Flash: Efficient, Stable and Optimal K-Anonymity

Florian Kohlmayer†‡, Fabian Prasser†‡, Claudia Eckert∗, Alfons Kemper∗ and Klaus A. Kuhn†
∗Technische Universität München, Department of Computer Science, 85748 Garching, Germany
†Technische Universität München, University Medical Center (Klinikum rechts der Isar), 81675 München, Germany

Abstract—K-anonymization is an important technique for the de-identification of sensitive datasets. In this paper, we briefly describe an implementation framework which has been carefully engineered to meet the needs of an important class of k-anonymity algorithms. We have implemented and evaluated two well-known algorithms within this framework and show that it allows for highly efficient implementations. Regarding their runtime behaviour, we were able to closely reproduce the results from previous publications but also found some algorithmic limitations. Furthermore, we propose a new algorithm that achieves very good performance by implementing a novel strategy and exploiting different aspects of our implementation framework. In contrast to the current state-of-the-art, our solution offers algorithmic stability, with execution time being independent of the actual representation of the input data. Experiments with different real-world datasets show that our approach clearly outperforms previous algorithms.

I. INTRODUCTION

The amount of data collected about individuals is proceeding at an ever-increasing rate [1]. A number of incidents (e.g., [2]–[4]) have shown that simply removing all directly identifying information (e.g., a person’s name) is not sufficient to protect an individual’s privacy. In addition, detailed personal data of high quality is often required for analyses. In this context, anonymization is an important building block for balancing an individual’s privacy and the need for fine-grained data collections. To this end k-anonymity is a widespread technique. The basic idea is to protect a dataset against re-identification by transforming so called quasi-identifiers, which are attributes that could be used in a linkage attack. In this attack presumably anonymized data is linked to additional identified data, which can result in identity disclosure [4]. A dataset is k-anonymous if each data item can not be distinguished from at least k − 1 other data items [5]. An example dataset with quasi-identifiers age, gender and zipcode as well as a two-anonymous transformation are shown in Figure 1. Without loss of generality we assume a tabular data structure. Alternative methods have been proposed (e.g., Differential Privacy [6]), but k-anonymization is still important in many domains, e.g., biomedical research [7].

II. RELATED WORK AND CONTRIBUTION

Generally, there are multiple ways to transform a dataset into a k-anonymous representation, and different algorithms have been proposed. These can be classified along different axes. First, some algorithms implement global recoding, whereas others implement local recoding. Local recoding means that within a column, different generalization rules can be applied to equal values, whereas global recoding means that the same rule is applied. Local recoding is often used by clustering algorithms (e.g., [8]), whereas global recoding is mostly used by algorithms which utilize generalization hierarchies (see Section III for details). Depending on the coding model utilized, algorithms may find an optimal solution to the anonymization problem. Here, optimal describes the solution which results in minimal information loss according to a given metric. Solving this problem has been proven to be NP-hard [9]. There are approximation algorithms (e.g., [10]) which can find a solution that is within a guaranteed distance to the optimum. Various extensions to the k-anonymity model exist. The most important ones are ℓ-diversity [11], t-closeness [12] and δ-presence [13]. To derive the appropriate parameters for these algorithms, risk based anonymization [14] can be used. An extensive overview of previous privacy models and algorithms can be found in [15] and [16]. As long as they utilize global recoding with full-domain generalization, they can all be implemented in our generic framework and with our novel algorithm. Without loss of generality, we focus on the basic k-anonymity problem [5].

<table>
<thead>
<tr>
<th>Age</th>
<th>Gender</th>
<th>Zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>male</td>
<td>81670</td>
</tr>
<tr>
<td>45</td>
<td>female</td>
<td>81675</td>
</tr>
<tr>
<td>66</td>
<td>male</td>
<td>81925</td>
</tr>
<tr>
<td>70</td>
<td>female</td>
<td>81931</td>
</tr>
<tr>
<td>70</td>
<td>male</td>
<td>81931</td>
</tr>
</tbody>
</table>

Fig. 1. Example dataset

Approaches which find an optimal solution by applying full-domain generalization with user-defined hierarchies form an important class of k-anonymization algorithms [17]. The main reason for this is that global recoding delivers the best result in terms of statistical interpretability. Finally, the utilization of user-defined hierarchies allows the definition of different generalization strategies for different use-cases. Li et al. [18] have even shown that such algorithms can, under certain circumstances, guarantee a better degree of privacy than ones based on clustering and local recoding.

The algorithm of Samarati [19] is an early algorithm from this class. Because it is only able to find an optimal solution for a very simple utility metric (Height: see Section III) we will not discuss the algorithm in further detail. Instead, we focus

† Both authors contributed equally to this work.

