Query Processing with K-Anonymity

Mohamed Y. Eltabakh
Department of Computer Science *meltabakh@cs.wpi.edu*
Worcester Polytechnic Institute
Worcester, MA, USA, 01604

Jalaja Padma
Cisco Systems *jpadma@cisco.com*
San Jose, California, USA, 95134

Yasin N. Silva
Division of Mathematical & Natural Sciences *ysilva@asu.edu*
Arizona State University
Tempe, Arizona, USA, 85281

Pei He, Walid G. Aref, Elisa Bertino
Department of Computer Science *{phe, aref, bertino}@cs.purdue.edu*
Purdue University
West Lafayette, Indiana, USA, 47907

Abstract

Anonymization techniques are used to ensure the privacy preservation of the data owners, especially for personal and sensitive data. While in most cases, data reside inside the database management system; most of the proposed anonymization techniques operate on and anonymize isolated datasets stored outside the DBMS. Hence, most of the desired functionalities of the DBMS are lost, e.g., consistency, recoverability, and efficient querying. In this paper, we address the challenges involved in enforcing the data privacy inside the DBMS. We implement the k-anonymity algorithm as a relational operator that interacts with other query operators to apply the privacy requirements while querying the data. We study anonymizing a single table, multiple tables, and complex queries that involve multiple predicates. We propose several algorithms to implement the *anonymization* operator that allow efficient non-blocking and pipelined execution of the query plan. We introduce the concept of *k-anonymity view* as an abstraction to treat k-anonymity (possibly, with multiple *k* preferences) as a relational view over the base table(s). For non-static datasets, we introduce the *materialized k-anonymity views* to ensure preserving the privacy under incremental updates. A prototype system is realized based on PostgreSQL with extended SQL and new relational operators to support anonymity views. The prototype system demonstrates how anonymity views integrate with other privacy-preserving components, e.g., limited retention, limited disclosure, and privacy policy management. Our experiments, on both synthetic and real datasets, illustrate the performance gain from the anonymity views as well as the proposed query optimization techniques under various scenarios.

Keywords: Data Privacy, K-Anonymity, Query Processing, Database Management Systems

1. INTRODUCTION

Privacy preservation is a key requirement for organizations holding personal or sensitive data. Organizations need to comply with the privacy preferences of data owners and privacy laws that regulate the use and sharing of such data [2]. Examples of privacy laws include the United States Privacy Act of 1974, the Australian Privacy Amendment Act of 2000, and The Health Insurance Portability and Accountability Act of 1996. These requirements have motivated a significant amount of work to answer the challenging question: How to make use of the data, e.g., querying and analysing, while ensuring the required level of privacy of data owners? Data anonymization techniques, e.g., [11, 12, 16, 17], have proposed to achieve the privacy preservation by ensuring that the identity of each

individual cannot be distinguished from a group of other individuals whose records exist in the same dataset. *K-anonymity* [16, 17] is one of the foremost anonymization techniques proposed in literature. It ensures that the identity of each individual is hidden within a group of at least *k*-1 individuals. Clustering and generalization, e.g., [5, 15], are common approaches to implement k-anonymity as we will discuss in more detail in Section II. Several algorithms have been proposed to provide stronger privacy preservation over *k*-anonymity, e.g., *l*-diversity [12], and *t*-closeness [11]. However, most of these techniques are standalone anonymization techniques implemented at the application level. That is, they assume the data is exported outside the database system and stored externally as standalone datasets. Moreover, the anonymized version of the data is also stored outside the database system and released to various Applications. This approach has major drawbacks including: (1) Sensitive data is transferred from the database system to external applications, which may compromise the privacy of the data owners, (2) Most desirable functionalities of the DBMS are lost, e.g., consistency, recoverability, efficient querying, and authorization mechanisms, (3) The entire dataset needs to be anonymized even if users are interested in a single (or few) records, and (4) Anonymizing complex queries with multiple predicates and joined tables is not feasible in the standalone version.

In this paper, we propose extending the database system to support the privacy-preservation requirements from within the database engine to overcome the shortcomings discussed above. We implement the k-anonymity algorithm as a relational operator that can compose, along with the standard query operators, a privacy-aware query plans. We propose several query optimizations, e.g., pushing the *selection* operator below the *anonymization* operator, to build efficient query plans. We propose several algorithms, *block-level* and *tuple-level*, to implement the *anonymization* operator and allow efficient non-blocking and pipelined execution of the query plan. We introduce the notion of *anonymization views* (*A-views*, for short) to abstract the problem of anonymization as a relational view on the base tables containing sensitive data. We extend the *k*-anonymity to *multi-k*-anonymity to support personalized anonymization, i.e., different individuals may choose different *k* values. We study anonymizing a single table, multiple joined tables, and complex queries that involve multiple predicates. We also address several challenges that emerge when anonymizing complex queries and/or joined tables (which they can be anonymization views themselves). In the paper, we focus on implementing the k-anonymity algorithm, however, the proposed *anonymization view* is independent of the underlying anonymization technique. We realized the proposed system through extensions to PostgreSQL where we extended SQL to create and manipulate the anonymization views and introduced a new anonymization operator to the query engine.

The rest of the paper is organized as follows. In Section II, we discuss related work. In Section III, we present the architecture of the proposed open-source privacy-aware database system. In Sections IV, we introduce the needed definiens and concepts. In Sections V, VI, and VII, we present the logical and materialized A-views over single and multiple tables. Sections VIII presents the performance evaluation and experimental results. Finally, Section IX contains concluding remarks.

2. RELATED WORK

Data anonymization techniques, e.g., [11, 12, 16, 17], have been proposed to enforce privacy preservation requirements. *K-anonymity* [16, 17] is one of the most common anonymization techniques in literature. It ensures that the identity of each individual cannot be distinguished from at least a group of other *k*-1 individuals in the same dataset. In this technique, the attribute-combinations that may reveal the identity of an individual are called *Quasi-Identifiers* (*QI*, for short) while the attributes pertaining to sensitive values, e.g., the disease name, are called *Sensitive-Attributes* (*SA*, for short). *K*-anonymity algorithms can be implemented using clustering or generalization techniques, e.g., [5, 15]. In the clustering approach, clustering algorithms are employed to identify groups of similar records that are represented by a single record. In the generalization technique, each quasi-identifier attribute is associated with a domain generalization hierarchy (DGH) from which the QI values can be generalized to form groups of at least *k* tuples with identical QI values. Examples of DGHs and *k*-anonymized tables are given in Figs. 1 and 2, respectively. Fig. 1 shows three attributes, *Disease*, *YearOfBirth*, and *ZipCode*, associated with their DGHs. Fig.2 shows a list of patient records where the combinations of QI

attributes (*Birthdate* and *Area*) are 2-level anonymous, i.e., there are at least 2 identical records (w.r.t QI values) for every QI attribute combinations.

