# Multi-dimensional indexes for point and range queries on outsourced encrypted data

Sabrina De Capitani di Vimercati*, Dario Facchinetti†, Sara Foresti*,
Gianluca Oldani†, Stefano Paraboschi†, Matthew Rossi†, Pierangela Samarati*
* Università degli Studi di Milano, Italy – Email: *firstname.lastname*@unimi.it
† Università degli Studi di Bergamo, Italy – Email: *firstname.lastname*@unibg.it

***Abstract*—We present an approach for indexing encrypted data stored at external providers to enable provider-side evaluation of queries. Our approach supports the evaluation of point and range conditions on multiple attributes. Protection against inferences from indexes is guaranteed by clustering tuples in boxes that are then mapped to the same index values, so to ensure collisions for individual attributes as well as their combinations. Our spatial-based algorithm partitions tuples to produce such a clustering in a way to ensure efficient query execution. Query translation and processing require the client to store a compact map. The experiments, evaluating query performance and client-storage requirements, confirm the efficiency enjoyed by our solution.**

***Index Terms*—Data outsourcing, query execution, privacy**

## I. Introduction

The use of cloud providers for the storage of data is increasingly a necessity for most Big Data applications. Cloud providers offer storage with high levels of availability, reliability, scalability and performance, all associated with a relatively low cost, deriving from the large economies of scale enjoyed by cloud providers. A common concern when using cloud providers for the storage of sensitive data is the threat to the violation of confidentiality of the outsourced data [1]. Whereas violations of integrity or availability can produce effects that are visible to the customer, violations of confidentiality are usually hard to detect. This scenario is represented by the well-known *honest-but-curious* threat model [2]. The classical solution to this threat is represented by the use of encryption, so that the control of the physical representation of the data does not give access to the information content, as long as the cloud provider does not have access to the encryption key.

A major problem in this context is then enabling the fine-grained access and retrieval of data that are stored in encrypted form at the server. For real world applications, data are typically structured in relational tables, and access requests represented by SQL queries. In the past twenty years, the research and development community have dedicated significant effort to this problem, considering different lines of investigations. Possible approaches include: *i)* the use of searchable encryption (e.g., [3]), supporting the evaluation of conditions on encrypted data; *ii)* the use of trusted hardware components at the server (e.g., Intel SGX), offering a trusted execution environment residing at, but not accessible by, the server (e.g., [4]); *iii)* the association with the encrypted data of metadata working as indexes offering support for the evaluation of conditions (e.g., [5], [6]). All these approaches represent valid alternatives depending on the application scenario. The first two, while enjoying strong protection guarantees, suffer from a significant performance overhead, making them still not applicable in many practical scenarios. On the other hand, indexes, while applicable in practice, may suffer from a possible exposure to inferences, as they might leak information on the values behind them. The vulnerability of indexes typically resides in the frequencies of their occurrences, which can bear relationship with the plaintext values. Frequencies of both individual attributes values as well as combinations of them can be exposed to inference. A solution to this problem is guaranteeing indexes with collisions (i.e., mapping different plaintext values into the same index value), so to provide confusion and indistinguishability. Unfortunately, this is easier said than done, as constructing such an index requires addressing two (interconnected) aspects which are far from being trivial. First, an inevitable curse of dimensionality, while it can easily provide collisions and indistinguishability over one attribute, it is not so when multiple attributes need to be considered. The problem is complicated by the second aspect, which is the need to guarantee effectiveness of indexes (in terms of the limited overhead caused by spurious tuples returned to the clients due to collisions) and their efficiency (in terms of low performance overhead) for query execution. A third non trivial aspect is the need to limit the storage required at the client for (re)constructing indexes to translate queries on original plaintext data into queries on indexes at the server.

In this paper, we address all these problems and propose a *multi-dimensional index* (i.e., an index on multiple attributes) that is robust against inference exposure and, at the same time, performs well for query execution and requires limited storage at the client side. Our multi-dimensional index ensures not only that index values on individual attributes are guaranteed to appear at least a given number of times (i.e., no peculiar frequencies can be exploited for inference attacks) but that the same holds for their combination. In other words, each combination of index values enjoys the property of having at least a given number of occurrences. Besides providing protection against static inference attacks (which can no longer exploit frequencies of index values), our

This work was supported in part by the EC within the H2020 Program under projects MOSAICrOWN and MARSAL, by the Italian Ministry of Research within the PRIN program under project HOPE, and by JPMorgan Chase & Co under project “k-anonymity for AR/VR and IoT/5G”.

approach guarantees protection against dynamic observations from the storage provider, since tuples with the same index values are indistinguishable from one another (so are queries over them). The price to pay for such a protection is the overhead in query execution: being tuples with the same indexes indistinguishable one from the others since any query touching one of them would return all the others as well. It is therefore important to carefully group tuples for indexing so to limit the overhead in query execution and hence guarantee performance. While this can be trivial when only one attribute is to be indexed, it is far from being so (it is an NP-hard problem) when multiple attributes need to be indexed.