DOI 10.1109/SocialCom-PASSAT.2012.52
on the two most well-known algorithms, including the current state-of-the-art. In [20] LeFevre et al. proposed the Incognito algorithm, which is oriented towards dynamic programming. In [17] El Emam et al. described a k-anonymization algorithm, which implements a divide-and-conquer approach, and showed that it outperforms the algorithms presented in [20] and [19]. As our novel algorithm is from the same class, we will describe these algorithms in more detail in Section [V].

This article comprises the following scientific contributions: 

- a description of a generic framework for the efficient implementation of k-anonymization algorithms, 
- an evaluation of this framework when implementing the two major previous algorithms, 
- the presentation of a novel algorithm which implements a new strategy and makes extensive use of our implementation framework, and 
- a comprehensive evaluation with various differently-sized real-world datasets showing that our approach outperforms previous approaches in terms of performance and algorithmic stability.

In Section [III], we present the basic concepts behind optimal k-anonymization with global recoding via full-domain generalization. Section [V] reviews the most important related algorithms. Section [V] covers the fundamental concepts behind our implementation framework. Our novel algorithm is described in Section [VI] and Section [VII] presents an extensive evaluation. Section [VIII] summarizes the results and discusses directions for further improvement.

III. BACKGROUND

A. Generalization and Monotonicity

Generalization hierarchies define a method of iteratively generalizing the values of an attribute. Figure [2] shows generalization hierarchies for the quasi-identifiers in the example dataset. In the two-anonymous version from Figure [1], the individuals’ age is transformed to intervals of length 50 (level 1), the attribute gender is suppressed (level 1) and the two least significant digits are dropped from the zipcode (level 2).

![Generalization Lattice](image)

Different models exist for using generalization hierarchies to transform datasets. In this context, full-domain generalization means that all values of an attribute are transformed to the same level of the associated hierarchy. Most algorithms which utilize full-domain generalization operate on a data structure called Generalization Lattice. It is shown in Figure [3] for our example dataset and hierarchies. An arrow denotes that a transformation is a direct generalization of a more specialized transformation and can be created by incrementing one of the generalization levels defined by its predecessor. The transformation with minimal generalization (0, 0, 0) is at the bottom and represents the original input dataset, whereas the transformation with maximal generalization (2, 1, 5) is at the top. The transformation that has been applied to anonymize the dataset from Figure [1] is (1, 1, 2).

Monotonicity is a very important property, which enables optimizing globally-optimal k-anonymity algorithms.

The authors of [20] introduced the notion of monotonic generalization hierarchies. In a monotonic generalization hierarchy, the groups at level \( l + 1 \) are built by merging groups from level \( l \). This allows pruning of large parts of the search space, because all transformations which are successors of a transformation resulting in an anonymized dataset are also solutions to the problem. Analogously, all predecessors of a transformation which does not fulfill the given set of privacy criteria can not be a solution. This can be used to predictively tag (i.e. classify) large parts of the solution space. An example is shown in Figure [3] where the fact that (2, 1, 1) is not a solution implies that all of its predecessors are also no valid solution (dark gray). Furthermore, all successors of the solution (1, 1, 2) are also valid solutions (light gray).

Additionally, the authors of [17] proposed the utilization of monotonic utility metrics. The monotonicity criterion for metrics requires that on each path from the bottom node to the top node of the generalization lattice, the measured information loss (i.e. decrease in utility) increase monotonically. This implies that if a generalization lattice is divided into a set of (potentially overlapping) paths the global optimum can be determined by only comparing the local optima. Furthermore, there is no need to measure the information loss for transformations that have been tagged predictively, because they can never be a local or global optimum.

B. Metrics

In this section, we briefly cover the most important utility metrics. The interested reader is referred to [17], [19]–[23] for further details. One of the earliest monotonic metrics is the Height Metric, which is utilized by the algorithm presented in [19]. This metric, as well as the monotonic variant of the Precision Metric [24], measure information loss solely based upon a transformation itself (i.e., its generalization levels) and is therefore independent of the actual input dataset. In [17], a monotonic version of the Discernability Metric [23] has been presented, which estimates information loss based on the equivalence classes induced by a transformation. Another
relevant metric is the Entropy Metric [21], which compares the original input dataset with the transformed representation. If needed, weights can be assigned to quasi-identifiers to tailor a metric to a specific use case [17].

IV. PREVIOUS ALGORITHMS

In this section we quickly review the two major previous algorithms which are relevant to our context. Incognito implements a horizontal traversal strategy (traversing the lattice level by level), whereas OLA implements a vertical traversal strategy (alternating between levels).