Several algorithms have been proposed to provide stronger privacy-preservation over the k-anonymity technique, e.g., *l*-diversity [12], and *t*-closeness [11]. The *t*-closeness technique ensures that the distribution of sensitive values in a single anonymized group is as close as possible to the distribution in the base table. More sophisticated anonymization algorithms can also be embedded. These data anonymization techniques, as discussed in Section I, are standalone techniques that operate on the data at the application layer outside the database management systems. In contrast, the proposed materialization views ensure the data privacy inside the DBMS, and hence fully utilize the functionalities and query processing power of the DBMSs. Moreover, the abstraction of anonymization views is independent of the underlying anonymization algorithm being used. For example, and as a proof of concept, we implement the t-closeness algorithm in Datafly system [16] on top of the k-anonymity technique.

The concept of personalized privacy in [19] allows data owners to choose the level of generalization of sensitive attribute and to integrate it with k-anonymity to produce a stronger anonymized version of the data. We support sensitive attribute generalization. However, we keep generalization independent of anonymization, and provide it as an additional technique for protecting the privacy of sensitive data. While the integration of SA generalization with anonymization provides better utility, it may suffer from information leak through the locality of sensitive data. For example, as presented in [19], a tuple whose sensitive attribute is generalized to its parent value in the DGH containing 3 children is considered 3-anonymous, whereas in regular anonymization, the tuple can be part of a QI group having sensitive attributes that could be distributed across all the leaves in the DGH.

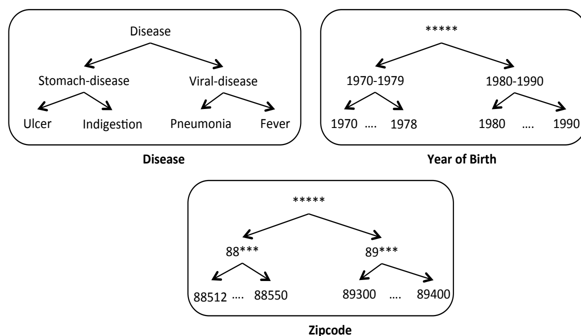

FIGURE 1: Domain Generalization Hierarchies.

Birthdate(QI)	Sex(QI)	Area(QI)	Disease(SA)
1970-74	F	Phoenix	Ulcers
1970-74	F	Phoenix	Indigestion
1985-89	M	Arkansas	Indigestion
1985-89	M	Arkansas	Fever
1985-89	M	Arkansas	Flu
1970-80	F	USA	Ulcers
1970-80	F	USA	Flu

FIGURE 2: An example of a 2-anonymous table.

Supporting data privacy at the database-system level is not a new approach. For example, Hippocratic Database [2] is an extension to the standard DBMS to manage the privacy of the data inside the database systems. Hippocratic databases include several privacy components to support policy management, limited disclosure, limited collection, limited retention, and compliance, among others. When a query is issued, the Hippocratic database system ensures that the recipient of the query answer receives only the data (s)he is allowed to see as per the policies. A data model to enforce the limiting disclosure principle enunciated in Hippocratic databases is proposed in [10]. In [3, 6], a Hippocratic fine-grained access control is proposed to limit the data disclosure. In [9], a middleware system is designed to implement and integrate several Hippocratic components. Other extensions to Hippocratic databases have been proposed in [1, 4, 10]. However, Hippocratic databases do not address data anonymization as a mechanism for privacy preservation, and they do not study the privacy as an operator inside the database engine that interacts with the other query operators. Most extensions to Hippocratic databases have been at the application-level or middleware-level between the application and the database. Anonymization views differ from the above proposals in that they are fully integrated inside the database.

Multi-relational k-anonymity [13] highlights several privacy attacks possible in multi-relational scenario and proves that the algorithms for a single relation do not achieve anonymity in the case of multiple relations. In this paper, we address the issue of anonymizing joined tables and propose different semantics for query results to ensure data privacy. For example, while [13] proposes hiding every tuple in some scenarios, we propose reporting false-positive tuples in addition to the correct (true-positive) tuples to ensure the privacy while maintaining high utility of the query result. Metrics to measure the privacy and utility of anonymized data are discussed in [20]. We adopt the *Normalized Certainty Penalty* proposed in [20] as a measure of utility.

3. SYSTEM ARCHITECTURE

The architecture of the proposed privacy-preserving database system is given in Figure 3. The system is an extension to our Hippocratic PostgreSQL system published in [21] which only supports a simple case of single-table anonymization. The goal is to develop a complete privacy-aware open-source DBMS by integrating the hippocratic database components, e.g., Policy Manager, Limited Disclosure, and Limited Retention, with the proposed anonymization techniques. The *Policy Translator* component is used to translate a P3P policy [7] into metadata. The *Parser* integrated with limited disclosure and retention support uses this translated metadata to produce query plans. We needed to change the *Parser*, *Planner*, and *Executor* of the database engine to integrate anonymization-related changes. We integrate and extend the ideas in [2, 9, 10] to implement our *limited disclosure* module. We use keywords *PURPOSE* and *RECIPIENT* to associate a purpose and a recipient to a given query, for example:

```
SELECT p.name, p.birth, p.sex, p.disease
FROM patient p PURPOSE research RECIPIENT lab;
```

Even though the definition of *limited retention* in Hippocratic databases [2] is to retain the information only as long as required, deletion of information after this period is not always trivial. Our approach to *limited retention* is through privacy policy specification of the retention periods for various attributes by each data owner. Every query complies with the policy for retention and does not display data that has exceeded its retention period. The *limited retention* and *limited disclosure* components are described in detail in [21]. The anonymization support and its interactions with the above privacy preserving modules are explained in the next sections.

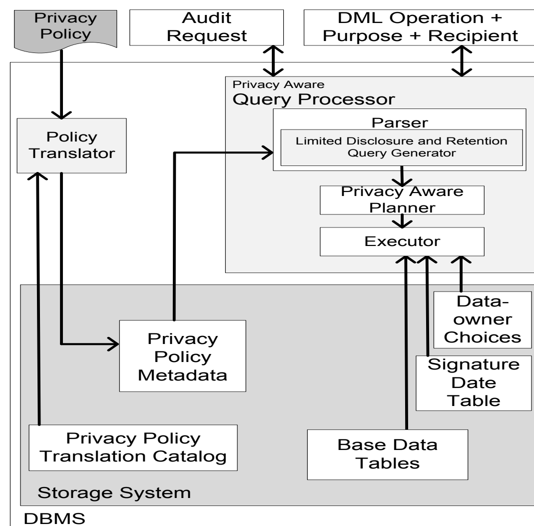

FIGURE 3: Hippocratic PostgreSQL architecture

4. DEFINITIONS AND CONCEPTS

Anonymization views (*A-views*) can be defined on a single table, multiple tables, or arbitrary SQL queries. They can be logical or materialized. A-views are similar to the standard database views except that they employs anonymization operators during query processing to ensure that the anonymization requirements are met. Before we describe how A-views work, we present the needed definitions and concepts.

Multi-k-Anonymity: The concept of *k*-anonymity is introduced to protect the privacy of individuals [16, 17]. It associates one global privacy measure *k* to all individuals. However, in order to be applicable to a wide range of applications and to support personalized privacy, we need to extend *k*-anonymity to support *multi-k*-anonymity. That is, every individual may specify his/her preference for *k*. Furthermore, we allow the data owners to specify a *k* value for each purpose/recipient combination. These preferences are part of the privacy policy signed by the data owners and are kept inside the database. The flexible *multi-k*-anonymity model is highly motivated by the diverse privacy requirements of the data owners. For example, a hospital database may contain information about ordinary and well-known individuals who, most probably, have different privacy constraints.

