Limiting Disclosure in Hippocratic Databases

Kristen LeFevre†* Rakesh Agrawal† Vuk Ercegovac* Raghu Ramakrishnan*
Yirong Xu† David DeWitt*

†IBM Almaden Research Center, San Jose, CA 95120
*University of Wisconsin, Madison, WI 53706

Abstract

We present a practical and efficient approach to incorporating privacy policy enforcement into an existing application and database environment, and we explore some of the semantic tradeoffs introduced by enforcing these privacy policy rules at cell-level granularity. Through a comprehensive set of performance experiments, we show that the cost of privacy enforcement is small, and scalable to large databases.

1 Introduction

The Lowell database research self-assessment of June 2003 points to data privacy as an important area for future research [4]. One of the defining principles of data privacy, *limited disclosure* [6], is based on the premise that data subjects[1] have control over who is allowed to see their personal information and for what purpose. For example, a patient entering a hospital provides some information at the time of registration with the understanding that this information may only be used under certain circumstances; for example, the billing office may use the patient's address information to process insurance claims, but the hospital may not give patient address information to charities for the purpose of solicitation without consent [1].

Increasingly, organizations want the ability to define a *privacy policy* that describes such agreements with data subjects and to ensure that the policy is enforced with respect to all data access. Essentially, a privacy policy is comprised of a set of rules that describe to whom the data may be disclosed (the *recipients*) and how the data may be used (the *purposes*). Additional *conditions* may be specified to govern disclosure. For instance, a policy may specify that a particular data item may be disclosed, but only with "opt-in" consent from the data subject, or that the data item will be disclosed unless the subject has specifically "opted out" of this default. The policy may also specify more complex conditions; for example, a patient's medical history may only be seen by nurses assigned to the same floor. While there is recent work on defining languages for specifying privacy policies (e.g. P3P [12], EPAL [7]), database mechanisms for enforcing such policies have not been investigated.

An approach often taken is to enforce privacy policies at the application level [8]: First, the application issues the query to the database and retrieves the result. Then, the application scans the resulting records and filters prohibited information (for example, by setting it to *null*). However, this approach leads to privacy leaks when applied at the cell level. Consider a query involving a predicate over a privacy-sensitive field: `SELECT Name, Disease FROM Patients WHERE Disease = "Hepatitis"`, and Bob, who has hepatitis, and chose to disclose his name but not his disease history. The query result contains Bob's record with the `Disease` value filtered out. Unfortunately, this allows anyone looking at the results to conclude that Bob has hepatitis.[2]

1.1 Requirements for Limited Disclosure Mechanisms

A solution to the limited disclosure problem should ideally protect information according to the appropri-

[1] We use the term *data subject* to mean the individual whose private information is stored and managed by the database system.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

[2] An alternative might retrieve all of the patient records, not just those with a particular disease, and apply the privacy-sensitive predicate in the application. However, this approach leads to significant performance problems as much data must be unnecessarily fetched from the database. Query execution is more difficult yet when we consider more complicated queries, such as those involving aggregates or joins, because we must extract a significant amount of data from the database, and then perform a large amount of the query processing at the application level.

ate policies with minimal privacy-checking overhead. Further, given the time and expense required to modify existing application code, such a solution should require minimal change to existing applications and little reorganization of existing data.

It is particularly important that we manage disclosure at a very fine granularity because privacy policies can refer to "data items" at the level of an individual cell in a relational table. Traditional databases provide access control at the table level and use the view mechanism to restrict access to certain columns or rows of a table. Some systems [19] now provide access control at the row level, but this is still inadequate. Consider Alice, who has opted to allow the hospital to release her email address but not her phone number to charities. Bob might choose to provide his phone number, but not his address. Row-level enforcement must either filter information that should be permitted, or disclose prohibited information.

1.2 Contributions and Paper Organization

Our principal contribution is a database mechanism, introduced in Section 2, for enforcing privacy policies, with the following features:

- Privacy policies can be stored and managed in the database.
- A broad range of privacy policies expressible in high-level privacy specification languages (e.g. the privacy rules expressible in P3P, including opt-in and opt-out choices, and more complex conditions as seen in EPAL) can be enforced.
- Enforcing privacy policies does not require any modification to existing database applications.