A. Incognito

LeFevre et al. proposed the Incognito algorithm [20], which implements an approach related to dynamic programming. The general idea is that if a transformed subset of the quasi-identifiers is not k-anonymous, the transformation of the complete dataset cannot be k-anonymous either. Therefore it constructs generalized lattices for each individual subset of n quasi-identifiers and traverses them by performing a bottom-up, breadth-first search. It utilizes predictive tagging to prune parts of the local search space. The transformations that have been found to be no solution for a subset of size \( m < n \) cannot be a solution for a subset of size \( m + 1 \). This allows to predictively tag transformations from generalization lattices constructed in subsequent iterations. The algorithm halts when the lattice for all \( n \) quasi-identifiers has been processed.

An example is shown in Figure 4. It shows the lattices built by Incognito for the example dataset. The algorithm starts by focusing on the quasi-identifier age. It builds a lattice and checks if the column is anonymous when applying \((0)\). As this is not the case, the algorithm proceeds with \((1)\), which is a possible solution. This results in predictive tagging of \((2)\). The same procedure is applied to the quasi-identifiers gender and zipcode. After checking all subsets of size one, Incognito proceeds with all subsets of size two. In this step, it is possible to tag all transformations that contain at least one transformation that was shown to not be a valid solution in a previous iteration. For example, in the lattice for the columns age and zipcode, all transformations which define level 0 for age and level 0 or 1 for zipcode can predictively be excluded from the set of solution candidates. Therefore, the first transformation that is checked by Incognito in this lattice is \((1, 2)\). As it results in a k-anonymous dataset, predictive tagging can be applied to all other transformations in this lattice. The other lattices are processed analogously. For more details the interested reader is referred to [20].

B. OLA

El Emam et al. proposed a k-anonymization algorithm called Optimal Lattice Anonymization (OLA) [17] and showed that it outperforms the approaches presented in [19] and [20]. It implements a divide-and-conquer approach. The idea is to decompose a lattice into smaller sublattices and utilize predictive tagging to prune parts of the search space. A sublattice \((b, t)\) is defined by a bottom node \( b \) and a top node \( t \) and contains \( b \) and \( t \) as well as all nodes that are generalizations of \( b \) and specializations of \( t \). OLA starts by processing the complete lattice. It then constructs sublattices by enumerating all nodes \( M \) on level \( \lfloor \frac{1}{2}(b.\text{level} + t.\text{level}) \rfloor \) of the current lattice. If a node \( m \in M \) has not been tagged already, it is checked for k-anonymity and predictive tagging is applied. If \( m \) is tagged as anonymous, the algorithm proceeds with the lower sublattice \((b, m)\), otherwise it proceeds with the upper sublattice \((m, t)\). This process halts when all sublattices have been enumerated.

The first iteration of the algorithm for our example dataset is shown in Figure 5. It starts by enumerating all nodes on level 4, and we assume that it starts with \((2, 1, 1)\). As this transformation has not been checked before, it is determined
whether it results in an anonymized dataset. Because this is not the case, \((2,1,1)\) as well as all of its specializations (dark gray) are predictively excluded from the set of solution candidates. The algorithm then proceeds to the upper sublattice \(((2,1,1),(2,1,5))\), which contains the light gray nodes. In the subsequent steps, it will construct the sublattices \(((2,1,1),(2,1,3))\) and \(((2,1,2),(2,1,2))\), effectively checking \((2,1,1),(2,1,3)\) and \((2,1,2)\) to find the local optimum \((2,1,2)\). The algorithm will then further proceed to the next node on level 4 which is \((2,0,2)\) and construct the according lower sublattice, as this transformation is a valid solution. More details are available in [17].

V. IMPLEMENTATION FRAMEWORK

This work is based upon a generic framework for the efficient implementation of \(k\)-anonymity algorithms. In [25], we presented an efficient implementation of the OLA algorithm on top of it. We quickly review the framework in this section and describe the fundamental ideas behind it:

1) The process of checking individual transformations for \(k\)-anonymity is the main bottleneck for this class of anonymization algorithms and should be as efficient as possible.

2) General purpose database systems are not well suited for \(k\)-anonymity algorithms, because they have been designed for much more complex query and transaction processing.

3) Given the current trend towards in-memory data management as well as the ability to use data compression techniques, holding all data in main memory is feasible.

The groundwork of this framework is a carefully-designed memory layout, which enables the efficient application of different generalization strategies to an input dataset. Additionally, the anonymization operators are problem-aware and intertwined with the rest of the algorithm. This allows for several further optimizations. The basic implementation, including the first optimization, can be used for all generalization-based anonymization algorithms which use global recoding with full-domain generalization. The other optimizations further require monotonic generalization hierarchies.

A. Basic implementation

Our framework holds all data in main memory and implements dictionary compression on all data items. Generalization hierarchies are represented in a tabular manner. An example for the hierarchy of the attribute \(age\) from Figure \(2\) is shown in Figure \(6\).