Domain Generalization Hierarchy (DGH): The k-anonymity algorithm uses domain generalization hierarchies to generalize the values in certain attributes. Example of DGH is depicted in Fig. 4. In our system, these DGHs are stored in a database table. A single DGH can be associated with one or more columns in the database. To create and populate the DGHs, we introduce the following extended SQL commands:

CREATE DGH <*dgh_name*>;
INSERT INTO DGH <*dgh_name*> <*child, parent*>;

The association between the DGHs and the database columns is established when creating the anonymization views.

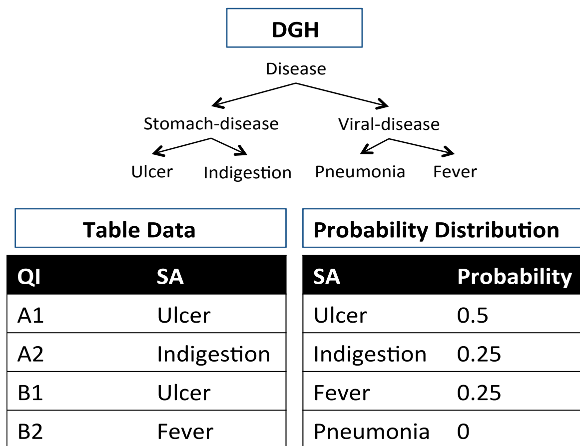

Table Data

QI	SA
A1	Ulcer
A2	Indigestion
B1	Ulcer
B2	Fever

Probability Distribution

SA	Probability
Ulcer	0.5
Indigestion	0.25
Fever	0.25
Pneumonia	0

FIGURE 4: DGH, Table, and Sensitive Attributes Distribution

```
CREATE [MATERIALIZED] ANONYMIZATION_VIEW a_view_name
ON       query
WITH     ANONYMIZATION_ID id_col
ANONYMIZATION_QUASI_ID ({qi_col[DGH_NAME dgh_name]} [,..] )
ANONYMIZATION_SENSITIVE_ATTR ({sa_col[DGH_NAME dgh_name]} [,..])
profile_col REFERENCES k_profile_table(k_col [,sa_level_col])
```

FIGURE 5: *Create Anonymization View* command

***Sensitive Attribute (SA) Generalization**:* Our work adopts the idea of sensitive-attribute generalization proposed in [9, 19] to provide additional privacy to data owners. However, unlike [19], this feature is kept independent of the anonymization process. That is, apart from specifying the *k* values, the data owner may specify the level of generalization (in DGH) of the sensitive attribute(s) for every purpose/recipient combination. For example (refer to Figure 4), if a person suffering from 'Ulcer' specifies a generalization level of 1, then the *Disease* attribute (the SA attribute) is generalized to 'stomach-disease', otherwise, it is reported as is.

Anonymization View (A-view): We introduce a new SQL construct to define an anonymization view as presented in Fig. 5. The command declares the identifiers, quasi-identifiers, and sensitive attributes of the data to be anonymized. Additionally, it associates DGHs to the quasi-identifiers. The *profile_col* gives the provision of choosing any attribute as the foreign key to a table containing the k-values and sensitive attribute generalization levels for various combinations of purpose and recipients. Anonymization views can be either *logical* or *materialized* (if the keyword MATERIALIZED is used in the command). Logical A-views suffer from temporal attacks if the datasets are not static [17]. This is because updating the data will results in different anonymized dataset each time. Hence, in the case of non-static datasets, we materialize the A-views to prevent these temporal attacks under incremental updates. Details about materialized A-views are discussed in Section VII.

5. ANONYMIZATION VIEW ON SINGLE TABLE

We propose two different query plans to anonymize queries on single tables. The two plans differ in where to plug-in the *anonymization* operator w.r.t the *selection* operator. The anonymization operator implements an extended version of the Datafly algorithm in [16] to accommodate for *multi-k*-anonymity and *t*-closeness. A key challenge in both plans is that anonymizing only the subset of tuples in the final query answer may result in a privacy breach by linking of the answers from multiple queries. For example, since queries can have different sets of tuples in the final answer, a quasi-identifier attribute of a specific tuple may have 3-level generalization in Query Q1 while having a 2-level generalization in Query Q2 (that is because the set of anonymized tuples are different). This clearly results in a privacy breach since the intended level of generalization in Q1 is lost once the adversary has the results from Q2. Therefore, the anonymization should relay on the entire collection of tuples in the base table instead of considering only the final query answer. Conceptually, if a logical A-view is defined on a table T, then any query on that A-view will use either of the two query plans presented in Fig. 6. The two plans are described next.

5.1. Anonymize-then-Select Plan

The Anonymize-then-Select plan performs the anonymization as early as possible in the query pipeline, i.e., immediately after the table-scan. Consequently, the selection predicates and/or projected attributes are not pushed inside the table-scan operator but rather delayed until after the anonymization is performed on the entire table. In the Anonymize-then-Select plan, a block of tuples is read from disk and these tuples are then anonymized and pipelined to the next query operator in the plan. Thus, the anonymization operator does not block the query pipeline until all tuples in the base table are anonymized. Instead it operates as a *block-level* anonymization. The detailed steps are as follow.

Step 1: Index-Scan

Using a direct table scan to read the data blocks from disk may produce tuples with different order depending on the table layout on disk (recall that the anonymization operator processes one block at a time). Thus, the anonymization algorithm may generate different outputs, which may lead to a privacy breach. To overcome this issue, Anonymize-then-Select plan uses a pre-defined index on the identifier attribute to perform an index-scan over the data (See Fig. 6). Thus, the anonymized blocks are guaranteed to have the same set of tuples independent of how the data is stored on disk.

Step 2: Sensitive Attribute Generalization

Each data owner can optionally specify a generalization level for SA attributes. If the generalization level is *m*, then SA attributes are generalized to their m^{th} ancestor in its DGH.

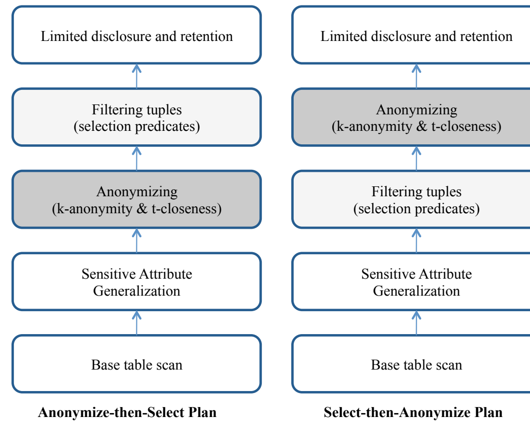

FIGURE 6: Anonymization query plans

Step 3: Block-Level Anonymization

We assume that each tuple belongs to a single data owner, e.g., patient record. If the *k* anonymity value is set zero, then no anonymization is required. If *k* =1, then it is 1-anonymous, i.e., identifiers are hidden, but will not be part of any QI group. The anonymization algorithm works as follow.