The power of our approach is based on two technical properties: first, it supports cell-level enforcement; second, it allows full use of SQL query capabilities to express conditions. In more detail, our contributions are:

- We investigate the alternative semantics for limited disclosure in relational databases and present two models of cell-level limited disclosure: *table semantics* and *query semantics*, weighing the relative semantic and performance tradeoffs implicit in the two models. (Section 3)
- We provide techniques for enforcing a broad class of privacy policies by automatically modifying all queries that access the database in a way that ensures the desired disclosure semantics. Rather than viewing the disclosure control problem as one of checking against a list of privileges, we transform it into a query modification problem. Thus, our implementation automatically benefits from years of experience in database query processing, including parallelism. We examine several implementation issues, including privacy meta-data storage, query modification algorithms, and structures for storing conditions and individual choices. (Section 4)
- An experimental evaluation of our techniques shows that our approach has low overhead and frequently speeds up queries by using privacy-related restrictions as additional selection conditions. The conventional wisdom has been that cell-level access control is prohibitively expensive. We provide extensive experiments comparing the implementation alternatives that we present and demonstrate that our implementation is scalable to large databases. (Section 5)

Although the work presented here has been done in the Hippocratic database setting [6], it has much wider application. A growing range of applications in content management, customer support, financial analysis, and e-commerce require cell-level access control. In fact, many such applications might use privacy policies in the dual role of *access control policies*.

1.3 Related Work

The work in database access control can largely be grouped into the areas of discretionary and mandatory [21]. Discretionary access control allows a database administrator to grant and revoke access privileges, which typically refer to entire tables or views; optionally, the DBA may specify that others are authorized to grant and revoke privileges [14]. Role-based access control is an additional refinement which allows this type of privilege to be granted not to an individual user, but to the user's group, or role [23].

The mandatory access control model involves a single set of rules governing access to the entire system. A well-known model of mandatory access control, the Bell-LaPadula multilevel secure database, defines permissions in terms of objects, subjects, and classes [9]. Each object is a member of some class, for example "Top Secret" or "Unclassified," which typically form a hierarchy. The model also allows for the possibility of polyinstantiation [18]. These formalizations are further refined by [15, 16], and a schema decomposition allowing cell-level classification to be expressed as row-level classification is described in [20].

To our knowledge, the only cell-level implementation of multi-level security was done by SRI in the SeaView system [18], but its performance was never published [3]. The idea of modifying queries for access control was introduced in [24], and the idea of "reformulating" queries for security was also alluded to by [25]. A query rewrite mechanism to control access to federated XML user-profile data was used by [22]. Oracle's Virtual Private Database product allows for the definition of access control functions, which may be data-driven, and which operate at the row level through addition of predicates [19]. Some content-management applications have enforced fine-grained security by introducing an application layer that modifies queries with conditions that enforce access control policies, e.g. [2, 17], but they are application-specific

in their design and do not extend a DBMS for general use.

There has been extensive research in the area of statistical databases motivated by the desire to provide statistical information (sum, count, etc.) without compromising individual information (see surveys in [5, 26]). It was also shown that we cannot provide high quality statistics and, at the same time, prevent partial disclosure of individual data. Our goal in this paper is to provide database support that allows individual queries to respect privacy policy rules and individual subject choices, and we assume that additional mechanisms such as query admission control and audit trails [5] are in place to guard against the inference problem. This paper also does not purport to address the use of covert channels to leak information[3], but assumes that the appropriate security mechanisms are in place to control such leaks [10].

2 System Overview

We have developed a database architecture for enforcing limited disclosure as expressed by privacy policies. The basic components are the following:

- **Policy definition:** Privacy policies are expressed electronically and stored in the database where they can be used to enforce limited disclosure. Our prototype provides a policy "meta-language" for defining privacy policy rules. Privacy policies defined using P3P [12] or EPAL [7] may be translated into this meta-language and then stored in the database, making our architecture policy-language independent.
- **Privacy meta-data:** The rules and conditions prescribed by the privacy policy are stored as privacy meta-data tables in the database. The structures for storing this information are described in Section 4.1.
- **Application context:** Each query must be associated with a purpose and recipient. In our system, this information is inferred based on the context of the application issuing the query, which is similar to the approach described in [19]. The query interceptor infers the purpose and recipient of the query based on context information stored in an additional meta-data table.[4]
- **Query modifier:** SQL queries issued to the database are intercepted and augmented to reflect the privacy policy rules regarding the purpose and recipient issuing the query. The results of this new query are returned to the issuer. Several query modification algorithms are detailed in Section 4.2.
- **Disclosure model:** The result of executing a privacy-modified query will reflect one of two cell-level limited disclosure models, as described in Section 3.