![Tabular generalization hierarchy](image)

One dictionary \(dic_0, \ldots, dic_{n-1}\) per quasi-identifier is used to map the values contained in the corresponding column onto integer values. By encoding the values from the input dataset before the values contained in the higher levels of the generalization hierarchies it is guaranteed that the original values of a column with \(m\) distinct values get assigned the numbers 0 to \(m - 1\). This allows for an efficient representation of the generalization hierarchies \(hier_0, \ldots, hier_{n-1}\) as two-dimensional arrays. An excerpt of the resulting memory layout for the example dataset is shown in Figure 7. The values of the attribute \(age\) from column 0 and the relevant values from the generalization hierarchy are encoded in the corresponding dictionary \(dic_0\).

![Example data representation](image)

The associated hierarchy \(hier_0\) is represented as a two-dimensional array where the \(i\)-th row contains the values for the original data item, which is encoded as \(i\) in the dictionary. The \(j\)-th column stores the corresponding transformed value at the \(j\)-th level of the hierarchy. The other quasi-identifiers are represented analogously. The input dataset itself is modelled as a row-oriented integer array \((data)\). Additionally, a similar data structure \(buffer\) is maintained which is used to store a transformed representation of the original data. Based on this memory layout, transforming a value from the input data in cell \((row, col)\) to the value defined on level \(level\) of its generalization hierarchy and storing it in the buffer can be implemented by a simple assignment:

\[
buffer[row, col] ← hier_{col}[data[row, col], level]
\]

When checking a transformation, the algorithm iterates over all rows in the dataset and applies the assignment \((1)\) to each cell. Afterwards, the transformed row is passed to a groupify operator, which computes the equivalence classes by adding the rows to a hash table. Finally, the \(k\)-anonymity check is applied by checking whether all classes are of size \(\geq k\). Furthermore, a suppression parameter \(s\) can be specified. It defines an upper bound for the number of rows that can be suppressed in order to still consider a dataset \(k\)-anonymous. This further reduces information loss, as the minimum size \(k\) is not enforced for all equivalence classes. Instead, classes of size \(< k\) are removed from the dataset as long as the total number of suppressed rows remains under the threshold. This basic implementation is already very efficient and has an amortized runtime complexity of \(O(n * m)\), where \(m\) is the number of rows and \(n\) is the number of columns.

B. Optimizations

The following section presents some further optimizations that exploit similarities between transformations which are checked consecutively.
a) Projection: Because transformed data is materialized in a buffer, it is possible to only transform those parts of the data that actually change. A projection can be applied if two consecutive transformations $s_1$ and $s_2$ define the same level of generalization for some quasi-identifiers. These columns are already represented in correct state and do not need to be transformed again. An example is shown in Figure 8. Here, only column 2 must be transformed when first checking $(2,1,1)$ and then $(2,1,3)$.

b) Roll-Up: When an algorithm moves from a transformation $s_1$ to a transformation $s_2$ which is a generalization of $s_1$, the equivalence classes resulting from $s_2$ can be built by merging the classes resulting from $s_1$ if the hierarchies are monotonic. This is called roll-up and is implemented by storing references to representative rows in the hash table. When a roll-up is performed, we iterate over all representative rows of the previous check and transform and groupify only those. A roll-up is also possible for the transition shown in Figure 8. In this figure, the classes are denoted by different shades of gray. We assume that the previous check resulted in $s_1 = (2,1,1)$ and $s_2 = (2,1,3)$.

c) Maintaining a buffer of snapshots: A series of roll-up operations can only be performed on a path of transformations that do not result in an anonymized dataset. A similar technique can be applied for other transitions, if a buffer is managed which contains snapshots of the equivalence classes that result from testing transformations for anonymity (called history). The classes from a transformation $s$ can then be built by merging the classes from a more specialized transformation $s'$ for which a snapshot exists. If multiple suitable snapshots are available, we pick the one that consists of the fewest classes. Furthermore it is only necessary to store snapshots for transformations that are not valid solutions, because otherwise all generalizations will be tagged predictively. The history has a predefined maximum size which is enforced by a LRU eviction policy and it only stores snapshots which consist of not more than a predefined number of classes (time-space tradeoff). We again store snapshots as a set of tuples of representative rows and sizes, which is very compact. This optimization does not fully replace the roll-up optimization, as it only stores a limited number of snapshots of a predefined maximum size.

d) Putting it all together: The individual optimizations presented in the previous sections can be combined with each other. The transition from Figure 8 can, for example, benefit from performing a roll-up as well as a projection. There are, however, several restrictions for valid combinations of optimizations, which ensure that the buffer is always in a consistent state. Combinations are mainly restricted in the context of projections, as these can only be performed successively if the current check allows a roll-up to be performed or when all rows have been transformed previously. For further details about the framework the interested reader is referred to [25].