Our approach for index construction employs a spatial-based representation of tuples to be outsourced and a clever algorithm performing recursive cuts on such space, resulting in a partitioning of tuples for indexing. As confirmed by the experimental evaluation, our proposal provides for effective and efficient query evaluation, enjoying limited overhead and limited storage requirements at the client side.

## II. Basic concepts

We frame our work in the context of relational database systems (which are still at the basis of almost all applications). We then illustrate our approach with reference to the outsourcing of a relation $r$ defined over schema $R(a_1, \ldots, a_n)$, where each attribute $a_j$ is defined over a domain $\mathtt{d}(a_j)$, for $j = 1, \ldots, n$. In the following, we use notation $\mathtt{val}(a_j)$ to denote the set of values of attribute $a_j$ stored in $r$ (i.e., $\mathtt{val}(a_j)$ = SELECT DISTINCT $a_j$ FROM $R$). As an example, Figure 1(a) illustrates a relation $r$ with three attributes: `Name`, `State`, and `Age`. Here, $\mathtt{d(Age)}=\{0, \ldots, 120\}$ and $\mathtt{val(Age)}=\{27, 30, 35, 38, 42, 45, 50\}$. To protect the confidentiality of data and make them non-intelligible to the storage provider, the owner encrypts the relation at the tuple level before outsourcing it, using a symmetric encryption scheme with a key shared with authorized users only. Queries on the encrypted relation are supported via a set of *indexes* associated with a set $\mathcal{I}=\{a_1, \ldots, a_l\}\subseteq R$ of attributes in the original relation on which conditions need to be evaluated in the execution of queries (`State` and `Age` for our running example). An encrypted and indexed relation is formally defined as follows.

*Definition 2.1 (Encrypted and indexed relation):* Let $r$ be a relation over schema $R(a_1, \ldots, a_n)$, and $\mathcal{I} = \{a_1, \ldots, a_l\} \subseteq R$ be a subset of the attributes in $R$. The *encrypted and indexed* version of $r$ is a relation $r^e$ over schema $R^e(et, i_1, \ldots, i_l)$ where $\forall t \in r$, $\exists t^e \in r^e$ such that $t^e[et]$=$E_k(t)$, with $E_k$ a symmetric encryption function with key $k$, and $t^e[i_j]$ the index value derived from $t[a_j]$, $j = 1, \ldots, l$.

According to this definition, the encrypted and indexed version $r^e$ of relation $r$ has an attribute $et$, which is the encrypted representation of the tuples in the plaintext relation, and an attribute $i_j$, which is the index for attribute $a_j$ in $\mathcal{I}$, $j = 1, \ldots, l$. Figure 1(d) illustrates an example of encrypted and indexed version of the relation in Figure 1(a), where `State` and `Age` are indexed. For simplicity, in the example we use Greek letters to represent index values.

Our goal is to compute a multi-dimensional index that is effective and efficient for the execution of queries with support for equality ($=$) and range ($>, \geq, <, \leq$) conditions.

## III. Multi-dimensional tuple partitioning

Our approach for partitioning tuples for indexing employs an algorithm similar to the one used by the Mondrian anonymization algorithm [7], [8]. While similar, our algorithm bears differences to accommodate the fact that we need to cluster tuples to produce obfuscated indexes performing well for query evaluation (in contrast to cluster tuples for semantically meaningful generalization). Our partitioning process works then in a multi-dimensional space, with one dimension for each indexed attribute, and where tuples correspond to points in the multi-dimensional space where their coordinates correspond to the values of the indexed attributes in the tuples. Figure 1(b) shows the two-dimensional representation for the indexing of attributes `State` and `Age` of the relation in Figure 1(a). Since more tuples can have the same values for the indexed attributes, a point in the multi-dimensional space can correspond to more than one tuple, which is represented in the figure with the number of occurrences associated with it (omitted in our example since it is always equal to 1).

To construct the multi-dimensional space on which the algorithm operates, by partitioning tuples in *boxes* of at least $\mathsf{b}$ tuples, we classify attributes to be indexed into two categories:

- *continuous* attributes (e.g., `Age` in Figure 1(a)), characterized by a total order relationship on their domain, and on which range conditions need to be supported;
- *nominal* attributes (e.g., `State` in Figure 1(a)), which do not have a semantic order in their domain and hence on which only equality conditions make sense.