1) Retrieve the next block of tuples to process.
2) Identify the *k* value for each tuple from the A-view definition.
3) Mark the tuples having *k* = 0 as anonymized, and hide the identifier attribute of all the remaining tuples. Mark the tuples having *k* = 1 as anonymized .
4) Compute the probability distribution of the sensitive attribute values over the tuples in the block.
5) While there exist unanonymized tuples, repeat the following:
   5.1) Select the QI attribute having the maximum number of distinct values → Say QI_{max}.
   5.2) Generalize QI_{max} one level in its DGH. Then, merge all tuples with identical QI values in one group.
   5.3) If a group size is larger than or equals to the group's maximum *k* (max *k* for the tuples in the group), then compute the probability distribution for this QI group and calculate the *t*-closeness. If the group satisfies *t*-closeness, all tuples in that group are marked anonymized. Otherwise, repeat Step 5.1).
   5.4) If there are not enough tuples to be anonymized, then hide all identifiers, quasi identifiers, and sensitive attributes of the remaining tuples in the block. Then mark these tuples as anonymized.
6) If there are more blocks, then go to 1), otherwise exit.

Note that the anonymization operator does not wait until the anonymization of the entire block is performed to pass on the anonymized tuples to the next node in the query tree. Instead, anonymized tuples are pipelined after each iteration (Step 5 in the algorithm).

Step 4: Selection

Selection predicates on identifier attributes will not return any anonymized tuples since the identifier values are hidden in Step 3.3 above. For quasi-identifiers, it is proposed in [13, 14] that if tuple *t* is the only true query result, then *t* will not be reported in the answer to preserve the *k*-anonymity requirement of t's owner. In contrast, in our model, we report *t* along with at least *k*-1 other tuples that will have the same QI generalized values. Some of these reported tuples in t's QI group could be false-positive tuples, i.e., without generalization, they do not satisfy the query. This approach of including false-positive tuples preserves the anonymity requirements as well as improves the utility of the query, i.e., the number of true-positive reported tuples.

We currently support only equality predicates on QI and SA attributes. After generalizing the values in the QI and SA attributes, the equality operator needs to be extended such that an ancestor value in a

DGH should match any descendant value under that ancestor. For this purpose, we introduce the operator *AVLIKE (A-view LIKE)* that is used as illustrated in the following example.

Select * from PATIENT_AVIEW where zipcode = ‘88512’;

Assuming that *zipcode* is a QI attribute, then the query is re-written as:

Select * from PATIENT_AVIEW where zipcode AVLIKE ‘88512’;

Assume that two tuples t1 and t2 with zipcodes ‘88512’, and ‘88513’, respectively, are both generalized to ‘88***’, then both tuples will now satisfy the query (t1 is true-positive while t2 is false-positive). This algorithm is guaranteed to return all true-positive tuples along with their QI generalized groups.

Step 5: Limited Disclosure & Retention

The projection clause is manipulated to comply with the opt-in/opt-out choices of the tuple’s owners. Otherwise, the attribute value is replaced with a null value. The conditions for limited retention are also embedded within this clause. Hence, the algorithm combines the anonymization with the limited disclosure and retention to provide maximum privacy.

5.2. Select-then-Anonymize Plan

The Select-then-Anonymize plan is motivated by the fact that the former plan needs to anonymize the entire base table even if there is only one true-positive tuple. Therefore, for high-selectivity queries, it is more efficient to first find the true-positive tuples, and then construct their QI groups. In this case, the anonymization is a *tuple-level* anonymization that provides even better pipelining in the query plan and better response time compared to the *block-level* anonymization. The trick is how to form QI groups for the true-positive tuples that are still base-table dependent and not query-answer dependent. The detailed steps are as follow.

Steps 1, 2 & 3: Scan, SA Generalization, and *Selection*

In this step, tuples are scanned from the base table. The limitation of using pre-defined index scan (as in Anonymize-then-Select) does not apply here since the Select-then-Anonymize plan processes one tuple at a time. The SA attributes are generalized and predicates on these attributes are re-written to use AVLIKE operator instead of equality operators. If there are predicates on identifier attributes, the tuples will pass from the selection operator. But, they will be filtered in the anonymization operator (Step 4-3 below) except for tuples having k = 0.

Step 4: Anonymization

Select-then-Anonymize is based on the fact that every tuple *t* belongs to a particular QI group, say *g(t)*, in the anonymized version. Thus, the true-positive tuples are first selected, and then the QI groups for each of these tuples are then retrieved from the base table. The criterion for forming the QI groups is the same as in Anonymize-then-Select plan . One trick that Select-then-Anonymize uses is that the sequence in which the QI attributes are selected for generalization (Step 3-5.1, Section V.1) can be pre-computed for a given table, e.g., an example sequence can be: *zipcode, disease, disease, zipcode*, …. This order can be pre-computed by estimating the number of distinct values in each of the QI attributes after each generalization step. Given that QI generalization sequence is computed, the algorithm executes as follow.

1) Scan tuples from the table being queried. Filter the tuples based on selection predicates. If the selection predicate is based on SA, use the AVLIKE operator. Indicate if the selection predicate is based on identifier or QI attributes.
2) Identify the k-requirement for each true-positive tuple.
3) If the selection predicate is based on identifiers, then report only the tuples having k = 0.
4) Hide the identifier attribute of all tuples with k ≥ 1. Mark the tuples having k = 1 as anonymized.
5) Compute the probability distribution of the sensitive attribute values in the table.
6) For every tuple *t* remaining after step 5, go to Step 7.

7) Initialize set g(*t*) to *t*.
8) While the size of g(t) is less than the maximum *k*- requirement of any tuple in g(t) or g(t) does not satisfy t-closeness repeat Steps 9-10.
9) Select the next QI attribute to be generalized from the pre-computed order → Say QI_{max}.
10) Generalize the values in QI_{max}. Retrieve form the base table all tuples with values either matching or descendant to the value in QI_{max} and add them to g(t). Go to Step 8.
11) If the selection is based on QI attributes, return the tuples in g(t) (they are the QI group of *t*). Otherwise, return *t* only.

Step 5: Limited Disclosure and Retention

This step is performed as in the Anonymize-then-Select plan.

Unanonymized table

ID	QI1	QI2	SA	OA
1	A1	35	Sa1	Oa1
2	A2	35	Sa1	Oa2
3	A3	36	Sa2	Oa2
4	A4	31	Sa3	Oa3
5	B1	32	Sa4	Oa2
6	B2	32	Sa5	Oa4
7	B2	36	Sa3	Oa5

Anonymized table

QI1	QI2	SA	OA
A*	30-40	Sa1	Oa1
A*	30-40	Sa1	Oa2
A*	30-40	Sa2	Oa2
A*	30-40	Sa3	Oa3
B*	30-40	Sa4	Oa2
B*	30-40	Sa5	Oa4
B*	30-40	Sa3	Oa5