VI. THE FLASH ALGORITHM

In this section, we present our novel algorithm: Flash. It traverses the lattice in a bottom-up breadth-first manner while generating paths which branch like lightning flashes. It is based upon the following observations:

1) Predictive tagging can be best exploited if a lattice is traversed vertically and in a binary fashion.

2) When traversing a lattice vertically, execution time becomes volatile in terms of the representation of the input dataset (e.g., the order of its columns). This must be prevented by implementing a stable strategy.

3) In order to achieve maximal performance, the algorithm should prefer checking transformations that allow the application of multiple optimizations.

A. Basic Algorithm

As shown in Algorithm 1, Flash iterates over all levels in a lattice, starting at level 0. It enumerates all nodes on each level and calls FindPath(node) if a node is not tagged already. Algorithm 2 shows that this function implements a greedy depth-first search towards the top node. The search terminates when either the top node is reached or the current node does not have a successor that is not already tagged.

Algorithm 1: Outer loop of the Flash algorithm

Input: Lattice lattice

begin
heap ← new min-heap
for $l = 0 \rightarrow \text{lattice.height} - 1$ do
  foreach node $\in \text{lattice}[l]$ do
    if node.tagged then
      path ← FindPath(node)
      CheckPath(path, heap)
    while !heap.isEmpty do
      node ← heap.extractMin
      foreach up $\in$ node.successors do
        if up.tagged then
          path ← FindPath(up)
          CheckPath(path, heap)

When a path has been built, the function CheckPath(path, heap) is called on it. As can be seen in Algorithm 3, it implements a binary search. It starts by checking the node at position $\lfloor \frac{1}{2}(\text{path.size} - 1) \rfloor$. 

Whenever a node is checked for k-anonymity, we also apply predictive tagging within the whole generalization lattice. Depending on the result of the check, the algorithm then proceeds with the lower or upper half of the path. Whenever a transformation is checked explicitly and determined to not result in an anonymous dataset, we add it to a heap. If a node is checked explicitly and determined to be a possible solution, we store a reference to it, as it could be the local optimum. There is no need to check whether a node has already been tagged, because by definition, Algorithm 2 always returns a path of transformations with unknown anonymity properties. Furthermore, predictive tagging is always applied in the direction opposite to the one taken by the algorithm. Another important thing to note is that after the search terminates, the variable optimum will always hold a reference to the local optimum on the path (if there is any). The global optimum is determined by comparing the current local optimum with the current global optimum in STORE(optimum).

Algorithm 2: FINDPATH(NODE)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>path ← new list</td>
</tr>
<tr>
<td>2</td>
<td>while path.head() ≠ node do</td>
</tr>
<tr>
<td>3</td>
<td>path.add(node)</td>
</tr>
<tr>
<td>4</td>
<td>foreach up ∈ node.successors do</td>
</tr>
<tr>
<td>5</td>
<td>if !up.tagged then</td>
</tr>
<tr>
<td>6</td>
<td>node ← up</td>
</tr>
<tr>
<td>7</td>
<td>break</td>
</tr>
<tr>
<td>8</td>
<td>return path</td>
</tr>
</tbody>
</table>

A node is a tuple \( n = (n_0, \ldots, n_j) \) with \( 0 \leq n_i \leq m_i \) for all \( 0 \leq i \leq j \), where \( m_i \) defines the highest level in the hierarchy of the \( i \)-th quasi-identifier. Furthermore \( \text{distinct}(i, l) = |\{\text{hier},[x][y] \mid y = l\}| \) returns the distinct number of values on level \( l \) of the \( i \)-th generalization hierarchy. We define three criteria for each node \( n \). The criterion \( c_1(n) \) returns the level of the node in the lattice:

\[
c_1(n) = \sum_{i=0}^{j} n_i
\]

The criterion \( c_2(n) \) is used to differentiate between nodes on the same level and resembles the Precision metric. It returns the average generalization of all quasi-identifiers:

\[
c_2(n) = \frac{1}{j} \sum_{i=0}^{j} \frac{n_i}{m_i}
\]

Finally, the criterion \( c_3(n) \) is utilized to differentiate between nodes on the same level which also imply the same average generalization. It returns the average of the number of distinct values on the current level of each quasi-identifier:

\[
c_3(n) = 1 - \frac{1}{j} \sum_{i=0}^{j} \frac{\text{distinct}(i, n_i)}{\text{distinct}(i, 0)}
\]

These three criteria are then combined into a vector in \( \mathbb{R}^3 \) in the following way:

\[
c(n) = \begin{pmatrix} c_1(n) \\ c_2(n) \\ c_3(n) \end{pmatrix}
\]