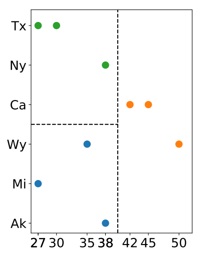
When partitioning tuples in boxes, care must be taken to put as much as possible tuples with the same values for an attribute in the same box. Also, for continuous attributes, close values should fall as much as possible in the same space. (Note that this might intrinsically not be possible for all attributes.) Consistently with these observations, values of continuous attributes are considered in their natural (we assume increasing) order along the axis of their dimension, while values of nominal attributes are considered in increasing order of their relative frequencies in the tuples.

The partitioning process works recursively, cutting, at each step, a space (the whole space in the first step) with respect to a selected attribute and a value in its domain as threshold. The cut divides the space in two sub-spaces, each containing the points (i.e., the tuples) falling on its side of the cut. The process is recursively repeated on each of the two resulting sub-spaces, and terminates when any further cut would generate a partition with less than $\mathsf{b}$ tuples. At each step, the attribute chosen for the cut is the one that, in the considered space, has the maximum *span*. For continuous attributes, the span is the distance between the minimum and maximum value that the tuples in the (sub-)space assume. For nominal attributes,

$r$

| | Name | State | Age |
|---|---|---|---|
| $t_1$ | Ada | Ak | 38 |
| $t_2$ | Bob | Mi | 27 |
| $t_3$ | Coy | Wy | 35 |
| $t_4$ | Dan | Ca | 42 |
| $t_5$ | Eve | Ca | 45 |
| $t_6$ | Fay | Wy | 50 |
| $t_7$ | Gil | Ny | 38 |
| $t_8$ | Hal | Tx | 30 |
| $t_9$ | Ian | Tx | 27 |

(a)



(b)

| | State | Age |
|---|---|---|
| $t_1$ | AkMiWy | [27,38] |
| $t_2$ | AkMiWy | [27,38] |
| $t_3$ | AkMiWy | [27,38] |
| $t_4$ | CaWy | [42,50] |
| $t_5$ | CaWy | [42,50] |
| $t_6$ | CaWy | [42,50] |
| $t_7$ | NyTx | [27,38] |
| $t_8$ | NyTx | [27,38] |
| $t_9$ | NyTx | [27,38] |

(c)

$r^e$

| | et | $i_{\texttt{State}}$ | $i_{\texttt{Age}}$ |
|---|---|---|---|
| $t^e_1$ | but6yv | $\omega$ | $\alpha$ |
| $t^e_2$ | lmoe!. | $\omega$ | $\alpha$ |
| $t^e_3$ | p4?llq | $\omega$ | $\alpha$ |
| $t^e_4$ | gbS1.X | $\gamma$ | $\beta$ |
| $t^e_5$ | c493pw | $\gamma$ | $\beta$ |
| $t^e_6$ | WD.23b | $\gamma$ | $\beta$ |
| $t^e_7$ | Q.co43 | $\tau$ | $\alpha'$ |
| $t^e_8$ | de31As | $\tau$ | $\alpha'$ |
| $t^e_9$ | xMe1!K | $\tau$ | $\alpha'$ |

(d)

Map$_{\texttt{Age}}$

| Bucket | Count |
|---|---|
| [27,38] | 2 |
| [42,50] | 1 |

(e)

Map$_{\texttt{State}}$

| Value | Buckets$^\sigma$ |
|---|---|
| Ak | AkMiWy |
| Ca | CaWy |
| Mi | AkMiWy |
| Ny | NyTx |
| Tx | NyTx |
| Wy | AkMiWy, CaWy |

(f)

Fig. 1: Plaintext relation (a), its spatial representation (b), partitioning (c), encrypted and indexed relation (d), and maps for attribute Age (e) and State (f)

it is the number of distinct values that the tuples in the (sub-)space assume. For instance, with reference to the relation in Figure 1(a), the span for attribute Age is $50 - 27 = 23$, while the span for attribute State is 6. The value chosen as threshold for the cut is the median for continuous attributes, and the value that splits the (sub-)space in two sub-spaces with nearly 50% of the tuples each for nominal attributes. Note that cuts change the relative frequency of values in the generated sub-spaces, and hence also the order in which values for nominal attributes are considered on their axis.

## IV. Index construction

At the end of the partitioning process, the tuples in $r$ are grouped in non-overlapping boxes (i.e., disjoint groups of tuples whose union corresponds to $r$) such that each box contains at least b tuples. The dotted lines in Figure 1(b) denote the cuts performed and hence the resulting boxes for our example. Two cuts have been performed, resulting in three boxes, each containing three tuples.