[3]For example, error messages generated during the evaluation of privacy rules could become a *covert channel* for leaks.

[4]An alternative would extend the syntax of a SQL query to include purpose and recipient information, for example `SELECT * FROM Patients FOR PURPOSE Solicitation RECIPIENT External_Charity`. Because this method requires extensions to the query language and modification to existing applications, we elect to use application contexts, though the rest of our implementation is compatible with either alternative.

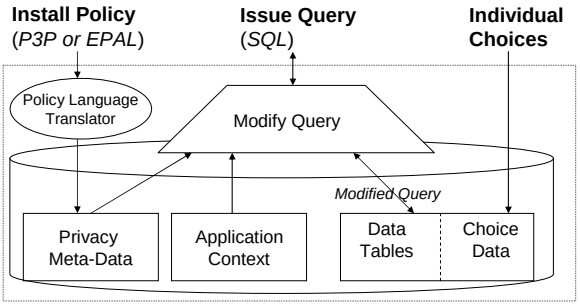

Figure 1: Implementation architecture overview

2.1 Policy Meta-Language, Rules and Conditions

A privacy policy will be considered to be a set of rules of the form $<data, purpose\text{-}recipient\ pair, condition>$. An example of such a rule is $<address, solicitation\text{-}charity, optin = yes>$, meaning that a data subject's address can be released to a charity organization for the purpose of solicitation, provided the subject has explicitly consented to this disclosure. A *condition* is any boolean predicate that is expressible in SQL. Electronic privacy policies are programatically translated into this policy "meta-language" before being deployed.

A conditional predicate may refer to the data table T, in addition to any other data tables. By joining T with other tables on primary key $T.key$, these conditions can be made to depend on any attributes of the "current" T row, and by referring to the context environment variable (which we denote $USERID) that identifies the user issuing the current request, the conditions can be made specific with respect to the current user. For example, suppose we were to define a condition to govern the disclosure of patient data to hospital nurses such that, for treatment, nurses may only see the medical histories of patients assigned to the same floor. This condition can be expressed using the following predicate:

```
EXISTS (SELECT NurseID
    FROM Nurses
    WHERE Patients.floor = Nurses.floor
    AND $USERID = Nurses.NurseID)
```

Another example of data-driven conditions arises when we consider defining disclosure based on application-defined user groups. In this scenario, the application owns some database table *D(key,* a_1*,* a_2*,...)* storing application data. The application also maintains a list of users *U(uid)* and groups *G(gid)*, a

table mapping users to groups, *M(uid, gid)*, and an access control list *A(key, a_i, gid)* associating each record in D with a list of groups that have access to attribute a_i. Suppose one of the privacy policy rules allows disclosure of data attribute $D.a_i$ to purpose P and recipient R only when the current user belongs to a group which has been granted access to $D.a_i$. Such a condition is specified as a simple SQL predicate joining tables M and A:

```
EXISTS
    (SELECT gid
    FROM M, A
    WHERE M.gid = A.gid AND M.uid = $USERID
    AND A.key = D.key AND A.a_i)
```

A potential source of difficulty in translating from a high-level policy specification into our meta specifications could be the gap in vocabulary. Policy specifications are written in terms of an information model that might use entity names that need translation into the column and table names used in the meta specification. However, the necessary mapping can be specified using a GUI tool and then used in the translator from the high-level specification into meta specification.

Another difficulty could be the difference in the expressive power of the the two condition languages and the difference in the execution models. We investigated this problem by experimenting with translating conditions from P3P and EPAL into our condition language. Writing the translator for P3P was straightforward. We could also translate most of EPAL, although this translation was more challenging. The EPAL engine evaluates rules sequentially, in the order they appear in the policy. Our policy language is set-oriented, so we had to modify the execution semantics of EPAL so that if two or more rules qualify, they are all fired to allow access to data. Similarly, EPAL conditions support some operators and data types for which there is no corresponding analog in SQL. However, not supporting these constructs does not appear to be a limitation in practice; we looked at several EPAL policies and found easy translations for them into our policy meta-language.