FIGURE 7: Example demonstrating privacy breach by querying QI and SA attributes

Special case: Predicates on both quasi-identifiers and sensitive attributes-

Queries that involve selection predicates on quasi-identifiers in combination with either sensitive attributes or other attributes or both need special handling. We explain this case with the help of an example given in Fig. 7. The attribute 'OA' is neither QI nor SA attributes. However, revealing the mapping between such attributes and QI attributes helps in mapping an individual to his/her SA attribute. For example, if the adversary is able to extract the information that individual with ID=1 (QI1= A1 and QI2= 35) has OA=*Oa1*, (s)he can easily deduce by looking at the corresponding QI group that the individual's SA value is *Sa1*. For this reason, consider the case of predicates on QI/SA combination separately. From Fig. 7, we observe that performing the selection before anonymization may cause privacy breach. For example, an adversary interested in finding the sensitive attribute of individual with ID=4 can issue the following queries.

```
SELECT * FROM T WHERE QI1=`A4' AND QI2=`31' AND SA=`Sa1';
SELECT * FROM T WHERE QI1=`A4' AND QI2=`31' AND SA=`Sa2';
```

Both queries will return empty results as there are no matching tuples and so the adversary successfully maps the individual to sensitive attribute value `Sa3'. It can be noted that the Anonymize-then-Select algorithm performs the selection after anonymization and so gives out some false positives even in this scenario. Therefore, Anonymize-then-Select does not suffer from this privacy breach. The issue with Select-then-Anonymize can be resolved by postponing the predicate evaluation of SA/OA attributes until after the anonymization takes place. This ensures that the results are consistent from both plans and that the adversary does not gain any further knowledge beyond what can be obtained from querying the anonymized version of the entire table.

5.3 Anonymize-then-Select vs. Select-then-Anonymize Plans

Although it is guaranteed that both plans will give query results that comply with the privacy policy, there are a few trade-offs in choosing one plan over the other.

Execution Time: Anonymize-then-Select plan has the drawback of anonymizing the entire base table even if the query is highly selective, e.g., only one true-positive tuple exists. In Contrast, Select-then-Anonymize plan anonymizes very few tuples (only the true-positives and their QI groups). However, during the anonymization of a tuple, Select-then-Anonymize plan issues several queries that may lead to a large number of random disk accesses. The number of database queries per tuple depends on the number of iterations required to form the tuple's QI group, which in turn depends on the distribution of *k*-values i.e., the smaller the average k-value, the lesser the number of iterations required to form the QI group for a given tuple. Therefore, the higher the selectivity of the query and the smaller the k-values, the greater the advantage of using Select-then-Anonymize plan and vice versa.

Utility: The utility is measured in terms of the lesser number of false-positive tuples included in the answer. The utility is better for Select-then-Anonymize plan, where the maximum number of false positives for a given tuple *t* is $|g(t)| - 1$. In contrast, for Anonymize-then-Select plan, the maximum number of false positives per tuples is $|Table| - 1$.

In Fig. 8, we demonstrate the difference in query answers from the two plans with a simple example. The base table *PATIENT* consists of four attributes and five tuples. The attribute *Name* is an identifier, *Birth* and *Zipcode* are quasi-identifiers, and *Disease* is a sensitive attribute. These attributes use the domain generalization hierarchies given in Fig. 1. The issued query has a single selection predicate on Zipcode QI attribute. The table contains only one true-positive tuple as illustrated in the figure. The answers from both plans meets the privacy requirements, yet the utility of Select-then-Anonymize plan higher than that of Anonymize-then-Select since the former produces less false-positives.

6. ANONYMIZATION VIEW ON JOINED TABLES

In this section, we study anonymization over multiple tables and/or A-views. We analyse the 2-way joins (extension to more than two entities is straightforward). In Fig. 9, we classify the 2-way join cases based on the join predicates as well as whether the joins between base tables and/or A-views. Joins on QI or SA attributes are allowed using the *AVLIKE* operator. For example, consider joining an A-view *A* and a base table *B*. Anonymize-then-Select plan follows the regular process of constructing the A-view *A*, and then joins this result with the raw table B. The intuition behind not joining earlier is that the outside entity B should not be able to *see* the sensitive raw data associated with the A-view *A*. Consider the case where table *B* (outside entity) has only one tuple and it is joined with the A-view *A* on the A.ID attribute. If *B* were joined with the base table of the A-view *A*, the result could be a single tuple. Anonymizing this single tuple would not protect privacy as the mapping between the identifiers and sensitive attribute for the tuple is implicitly exposed.

Select-then-Anonymize plan also ensures that the raw table joins with the A-view only. However, as an intermediate step, Select-then-Anonymize plan calculates the semijoin of *A* with *B*. This produces the tuples in *A* that would be participating in the join and thus need to be anonymized. The anonymized result is then joined with the raw table. The advantage of this approach over Anonymize-then-Select plan is the higher utility in cases where join predicates are based on QI. The reasons are the same as the ones explained in Section V.

In the case of joining two A-views, Anonymize-then-Select plan follows a similar approach to the one in the previous case. Each A-view is constructed independently and the results are joined using the *AVLIKE* operator, if needed. Select-then-Anonymize plan calculates the semi-join of *A* with *B* and *B* with *A*, thus identifying the tuples of *A* and *B* that will participate in the join. These tuples are independently anonymized (Section V.2) and then joined.

Name	Birth	Zipcode	Disease
P1	1984	88512	Ulcer
P2	1988	88540	Indigestion
P3	1979	88541	Fever
P4	1975	89321	Fever
P5	1977	89344	Pneumonia

PATIENT

Name_op	Birth_op	Zipcode_op	K	SA Level	Purpose	Recipient
F	F	T	2	1	Treatment	Nurse
T	T	T	2	0	Treatment	Nurse
F	F	T	3	1	Treatment	Nurse
T	T	T	2	1	Treatment	Nurse
T	T	T	2	0	Treatment	Nurse

PATIENT CHOICES

QUERY: ***Select * from PATIENT where zipcode='88512' purpose='Treatment' recipient='Nurse'***

TRUE POSITIVE RESULTS WITH NO PRIVACY

Name	Birth	Zipcode	Disease
P1	1984	88512	Ulcer

ALGORITHM I

Sensitive attribute generalization

Name	Birth	Zipcode	Disease
P1	1984	88512	Stomach-disease
P2	1988	88540	Indigestion
P3	1979	88541	Viral-disease
P4	1975	89321	Viral-disease
P5	1977	89344	Pneumonia

Anonymization

Name	Birth	Zipcode	Disease
*	1980-1990	88***	Stomach-disease
*	1980-1990	88***	Indigestion
*	1970-1980	****	Viral-disease
*	1970-1980	****	Viral-disease
*	1970-1980	****	Pneumonia

Selection & Limited Disclosure

Name	Birth	Zipcode	Disease
*		88***	Stomach-disease
*	1980-1990	88***	Indigestion
*		****	Viral-disease
*	1970-1980	****	Viral-disease
*	1970-1980	****	Pneumonia

ALGORITHM II

Sensitive attribute generalization

Name	Birth	Zipcode	Disease
P1	1984	88512	Stomach-disease
P2	1988	88540	Indigestion
P3	1979	88541	Viral-disease
P4	1975	89321	Viral-disease
P5	1977	89344	Pneumonia

Selection

Name	Birth	Zipcode	Disease
P1	1984	88512	Stomach-disease

Anonymization & Limited Disclosure

Name	Birth	Zipcode	Disease
*		88***	Stomach-disease
*	1980-1990	88***	Indigestion

FIGURE 8: Example demonstrating Anonymize-then-Select versus Select-then-Anonymize plans

7. MAINTENANCE OF A-VIEWS

A logical A-view work fine if the data do not change over time. If the data gets updated, and the DBMS is re-computing the A-view with each query, then the comparison of various answers may lead to a privacy breach. The incremental maintenance of K-anonymization has been studies in [8, 18]. In our model, we proposed *Materialized A-views* to handle insertions to existing data. For simplicity, we consider the *single-k* anonymity requirement. The A-view is materialized in the format shown in Fig 10, where QI groups and the corresponding tuple ids are stored.