Intuitively, boxes determine the tuples that will be mapped to the same combination of index values. In other words, for each indexed attribute, all the values that fall in the same box will be mapped to the same index values. Since this applies to all indexed attributes, this implies that all tuples in the same box will be mapped to the same combination of index values. In the following, we use notation $\mathcal{B}$ to denote the set of all boxes, and $B_j{\in}\mathcal{B}$ to denote the $j$-th box. Also, for each box $B{\in}\mathcal{B}$ and attribute $a{\in}\mathcal{I}$, we denote with $B[a]$ the set of values of $a$, called *bucket*, covered by $B$. A bucket is expressed as an interval for continuous attributes and as a set of values for nominal attributes. Formally, for each box $B$ and attribute $a$:

- $B[a]$ = $[v$,$v']$ such that $v$ = $\min\{t[a] \mid t \in B\}$ and $v'$ = $\max\{t[a] \mid t \in B\}$, if $a$ is a continuous attribute;
- $B[a]$ = $\{t[a] \mid t \in B\}$, if $a$ is a nominal attribute.

Figure 1(c) reports the buckets of the three boxes corresponding to the partitioning in Figure 1(b). For readability, we represent each set as a string composed of all its elements, that is, AkMiWy stands for set $\{$Ak,Mi,Wy$\}$.

Note that, different boxes might be associated with the same bucket for one or more of their attributes. Formally, we may have $B_x[a]$ = $B_y[a]$, with $x \neq y$. For instance, in our example, box $B_1$ containing the first three tuples and box $B_3$ containing the last three tuples have $B_1$[Age] = $B_3$[Age] = [27,38].

Since, for each attribute, we want index values in different boxes to be different, the generation of indexes cannot depend only on the bucket. To map the same bucket of different boxes to different index values, we combine buckets for their indexing with a salt. This is formalized by the following definition.

*Definition 4.1 (Index function):* Let $r$ be a relation, $a{\in}\mathcal{I}$ be an indexed attribute, $\mathcal{B}$ be the set of boxes of relation $r$, and $h_k$ be a cryptographic hash function with key $k$. An *index function* for attribute $a$ is a function $\iota_a$:$\mathcal{B}{\rightarrow}I_a$ such that:

- $\forall B \in \mathcal{B}$, $\iota_a(B)$=$h_k(B[a]||\sigma)$, with $\sigma$ a randomly generated salt;
- $\forall a, a' \in \mathcal{I}$, $\forall B, B' \in \mathcal{B}$, with $\iota_a(B)$=$h_k(B[a]||\sigma)$, $\iota_{a'}(B')$=$h_k(B'[a']||\sigma')$, if $B[a] = B'[a']$ and ($a \neq a'$ or $B \neq B'$), then $\sigma \neq \sigma'$.

Indexes are then computed as the result of a cryptographic hash function on the concatenation of the bucket to be indexed and a salt. The second bullet in the definition dictates the use of a different salt for buckets that are equal but refer to different attributes (i.e., dimensions) or for a same bucket that appears in different boxes for the same attribute. Satisfaction of such a condition is guaranteed by generating salts using a pseudo-random generation function with a different seed for each attribute and using a different salt in the sequence for different occurrences of a same bucket. Hence, different attributes will be associated with a different sequence of randomly generated salts. For each attribute $a{\in}\mathcal{I}$, we denote with $\sigma_a(j)$ the $j$-th salt generated by function $\sigma$ with the seed of attribute $a$. Each bucket $B_x[a]$ is then associated with the $j$-th salt $\sigma_a(j)$, with $j-1$ the number of boxes $B_y \in \mathcal{B}$ such that $B_x[a]{=}B_y[a]$ and $y < x$. For instance, with reference to the example in Figure 1, bucket $B_3$[Age] is combined with salt $\sigma_{\texttt{Age}}(2)$ since $B_1$[Age]=$B_3$[Age] and $1 < 3$. Figure 1(d) shows the encrypted and indexed version of the plaintext relation in Figure 1(a). Here, combinations of index values $\langle\omega,\alpha\rangle$, $\langle\gamma,\beta\rangle$, and $\langle\tau,\alpha'\rangle$ are those computed for the three boxes in Figure 1(c). Indexes $\alpha$ and $\alpha'$ represent different salted versions of the same bucket (i.e., [27,38]).

## V. Client-side maps

The process illustrated in the previous sections enables the creation of indexes to be associated with the tuples in the plaintext relation to be outsourced. The encrypted and indexed relation (Def. 2.1) outsourced to the storage provider will then have, for each tuple in the original plaintext relation, its encrypted version and the values of the indexes computed as illustrated (Def. 4.1). For simplicity, in our examples we maintain tuples in the same order in the original and outsourced relations, but clearly tuples should be shuffled before upload them to the storage provider.

The next problem is the definition of the information to be stored at the client side to enable the translation of queries on the plaintext relation into queries on the encrypted and indexed relation that can then be executed at the storage provider. According to Def. 4.1, such information comprises:

- the cryptographic hash function $h$ along with the corresponding key $k$;
- the function $\sigma_a$ used for salt generation;
- a map, denoted $\text{Map}_a$, enabling the translation of plaintext attribute values into index values.