Nodes are traversed according to the totally ordered vector space induced by the relation \( \preceq \), which defines the lexicographical order, i.e. \( c(n_1) \preceq c(n_2) \) iff:

- \( c_1(n_1) < c_1(n_2) \), or
- \( c_1(n_1) = c_1(n_2) \land c_2(n_1) < c_2(n_2) \), or
- \( c_1(n_1) = c_1(n_2) \land c_2(n_1) = c_2(n_2) \land c_3(n_1) \leq c_3(n_2) \).
We apply this order when enumerating nodes on a level and when enumerating successors of a node. Finally, the priority function also serves as the key for entries in the heap. The strategy has to be implemented carefully. Sorting all levels in the lattice and all pointers to successors prior to the execution of the algorithm is too expensive. We therefore evaluate the priority function lazily and sort a nodes’ successors only when needed. Analogously, a level is sorted directly before iterating over it. This allows excluding all nodes that are already tagged from the sorting process.

C. Example

The first two iterations of the Flash algorithm can be seen in Figure 10. It starts by building a path from the bottom node (0, 0, 0) to the top node (2, 1, 5). This results in the rightmost path, indicated by the dotted lines. This path is then checked in a binary manner, which results in explicit checks of the transformations, indicated by the dotted lines. This path is then checked in the sorting process.

A. Datasets

The datasets include the 1994 US census database (ADULT), KDD Cup 1998 data (CUP), NHTSA crash statistics (FARS), the American Time Use Survey (ATUS) and the Integrated Health Interview Series (IHIS). The ADULT dataset serves as a de-facto standard for the evaluation of k-anonymity algorithms. An overview of the datasets is shown in Figure 11. They cover a wide spectrum, ranging from about 30k to 1.2M rows (2.52 MB to 107.56 MB) consisting of eight or nine quasi-identifiers. The associated generalization hierarchies have a height between 2 and 6 levels. The number of transformations in the generalization lattice, which is defined by the number of quasi-identifiers as well as the height of the associated hierarchies, ranges from 12,960 for the ADULT dataset to 45,000 for the CUP dataset.

B. Setup

The benchmarks were performed on a desktop machine equipped with a quad-core 3.1 GHz Intel Core i5 CPU running a 64-bit Linux 3.0.14 kernel. The algorithm was implemented in Java and executed on a 64-bit Sun JVM (1.6.0) with a heap size of 512 MB. We anonymized each of the datasets with \(2 \leq k \leq 10\), suppression rates \(s\) of 0%, 2% and 4% and the monotonic Discernability Metric. We furthermore incrementally enabled the optimizations, resulting in four different configurations. We executed each of these configurations for each algorithm (Incognito, OLA and Flash) and combination of the parameters \(k\) and \(s\), which resulted in 108 runs per dataset. The threshold for the maximum size of a snapshot was set to 20% and the size of the history was limited to 200 entries. Regarding the algorithms’ runtime behaviour we observed analogous results for other parameters. The results are reported without the time needed for initialization, which includes reading the entire data from disk and performing dictionary encoding (see Figure 11). The execution time of all three algorithms is dominated by the time spent on k-anonymity checks. As these are implemented in the same way for all algorithms, the comparison is fair.

 VII. Evaluation

For the evaluation we used five real-world datasets, most of which have already been utilized for benchmarking previous work on k-anonymity.

C. Overview

Figure 9 shows the workload averages (geometric mean over all values of \( k \) and \( s \)) for each dataset and algorithm. The basic implementation, which does not implement any optimizations, defines the 100% baseline. Optimizations are enabled incrementally, and it can be seen that all of them have a positive effect on the execution times of all algorithms for all datasets. Although we only show workload averages due to the limited space, it is important to note that none of the optimizations yields any overhead for any configuration. The Flash algorithm benefits the most, as it was explicitly designed to fully exploit the framework. Incognito benefits the least because of its horizontal traversal strategy, which, for example, completely prevents the roll-up optimization.

Figure 13 presents a comparison of the algorithms in terms of the geometric mean (logarithmic scale) of the execution times in seconds. Flash outperforms all algorithms and the speedup compared to the previous state-of-the-art (OLA) ranges from about 37% for CUP to a factor of 2.7 for IHIS.

4 seconds for all configurations despite ATUS and IHIS with 2% and 4% suppression rates. In these cases, more checks have to be performed to find the global optimum, which leads to execution times in the order of a minute.

Figure 16 compares the performance of Incognito and OLA to our algorithm. It shows the minimal and maximal factors between the execution times of the other algorithms and Flash over \( 2 \leq k \leq 10 \) for each configuration. It can be seen that Flash outperforms the other algorithms for all configurations on all datasets. The largest performance gain in comparison to the Incognito algorithm is achieved for ATUS \((k = 8, 0\%\) suppression rate) with a factor of more than 70. Flash especially outperforms OLA on the two largest datasets at a 0% suppression rate, with a factor of about 4 for ATUS \((k = 8)\) and a factor of about 5 for IHIS \((k = 10)\).