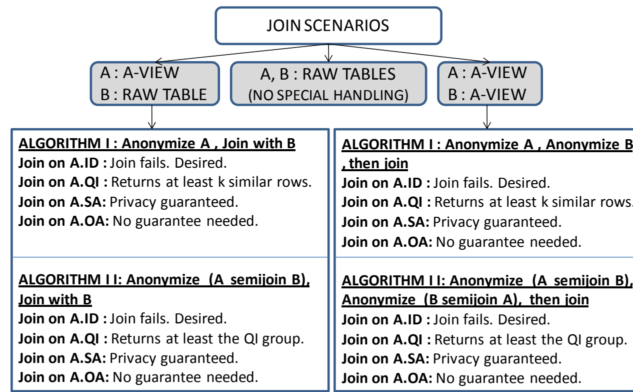

FIGURE 9: Join Scenarios

ID	QI1	QI2	SA	Others
{1,3,4}	30*	Person	{sa1,sa3,sa4}	{oa,oc,od}
{2,5}	40*	Male	{sa2,sa5}	{ob,oe}

FIGURE 10: Materialized A-view

Selection: Any query over a materialized A-view triggers a Pl/Sql procedure that uses the *AVLIKE* operator on QI attributes. In this case, the entire QI group is returned as output. For example, in Figure10, if the selection predicate is 'QI1 = 35', then 3 tuples formed out of the first cluster and returned as output. Selection predicates on SA or other attributes scan each of the tuples in the A-view (QI groups/ clusters) and processes the list of values to match the value in the predicate. Queries with selection predicates on identifiers return empty answer. The utility of materialized A-views is similar to that of Select-then-Anonymize plan except that in the case where the regular query result is empty. Select-then-Anonymize plan would return empty result whereas the materialized A-view may return a cluster that closely matches the predicate.

Insertion: Inserting to a base table of the materialized A-view triggers a Pl/Sql procedure that calculates the distance of the tuple being inserted from each of the tuples in the materialized A-view. Here, the distance refers to the sum of distances between corresponding quasi-identifiers. Each quasi-identifier value is a node in its DGH. The normalized distance between any two nodes in a DGH is given by the number of edges between the two nodes divided by the maximum distance between any two nodes in the DGH. This ensures that the distance measure for any QI is in the range [0 1]. The new tuple is inserted into the closest group (This may require modifying the QIs of the group). The closest group is the one where the total changes, measured in terms of normalized distance, of the inserted tuples and all existing tuples in the group, is the least. The ID, SA, and other attributes of the group are modified by adding the values of the new tuple to the corresponding lists. Upon insertions, the QI group attributes of other tuples can be only generalized to a higher node in DGH. In this case, temporal attacks from the insertion are avoided because the groups' attributes will become more generalized after the insertion. This is the trade-offs between the privacy and the utility.

Deletion: The select predicate in a delete command is processed on the base table and the IDs of the tuples to be deleted are noted. Each tuple is queried in the materialized A-view and is deleted from the group. If the group size decreases to a value less than 'k', the group is removed from the materialized A-view, and each of the remaining tuples in that group and individually inserted again into the materialized view following the insert procedure above.

Update: Update is handled as a delete of the old tuple followed by an insertion of new tuple with updated values.

8. PERFORMANCE ANALYSIS

8.1. Quality Measures

There is always a trade-off between privacy and utility in anonymization; the higher the privacy, the lower the utility of the data. The target is to maximize the utility of the anonymized data, while achieving the required levels of privacy. There are two types of utility in our system -1) utility of the query result (considers the number of false positives and false negatives) and 2) utility of the anonymized data (considers the effect of generalizing the data).

Precision and Recall: *Precision* is the ratio of true positives to the sum of true positives and false positives. Precision is always between 0 and 1, where 1 is the ideal value (no false positives). Since both plans return false positives when the selection predicates are on QI or SA attributes, the precision is below 1. In all other cases, the precision is 1. *Recall* is the ratio of the number of true positives to the sum of true positives and false negatives. Both plans are guarantee not to have false negatives (except for predicates on ID), and hence the recall is 1 in most cases.

Normalized Certainty Penalty (NCP): The NCP metric [20] measures the information loss for various choices of block sizes during the anonymization of large data. This metric can be used for the Anonymize-then-Select algorithm to identify the best block sizes to use since this algorithm is a block-level anonymization.

K-Deviation: To provide a measure of the unnecessary anonymization performed, we propose *k-Deviation*, which is the sum of differences between required *k* and achieved *k* for each tuple in the table. This value would be close to 0 for optimal algorithms. *k*-Deviation demonstrates the efficiency of the multi-k approach over the straightforward approach of anonymizing using single k, where k is set to the maximum required k over all tuples.

8.2. Experimental Results

We used a synthetically generated and real-world datasets in the experiments. The synthetic data include table 'PATIENT' with a maximum of 100K tuples, with 4 attributes {Name, Birthdate, Area and Disease}. 'Name' is the identifier, 'Birthdate' and 'Area' form the QI set, while 'Disease' is the sensitive attribute for the patient entity. Similar experiments are repeated over million tuples from the US Census Data (1990). Two identifiers 'ID' and 'NAME' are synthetically generated. The attributes 'AGE', 'SEX' and 'CITIZENSHIP' are used as QIs. 'SCHOOL (Education Level)' is considered the sensitive attribute. DGHs for QI and SA attributes are generated synthetically with a maximum depth of 4. The k-values and SA-Generalization levels for each of the million tuples are randomly generated. The k values range between 0 and 9 for all the experiments unless otherwise is specified. For all the experiments, it is assumed that the DGHs for all the attributes are in-memory data structures. The experiments are performed on Intel(R) CoreTM2 Quad CPU @ 2.83GHz machine with 3.5GB RAM.

Fig. 11 shows the runtime overhead of the anonymization operator (using Anonymize-then-Select plan) during query processing, with the block size set to 1024. Though the runtime with anonymization is asymptotically much higher than the runtime without anonymization, it is notable that the anonymization of 100k tuples is performed in less than 75 seconds for Synthetic dataset. In case of the US Census data set, the anonymization of 100k tuples is performed within 30 seconds and that of 1 million tuples is performed within 300 seconds. The time taken may vary with the algorithm used (algorithms that are more efficient in terms of utility may be slower); however, the fact that anonymization at the database engine level takes reasonable time is very promising. We observed in Fig. 12 that the run time varies with different block sizes (the table size in this experiment is set to 10K tuples). The trends show that higher block sizes marginally reduce the run time. This can be attributed to the fact that the larger numbers of tuples help in faster formation of QI groups and thus need a lesser number of iterations. Though this measurement helps in choosing the optimal block size for the dataset in hand, it is important to note that the available memory in the system (to store and anonymize the tuples) is a constraint on the maximum value of block size.