While the first two bullets only require a client to memorize one function, the third one, requiring attributes' maps, needs more consideration. Being maps stored client side, it is important to maintain them compact, to limit the storage needed at the client. As we will show in Section VII, our maps enjoy such compactness. We maintain attributes' maps in a compact form as follows.

**Continuous attributes.** For each continuous attribute $a$, $\text{Map}_a$ is a set of pairs (`Bucket`,`Count`), reporting the buckets in which the attribute has been divided and, for each bucket, the number of boxes in which it appears. Formally, $\text{Map}_a = \{\langle B[a], c\rangle \mid B \in \mathcal{B}, c = |\{B' \in \mathcal{B} \mid B'[a] = B[a]\}|\}$.

The counter associated with each bucket gives the number of distinct index values corresponding to the bucket and hence the number of salts to be used for reconstructing such indexes for query translation. Figure 1(e) illustrates the map for attribute `Age` that contains the information about the two buckets resulting from partitioning (i.e., [27,38] and [42,50]). Bucket [27,38] has 2 occurrences and the sequence of salts used in the generation of the corresponding index values is $\sigma_{\texttt{Age}}(1)$ and $\sigma_{\texttt{Age}}(2)$. Hence, the index values corresponding to bucket [27,38] are $h_k$([27,38]$||\sigma_{\texttt{Age}}(1)$)=$\alpha$ and $h_k$([27,38]$||\sigma_{\texttt{Age}}(2)$)=$\alpha'$ (see Figure 1(d)).

**Nominal attributes.** For each nominal attribute $a$, $\text{Map}_a$ is a set of pairs (`Value`,`Buckets`$^\sigma$) for each value $v$ in $a$, the buckets $B[a]$ that include $v$, concatenated with the corresponding salt value. Formally, $\text{Map}_a = \{\langle v, \{(B[a] \mid\mid \sigma)\}\rangle \mid v \in \texttt{val}(a), B \in \mathcal{B} : v \in B[a], \iota_a(B) = h_k(B[a] \mid\mid \sigma)\}$.

Figure 1(f) illustrates the map for attribute `State`. For simplicity, in the figure, noting that in our example all buckets have only one occurrence, and therefore only one salt is to be used, we report the unsalted bucket values.

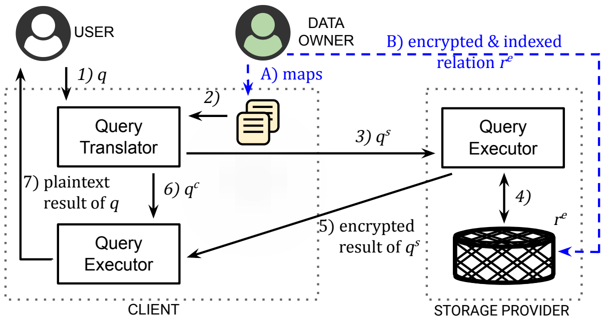


Fig. 2: Query execution process

## VI. Query translation and execution

Once the (encrypted and indexed) relation has been outsourced, it will reside at the external storage provider and only the information needed for query translation (i.e., the maps) will be stored at the client. Each query $q$ on $R$, formulated at the client side, will then need to be translated into a query $q^s$ operating on indexes at the provider side. The retrieved result (encrypted tuples whose indexes satisfy $q^s$) will be decrypted and query $q^c$ will be executed to eliminate possible spurious tuples, where $q^c$ is the same as $q$ but executed on the decryption of the result of $q^s$ instead of $R$. Figure 2 illustrates the overall architecture and query execution process.

Translating $q$ into $q^s$ requires mapping each of the conditions appearing in its WHERE clause into a condition on indexes. We illustrate such a mapping depending on whether the condition is on a continuous or nominal attribute.

**Continuous attribute.** Continuous attributes support both *point* (i.e., equality) as well as *range* conditions. Conditions can then be of the form "$a$ *op* $v$" or "$a$ BETWEEN $v_x$ AND $v_y$", with $a \in \mathcal{I}$, $v$,$v_x$,$v_y \in \texttt{d}(a)$, and *op* $\in \{>, \geq, <, \leq\}$.

To illustrate the mapping, we consider a general form capturing all the cases above and illustrate the mapping of condition "$a \in \texttt{Range}$", where `Range` is an (open or closed) interval specified by two values $v_l$ and $v_r$, with $v_l \leq v_r$. Such a general form captures all the conditions above, by simply considering $v_l = v_r = v$ for point conditions; $v_l$ the lowest value in $\texttt{val}(a)$ for $<$ or $\leq$ conditions; $v_r$ the highest value in $\texttt{val}(a)$ for $>$ or $\geq$ conditions; and the interval open on the left (right, resp.) side if values equal to $v_l$ ($v_r$, resp.) should be excluded. For instance, Age<30, is equivalent to Age∈[27,30).

