300, 2013.