In Fig. 13, we measured the k-Deviation and NCP metric for multi-k and single-k variations. We distribute k-values in such a way that 10 percent of the tuples have high k (k=50) and the remaining values have low k (less than 5). We also show the trend for 1 percent high k values (k=50). It is observed that the k-Deviation of multi-k technique is marginally smaller than the one of the single-k technique. The reason for this is that single-k anonymization forms larger groups unnecessarily. The multi-k technique has a marginal gain over the single-k in this case of varying block sizes as well. We measured the utility loss of anonymized data for the single-k and multi-k cases and observed that NCP is close to 0 for large datasets and block sizes. This is intuitive since having large number of tuples for anonymization helps in easier formation of groups without the need for much generalization. Again, it is noted that the multi-k technique has lower NCP than the single-k case. The reason is that the multi-k technique strives for smaller group sizes (avoids unnecessary generalization) when compared to single-k, and hence results in lower k-Deviation and NCP.

In Fig. 14, we study the performance of the Select-then-Anonymize algorithm (labelled as Algorithm II in the figure). The Anonymize-then-Select algorithm is labelled as Algorithm I in the figure. We set the table size to 100K tuples and vary the selectivity of the issued queries. For queries with non-QI predicates, the response time initially increases linearly with the increase in the number of selected tuples. This is intuitive since the tuples selected based on non-QI attributes do not often tend to form QI/equivalence groups; this results in many database queries being issued for every selected tuple to find its QI group. However, the rate of increase in the running time decreases after a certain point as the algorithm comes across tuples that are already reported and thus no database queries are issued for a fraction of the selected tuples. On the other hand, the time taken by queries on QI attributes increases very slowly initially and speeds up after a certain point. The query predicates use equality or `like' operators. The reason for the this behaviour is that with a small number of selected tuples, it is highly probable that they form a part of the same QI group and thus the number of database queries to be issued is small.

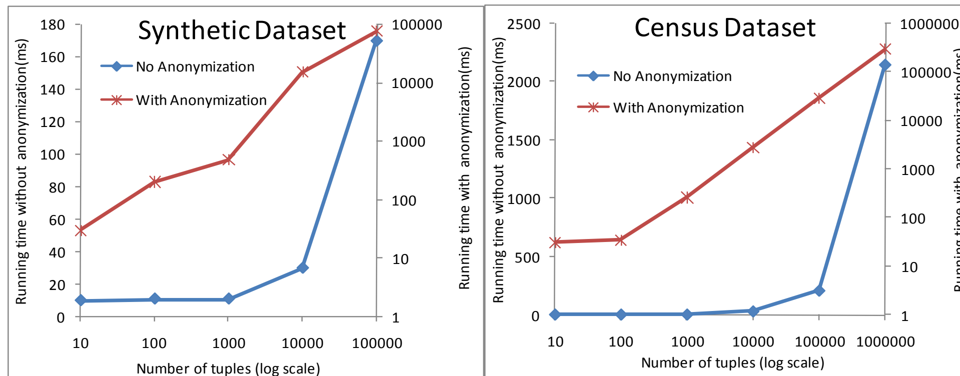

FIGURE 11: The runtime for Anonymize-then-Select algorithm under different datasets

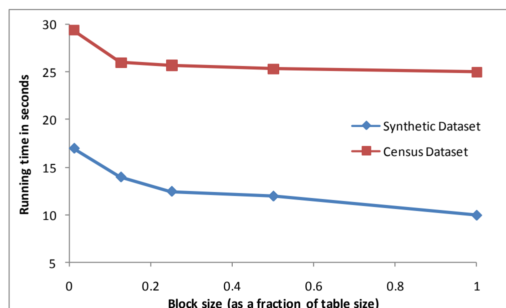

FIGURE 12: The runtime for Anonymize-then-Select under different block sizes

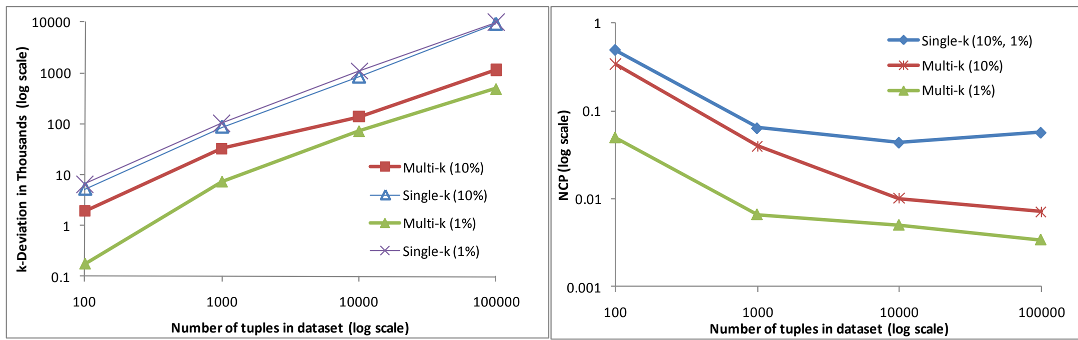

FIGURE 13: k-Deviation and NCP using Anonymize-then-Select plan (Census Dataset, sparse high-k values)

As the query result size increases, it implies that the quasi-identifiers of the selected tuples are not very similar and thus need more database queries to find their QI groups. It can be noted from Fig. 14 that Select-then-Anonymize (Algorithm II) performs better than Anonymize-then-Select (Algorithm I) if the query is highly selective, otherwise Algorithm I performs better since its anonymization cost is constant. The query selectivity and the distribution of the k values can be passed to the query optimizer to select the cheapest plan among the two. This optimization is left for future work.

In Fig. 15, we analyse the sensitivity of Select-then-Anonymize (Algorithm II) to the k value. We tested three values of k, 10, 50, and 200. The number of iterations, which maps to the number of issued database queries, increases with the increase in k. Thus, we see an increase in the runtime of Select-then-Anonymize plan, whereas Anonymize-then-Select plan takes similar runtime for all k-values.

Figs. 16 and 17 depict the performance of joins involving anonymization views. In Fig. 16, the join of an Anonymization View (100K tuples) is performed with a base table (10k tuples). As in the case of a single table, the trends are observed for QI-based and non-QI-based predicates. Fig. 17 demonstrates the runtime for the join of two anonymization views using synthetic datasets (100K and 10K tuples). we observe that the runtime almost doubled compared to joining an anonymization view with a base table. This is expected since the time taken to anonymize both relations is accounted. In both cases, the trend is similar to that in the single-table case. Similar trends are observed for Census dataset in the case Select-then-Anonymize plan, but the Synthetic data has much larger domain of distinct values for QI and non-QI attributes, and hence it displays much sharper trends.