A condition of the form "$a \in \texttt{Range}$" is translated into a condition on $a$'s index $i_a$, requesting it to be in the set of index values to which the $a$'s buckets intersecting `Range` have been mapped, that is, in condition:

- $i_a$ IN $(h_k(b||\sigma_a(j))$ s.t. $\langle b, c\rangle \in \text{Map}_a$, $b \cap \texttt{Range} \neq \emptyset$, $j = 1, \ldots, c)$

Here $\langle b, c\rangle \in \text{Map}_a$ denotes the different buckets in $\text{Map}_a$, $b \cap \texttt{Range} \neq \emptyset$ restricts the consideration to the ones intersecting `Range`, and $c$ expresses the number of salts to be used for each bucket $b$ (i.e., the number of distinct index values to which the bucket has been mapped).

For instance, consider attribute `Age` and its map in Figure 1(e). Condition "`Age`=30" is translated to "$i_{\texttt{Age}}$ IN ($\alpha$, $\alpha'$)", with $\alpha$=$h_k$([27,38]$||\sigma_{\texttt{Age}}$(1)) and $\alpha'$=$h_k$([27,38]$||\sigma_{\texttt{Age}}$(2)). Condition "`Age`>39" is translated to "$i_{\texttt{Age}}$ IN ($\beta$)", with $\beta$=$h_k$([42,50]$||\sigma_{\texttt{Age}}$(1)).

**Nominal attribute.** Nominal attributes support the evaluation of point conditions only. A condition"$a$=$v$" is translated into a condition on $a$'s index $i_a$, requesting it to be in the set of index values to which $v$ has been mapped, that is, in condition:

- $i_a$ IN($\{h_k(set_j)$ s.t. $m\in$Map$_a$, with $m$[`Value`]=$v$ and $set_j \in m$[`Buckets`$^\sigma$], $j = 1, \ldots, |m$[`Buckets`$^\sigma$]$|\}$)

For instance, consider attribute `State` and its map in Figure 1(f). Condition "`State`='Wy'" is translated to "$i_{\texttt{State}}$ IN ($\omega$, $\gamma$)", with $\omega$=$h_k$(AkMiWy) and $\gamma$=$h_k$(CaWy). Condition "`State`='Ca'" is translated to "$i_{\texttt{State}}$ IN ($\gamma$)".

## VII. Implementation and experiments

We built a prototype in Python that implements both the generation of the encrypted and indexed relation and the query evaluation process. We then tested our approach on the PUMS USA ACS 2019 dataset, containing 3.2M tuples [9]. The dataset schema includes two nominal (`State` and `Occupation`) and two continuous (`Age` and `Income`) attributes. Our experiments analyze the overhead in query execution and the size of the client-side maps.

**Indexing and encryption.** We realized a distributed version of the partitioning process illustrated in Section III. Our tool uses the scalable Apache Spark platform and parallelizes the indexing process relying on an arbitrary number of workers. Partitions of the dataset are assigned to the workers, which independently apply the indexing process on the tuples assigned to them. Each worker computes index values (Def. 4.1) using the Blake2b hash function and a different 16-byte salt for each attribute. The encrypted tuple (attribute *et*) is computed using XSalsa20 with Poly1305 MAC (to guarantee both confidentiality and integrity) with a 32-byte high-entropy key. The tool also generates a fresh nonce for each encryption invocation to provide indistinguishability of tuples with the same values for all encrypted attributes. The encryption functions are implemented by *PyNacl*. The encrypted and indexed relations generated by each worker are uploaded, in randomized order, to a containerized PostgreSQL DBMS.

**Query translation.** Each client side query $q$ is parsed using *sqlparse*, a SQL parser for Python, and translated as described in Section VI. Query $q^s$ is submitted to the PostgreSQL DBMS hosted at the storage provider, and its result is decrypted and checked for integrity by the client. The client executes query $q^c$ on an in-memory SQLite DB to filter spurious tuples and project the attributes of interest.