The experiments also show that our generic framework is suitable for the efficient implementation of other algorithms for optimal k-anonymity. Furthermore, OLA clearly outperforms the Incognito algorithm, up to an order of magnitude in some cases. We have not included numbers for the SuperRoots variant of Incognito, because our experiments revealed that it is only beneficial without suppression and decreases the performance in all other cases. Flash outperforms OLA in all cases, although our implementation of OLA is already highly efficient and includes several optimizations [25].

Fig. 16. Performance of Incognito and OLA compared to Flash [factor]

D. Details

Figure 12 shows a comparison of the actual execution times of the algorithms for \( 2 \leq k \leq 10 \) on selected configurations and datasets. It can be seen that the execution times of the algorithms do not differ only by a constant factor, but rather follow different trends. For each configuration, Flash offers an almost constant execution time and outperforms the other algorithms for each value of \( k \).

As can be seen in the Figures 14 and 16 this is also true for the other configurations. Figure 14 shows the minimal and maximal execution times of Flash for \( 2 \leq k \leq 10 \) on all datasets and suppression rates in seconds. It can be seen that Flash is able to find the optimal solution in well under 4 seconds for all configurations despite ATUS and IHIS with 2% and 4% suppression rates. In these cases, more checks have to be performed to find the global optimum, which leads to execution times in the order of a minute.

Figure 17 presents a comparison of the algorithms' memory consumption. The snapshots stored by the history dominate the overall memory footprint. Incognito uses the least memory, whereas OLA uses the most. Incognito has the lowest memory usage, because many nodes in the largest generalization lattice (for all quasi-identifiers) are already tagged when the lattice is traversed. It therefore creates significantly fewer snapshots.

Fig. 17. Memory consumption [MB]
than the other algorithms. Flash is designed to immediately exploit snapshots, which leads to early eviction of many entries in the buffer. It thus shows memory consumption which is in between the two extremes, Incognito and OLA.

Although the Flash algorithm has been designed to leverage our implementation framework as much as possible, it also offers competitive performance in other implementation scenarios. This can be seen by comparing Flash and OLA when all optimizations are disabled and the execution times are basically defined by the number of k-anonymity checks performed by an algorithm. In this case, Flash is up to 30% slower for the ADULT and FARS dataset with a 0% suppression rate. As these are the cases in which the execution times are very low, however, Flash still provides very good performance. In all other cases (including the larger datasets), the performance of Flash and OLA differs only by up to 10%. As soon as the optimizations are enabled, Flash outperforms OLA in all configurations by large margins. Furthermore, Flash offers algorithmic stability as is described in the following section.

E. Algorithmic Stability

In contrast to Incognito, Flash and OLA implement a vertical traversal strategy; however, only Flash offers stable execution times. Stable means that the execution time of the algorithm does not depend on the order of the columns in the input dataset or the algorithm that is used to build the generalization lattice. For example, a node \((1, 0, 0)\) represents a different transformation if the first column is swapped with another column in the dataset. This is an important property, as otherwise the behavior of an algorithm is not reproducible without explicit definition of these surrounding conditions.

As this property has not been discussed in any previous work, we assume that the experiments therein have been performed based on a natural ordering. First, the order of the columns in the input datasets has been preserved as is. Secondly, the successors of a node are ordered according to which quasi-identifier has been incremented. Finally, the nodes on a level are enumerated in the same way, assuming a breadth-first strategy during the lattice building process. This strategy has also been used for all example lattices in this paper and in our experiments (see, e.g., Figure 3). We were able to closely reproduce the previously published results.

The distribution of OLA's execution times can be determined if a fixed lattice building algorithm is used and OLA is executed for all permutations of the columns in the input datasets. Unfortunately, this is not feasible, as the number of permutations is factorial in the number of columns (e.g., \(9! = 362,880\)). Instead, we precomputed the anonymity property for all transformations in the lattices and used this information to simulate the execution of OLA for all permutations. The simulations take all optimizations from our framework into account. Each simulation results in a sequence of k-anonymity checks \(T\), which were performed while processing one permutation. Each check \(t \in T\) is defined by the number of active columns \((t_c,\) as induced by a projection, and the number of active rows \((t_r,\) as induced by the roll-up and history optimizations. \(c\) is the overall number of columns in the dataset. Based on this information, we developed a cost model which estimates the overall execution time. For one check \(t \in T\), the costs for transforming the data are defined as \(costs_\ell(t) = t_c \cdot t_r\), which is the number of cells that had to be transformed. Furthermore, the costs for grouping the data are \(costs_\ell(t) = c \cdot t_r\), because the costs of grouping always depend on the total number of quasi-identifiers in the dataset. In order to fit the resulting costs to the actual times measured in the experiments, we divided the results by a constant factor. The overall costs for a simulation \(T\) were defined as:

\[
\text{costs}(T) = \sum_{t \in T} costs_\ell(t) + costs_g(t) = \sum_{t \in T} t_c \cdot t_r + c \cdot t_r
\]