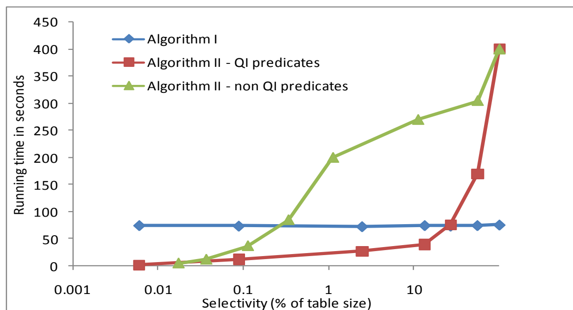

FIGURE 14: The runtime for Select-then-Anonymize plan (Synthetic Dataset)

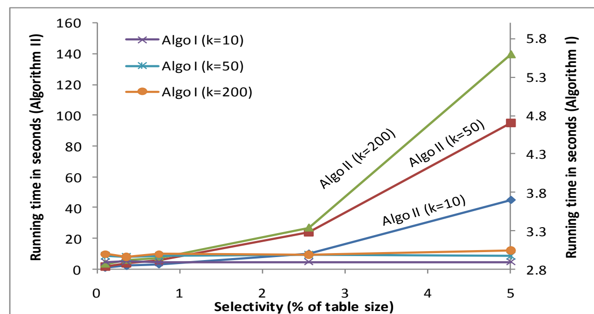

FIGURE 15: The runtime for Anonymize-then-Select plan and Select-then-Anonymize with varying `k'

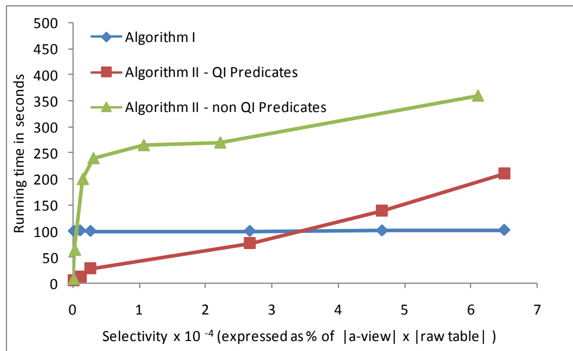

FIGURE 16: Joining an A-view and a base table

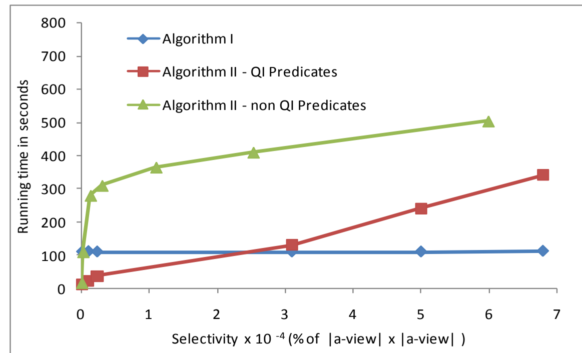

FIGURE 17: Joining two A-views

9. CONCLUSION

Protecting the privacy of the data inside the data's natural habitat, the DBMS, is an ever more demanding. It is (1) more secure, since the data is not transferred to an application layer or to a third-party, (2) more efficient, since the various query optimizations can be applied inside the database engine, (3) more flexible, since anonymization can be performed on arbitrary queries with multiple predicates, and (4) more reliable, since recovery mechanisms are already taken care of by the DBMS. In this paper, we proposed the concept of *anonymization views* (A-views) to enforce data privacy inside the DBMS. We presented two query plans Anonymize-then-Select and Select-then-Anonymize, which perform block-level and tuple-level anonymization, respectively, and studied the trade-offs between the two plans. For static datasets, we proposed the logical A-views to anonymize the data without the need to store the anonymized version. In contrast, for dynamically changing data, we proposed materialized A-views. Our experimental analysis and performance evaluation showed the feasibility of the proposed approach and highlighted the trade-offs among the proposed algorithms under different scenarios and settings.

REFERENCES

[1] Agarwal, R., Ameet Kini, Kristen LeFevre, Amy Wang, Yirong Xu, and Diana Zhou. *Managing Healthcare Data Hippocratically.* SIGMOD, 2004.

[2] Agarwal, R., Kiernan, J., Ramakrishnan Srikant, and Yirong Xu. *Hippocratic Databases. VLDB.* 2002.

[3] Agarwal, R., Paul, B., Grandison, T., Kiernan, J., Logan, S. and Rjaibi, W. *Extending Relational Database Systems to Automatically Enforce Privacy Policies. ICDE.* 2005.

[4] Agrawal, R., Bayardo, R., Faloutsos, C., Kiernan, J., Rantzau, R., and Srikant, R. *Auditing Compliance with a Hippocratic Database. VLDB.* 2004.

[5] Byun, J., Karma, A., Bertino, E., and Li, N. *Efficient k-Anonymization using Clustering Techniques. DASFAA.* 2007.

[6] Chaudhuri, S., Dutta, T. and Sudarshan, S. *Fine Grained Authorization Through Predicated Grants.* ICDE, 2007.

[7] Cranor, L., Langheinrich, M., Marchiori, M., Pressler-Marshall, M., and Reagle, J. *The platform for privacy preferences 1.0 (P3P1.0) specification.* W3C Recommendation, 2002.

[8] Jian Pei., Xu, J., Wang, Z., Wang, W., Wang, K., *Maintaining K-Anonymity against Incremental Updates.* 19th International Conference on Scientific and Statistical Database Management, 2007.

[9] Laura-Silva, Y N., and Aref, W. *Realizing Privacy-Preserving Features in Hippocratic Databases.* ICDE. 2007.

[10] LeFevre, K, Agarwal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y. *Limiting Disclosure in Hippocratic Databases.* VLDB. 2004.

[11] Li, N., Li, T., and Venkatasubramanian, S. *t-closeness: Privacy Beyond k-Anonymity and l-Diversity.* ICDE. 2007.

[12] Machanavajjhala, A., Gherke, J., Kifer, D., and Venkitasubramaniam, M. *l-Diversity: Privacy beyond k-Anonymity.* ICDE. 2006.

[13] Nergiz, Ercan M, Clifton, C., and Nergiz, A E . *Multi-relational k-Anonymity.* ICDE. 2007.

[14] Qihua Wang et al. *On the Correctness Criteria of Fine-Grained Access Control in Relational Databases.*VLDB. '07.

[15] Sweeney, L. *Achieving k-anonymity privacy protection using generalization and suppression.* International Journal on Uncertainty, Fuzziness and Knowledge-based Systems, '02.

[16] Sweeney, L. *Guaranteeing anonymity when sharing medical data, the datafly system.* Journal of the American Medical Informatics Association, 1997.

[17] Sweeney, L. *k-Anonymity:A model for protecting privacy.* International Journal on Uncertainty, Fuzziness and Knowledge-based Systems, 2002.

[18] Truta, Traian Marius, and Alina Campan. *K-Anonymization Incremenatal Maintenance and Optimization Techniques.* Symposium on Applied Computing. 2007.

[19] Xiao, Xiaokui, and Yufei Tao. *Personalized Privacy Preservation.* SIGMOD, 2006.

[20] Xu, Jian, Wei Wang, Jian Pei, Xiaoyuan Wang, Baile Shi, and Ada Wai-Chee Fu. *Utility-based Anonymization using Local Recoding.* KDD. 2006.

[21] Padma, J., Silva, Y., Arshad, U., Aref, W. G. *Hippocratic PostgreSQL.* ICDE. 2009.

[22] Frank D. McSherry. *Privacy integrated queries: an extensible platform for privacy-preserving data analysis.* SIGMOD, 2009