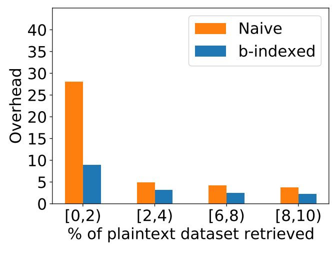
**Query overhead.** We compared our solution with a naive approach that builds boxes of b tuples, each by ordering tuples according to the values of a sequence of attributes and then splitting the ordered dataset in boxes of b contiguous tuples. We run two kinds of query: *1)* point queries for each attribute $a$ in the dataset schema (i.e., `State`, `Occupation`, `Age`, `Income`), and each value $v$ in `val`($a$); *2)* range queries for attribute `Age` and for each range [$v_i$,$v_j$] of values in `val(Age)`. Figure 3 compares the overhead in query execution for the naive approach and for our, called b-indexed, approach. Figures 3(a,c,e) refer to point queries and Figures 3(b,d,f) refer to range queries. The overhead is measured in terms of the average ratio between the number of tuples returned by the query on index $i_a$ and the original query on $a$, considering b with values 10, 25, and 50. For point queries, such an overhead is measured depending on the selectivity (%age of tuples returned) of the original query on the plaintext data (plotted on the $x$ axis). We assumed each query to be issued as many times as the frequency of the requested value. For range queries, Figures 3(b,d,f), the overhead is measured depending on the percentage of domain values covered by the range condition (plotted on the $x$ axis). As visible from the figures, our approach largely outperforms the naive one, whose overhead compared to ours is between 1.5x and 3x for point queries and between 5x and 10x for range queries.

It is interesting to note the limited overhead provided by our b-indexed approach, which is much smaller than the value of b. Such limited overhead is to be particularly appreciated in comparison with alternatives for privacy-aware query execution that suffers at least 30x overhead in size of each access (in some cases a more than 1000x overhead) [10], which grows significantly when indexes are introduced.

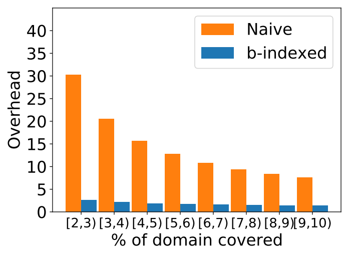
**Local data structure.** Figure 4(a) illustrates the size of the maps stored at the client side, varying the number of tuples in the dataset between 0.5 and 3 millions and b equal to 10, 25, and 50. As visible from Figure 4(b), our maps require in almost all the analyzed configurations less than 1 byte per tuple. Note that the storage space per tuple required by the map decreases as the number of tuples increases.

## VIII. Related work

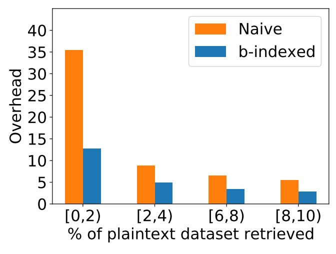
Several research efforts have addressed the problem of supporting queries on outsourced encrypted data through the definition of indexing techniques or specific cryptographic schemes (e.g., [3], [5], [6], [11], [12]). Such solutions must be applied with care due to the possible information leakage (e.g., [13], [14]). The definition of efficient solutions robust against inferences also depends on the specific queries to be supported. In particular, the support of range queries requires to define techniques that consider the order relationship characterizing the domain of the attributes on which queries have to be executed, which can complicate the definition of such techniques (e.g., [5], [15], [16]). While sharing with our approach the goal of supporting queries over encrypted data, these solutions operate on a single attribute. Our approach instead is based on a multi-dimensional interpretation of the dataset that allows the definition of indexes over multiple attributes. The problem of indexing multi-dimensional datasets has been already considered and resulted in the definition of multi-dimensional indexes for supporting queries with conditions on



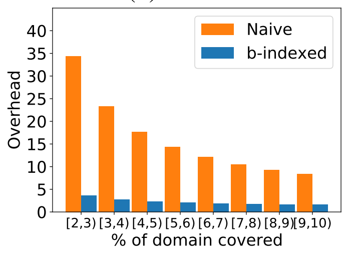
(a) b=10



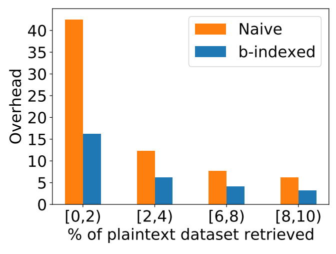
(b) b=10



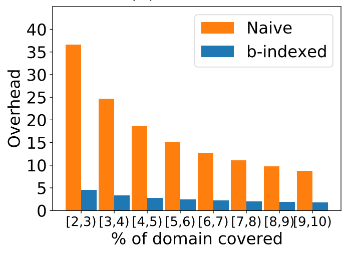
(c) b=25



(d) b=25

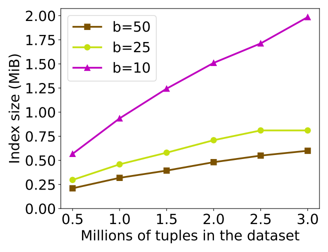


(e) b=50

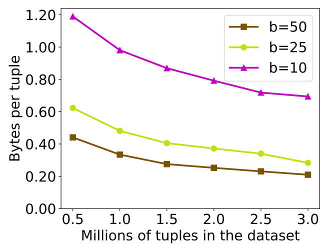


(f) b=50

Fig. 3: Point (a,c,e) and range (b,d,f) queries overhead



(a)



(b)

Fig. 4: Absolute (a) and relative (b) size of the maps

multiple attributes (e.g., [17]). These solutions, however, differ from our since they define one index only for the whole set of attributes/dimensions considered. Our approach defines a multi-dimensional index with a component for each attribute, considering the intrinsic multi-dimensional nature of relations.