Figure 15 shows the distribution of the costs for different values of \(k\) and selected datasets. The frequency of a cost estimate is represented by colors ranging from white (lowest) to black (highest). The solid lines (OLA) represent the costs of the standard OLA algorithm with natural ordering, which has been used in the other experiments (e.g., in Figure 12). The dotted lines (StableOLA) show the costs of an implementation of OLA, which uses the same strategy as the Flash algorithm and is thus stable. To this end, the nodes on level \(\{b, \text{level} + t, \text{level}\}\) for a sublattice \((b, t,\) are enumerated in the order induced by our traversal strategy (see Section VI-B). The dashed lines represent the costs of Flash.

As can be seen when comparing Figure 15 to Figure 12, our cost model closely resembles the actual execution times of the algorithms. Except for the FARS dataset at 2% and 4% suppression rates, the trends of the resulting frequencies are very similar to the trends measured in our experiments. In contrast, the costs estimated for StableOLA always follow the overall trend. On average, this variant does not outperform standard OLA. The costs of OLA vary greatly (e.g., by up to a factor of 38 for IHIS with \(s=0\%\)). The performance with natural ordering is sometimes excellent (e.g. CUP, \(s=4\%\)) and sometimes quite poor (e.g. IHIS, \(s=2\%\)). In contrast, the Flash
algorithm offers a stable execution time, which outperforms OLA with natural ordering and OLA with the same stable strategy in all cases. Flash even outperforms the best run of OLA (fastest permutation) in 114 out of 135 cases.

VIII. Discussion and Future Work

In this paper, we have presented a generic implementation framework for globally-optimal full-domain k-anonymity algorithms. We have shown that it is well suited for the efficient implementation of the two major algorithms from this class, Incognito and OLA. We confirmed that OLA outperforms Incognito in case of natural ordering, but showed that OLA is not stable in terms of execution times. We further presented a novel k-anonymity algorithm, which clearly outperforms Incognito and OLA. Due to its vertical traversal strategy, it fully exploits predictive tagging and has been designed to make exhaustive use of our implementation framework. It offers algorithmic stability by relying on a consistent strategy that implies a total order among the nodes in the generalization lattice. Even when it is not implemented on top of our framework, it displays highly competitive performance.

In the original work on OLA, the authors reported a total execution time of 20 s for a subset of the ADULT dataset \( (k = 5, s = 1\%) \) on a comparable testbed \([14]\). With our framework, OLA was able to compute the optimal solution for the exact same configuration in 550 ms, whereas Flash needed 310 ms (both numbers include initialization). In \([20]\), the authors also published results for the same dataset \( (k = 5, s = 0\%) \) and an implementation of Incognito on top of a relational database system. In contrast to more than 3 minutes in the original work, our implementation of Incognito was able to solve this problem in less than 1s (including initialization) for the exact same configuration. Flash needed about 100 ms. This shows that a large performance gain can be achieved by implementing a dedicated data management framework for k-anonymity algorithms. An additional feature of our framework is that monotonic metrics can be evaluated with nearly no additional cost. As all locally optimal transformations are explicitly checked for whether they result in a k-anonymous dataset, complex metrics, which are based on the data itself or the resulting equivalence classes (e.g. Entropy), can directly access the transformed data in the buffer or the equivalence classes in the hash table. It is further possible to implement the \( \ell \)-diversity or \( t \)-closeness privacy models with our framework, which allow further balancing of data utility and privacy.

In future work we are planning to better leverage the capabilities of modern multi-core processors by parallelizing our implementation framework as well as the Flash algorithm. Early experiments with simple intra-operator parallelization within our framework were promising, but the parallelization of the k-anonymity algorithm itself is challenging. In case of limited availability of main memory or very large datasets, a disk-based implementation of a k-anonymity algorithm might be needed. Although this can be implemented on top of a relational database system, we are currently investigating a disk-based version of our framework and are confident that it clearly outperforms any off-the-shelf relational database system. Our generic framework and the implementation of the Flash algorithm are available as open-source software.\[1\]

REFERENCES

[12] N. Li et al., “\( t \)-closeness: Privacy beyond k-anonymity and \( \ell \)-diversity,” in 23rd Int. Conf. on Data Engineering, 2007, pp. 106–115.

\[1\]http://arX.deidentifier.org