A line of work close to our is represented by approaches aimed at supporting query evaluation over data organized in multiple relations and/or fragments that cannot be joined by non-authorized subjects (e.g., [18]–[20]). These solutions reduce the precision of join operations introducing a degree $k$ of uncertainty (i.e., it is never possible to reconstruct a tuple in the join result with uncertainty lower than $1/k$). Protection is obtained by grouping tuples in the relations/fragments and performing joins at the group (in contrast to tuple) level.

## IX. Conclusions

We have addressed the problem of outsourcing encrypted data to external providers and defining indexes over them for enabling query execution. Our approach to index construction, working on multiple attributes, guarantees protection against inferences, while providing effective and efficient query execution, with support for both point and range conditions. Our experimental evaluation on a large publicly available dataset confirms the validity of our approach and therefore its applicability in practical scenarios.

## References

[1] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati, "Dynamic allocation for resource protection in decentralized cloud storage," in *Proc. of GLOBECOM*, Waikoloa, HI, USA, Dec. 2019.

[2] P. Samarati and S. De Capitani di Vimercati, "Cloud security: Issues and concerns," in *Encyclopedia on Cloud Computing*, S. Murugesan and I. Bojanova, Eds. Wiley, 2016.

[3] G. Poh, J. Chin, W. Yau, K. Choo, and M. Mohamad, "Searchable symmetric encryption: Designs and challenges," *ACM CSUR*, vol. 50, no. 3, May 2017.

[4] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "SGX-BigMatrix: A practical encrypted data analytic framework with trusted processors," in *Proc. of ACM CCS*, Dallas, TX, USA, Oct./Nov. 2017.

[5] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational DBMSs," in *Proc. of ACM CCS*, Washington, DC, USA, Oct. 2003.

[6] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proc. of ACM SIGMOD*, Madison, WI, USA, June 2002.

[7] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati, "Scalable distributed data anonymization," in *Proc. of IEEE PerCom*, Mar. 2021.

[8] K. LeFevre, D. DeWitt, and R. Ramakrishnan, "Mondrian multidimensional $k$-anonymity," in *Proc. of ICDE*, Atlanta, GA, USA, Apr. 2006.

[9] S. Ruggles, S. Flood, R. Goeken, J. Grover, E. Meyer, J. Pacas, and M. Sobek, "IPUMS USA: Version 10.0 [dataset]," Minneapolis, MN: IPUMS, 2020, https://doi.org/10.18128/D010.V10.0.

[10] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward," in *Proc. of ACM CCS*, Denver, CO, USA, Oct. 2015.

[11] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proc. of SOSP*, Cascais, Portugal, Oct. 2011.

[12] H. Wang and L. Lakshmanan, "Efficient secure query evaluation over encrypted XML databases," in *Proc. of VLDB*, Seoul, Korea, Sept. 2006.

[13] A. Ceselli, E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Modeling and assessing inference exposure in encrypted databases," *ACM TISSEC*, vol. 8, no. 1, Feb. 2005.

[14] M. Naveed, S. Kamara, and C. Wright, "Inference attacks on property-preserving encrypted database," in *Proc. of ACM CCS*, Denver, CO, USA, Oct. 2015.

[15] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical private range search revisited," in *Proc. of ACM SIGMOD*, San Francisco, CA, USA, June–July 2016.

[16] H. Van Tran, T. Allard, L. d'Orazio, and A. El Abbadi, "FRESQUE: A scalable ingestion framework for secure range query processing on clouds," in *Proc. of EDBT*, Nicosia, Cyprus, Mar. 2021.

[17] B. Wang, Y. Hou, M. Li, H. Wang, and H. Li, "Maple: Scalable multi-dimensional range search over encrypted cloud data with tree-based index," in *Proc. of ACM ASIACCS*, Kyoto, Japan, June 2014.

[18] G. Cormode, D. Srivastava, T. Yu, and Q. Zhang, "Anonymizing bipartite graph data using safe groupings," *PVLDB*, vol. 1, no. 1, Aug. 2008.

[19] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Fragments and loose associations: Respecting privacy in data publishing," *PVLDB*, vol. 3, no. 1, Sept. 2010.

[20] X. Xiao and Y. Tao, "Anatomy: Simple and effective privacy preservation," in *Proc. of VLDB*, Seoul, Korea, Sept. 2006.