3 Limited Disclosure Models

We introduce two models of cell-level limited disclosure enforcement: *table semantics* and *query semantics*. The table semantics model conceptually defines a view of each data table for each purpose-recipient pair, based on the disclosure constraints specified in the privacy policy. These views combine to produce a coherent relational database for each purpose-recipient pair, independent of any queries, and queries are evaluated against this database. In contrast, the query semantics model takes the query into account when enforcing disclosure. Both models mask prohibited values using SQL's *null* value.

Let a privacy policy consist of statements that involve m purpose-recipient pairs $P = <P_1, P_2, ..., P_m>$. In order to define the semantics of the limited disclosure model, every data table T with n columns will be (conceptually) extended with $m * n$ columns that record disclosure conditions for the corresponding cell for a given purpose-recipient pair.

We use the notation $T[i]$, $1 \leq i \leq n$, to refer to the data columns of T. We use $t[i,j], 1 \leq i \leq n, 1 \leq j \leq m$, to refer to the column containing the disclosure condition for data column i and purpose-recipient pair j. For any set of columns S in a table T, "$t[S]$ *nonenull*" denotes that every column in S is non-null in tuple t. We use $eval(t[i,j])$ to denote the boolean result of evaluating the (predicate expressing the) condition that controls disclosure of data column i in the current row of T to purpose-recipient pair j. We say that the cell in column i of the current row is *prohibited* to purpose-recipient j (or simply *prohibited*) if the value of $eval(t[i,j])$ is $false$.

3.1 Table Semantics Model

In the table semantics model, each purpose-recipient pair Pj is assigned a view over each table in the database, and prohibited cells are replaced with null values. Also, if any column in the primary key of a row is prohibited, then the entire row is prohibited.[5] This model is formally defined as follows:

Definition 1: (Table Semantics) Let T be a table with n data columns, and let K be the set of columns that constitute the primary key of T. For a given purpose-recipient pair Pj, the table T, seen as T_{Pj}, is defined as follows:

$$\begin{aligned}\{r | \exists t \in T \quad & \wedge \forall i, 1 \leq i \leq n \\ & (r[i] = t[i] \ if \ eval(t[i,j]) = true, \\ & r[i] = null \ otherwise) \\ & \wedge \ r[K] \ nonenull\}\end{aligned}$$

Consider a table containing patient information, as shown in Figure 2. The hospital allows patients to choose on an opt-in basis if they want these categories of information to be released to charities (recipient) for solicitation (purpose). Figure 3 shows the choices made by the patients. The resulting privacy-enforced table of patients according to table semantics is shown in Figure 4, assuming that `P#` is the primary key.

3.2 Query Semantics Model

With query semantics, prohibited data is removed from a query's result set based on the purpose-recipient pair and the query itself. Here, we do not aim to define a version of the underlying table for each

[5] A simpler cell-level enforcement model can be obtained by dropping the null-restriction on the primary key, but the privacy-enforced data tables will no longer be consistent with the relational data model, which requires primary keys to be non-null. We call this alternative *strict cell-level* semantics, but we do not consider it further.

P#	Name	Age	Address	Phone
1	Alice Adams	10	1 April Ave.	111-1111
2	Bob Blaney	20	2 Brooks Blvd.	222-2222
3	Carl Carson	30	3 Cricket Ct.	333-3333
4	David Daniels	40	4 Dogwood Dr.	444-4444

Figure 2: Full data table of patient information.

P#	P#	Name	Age	Address	Phone
1	√	√	√	√	√
2	×	×	×	×	×
3	√	×	×	√	√
4	√	√	×	×	×

Figure 3: Patient choices for disclosure of information to charities for solicitation.

P#	Name	Age	Address	Phone
1	Alice Adams	10	1 April Ave.	111-1111
3	-	-	3 Cricket Ct.	333-3333
4	David Daniels	-	-	-

Figure 4: Privacy-enforced table of patient information, using table semantics.

Name	Age
Alice Adams	10
-	-
David Daniels	-

Name	Age
Alice Adams	10
David Daniels	-

Figure 5: Comparing Table Semantics and Query Semantics for a simple projection

purpose and recipient, so a row in the query result set may include a null value for a column that is part of the primary key in the underlying schema. The query semantics model is defined as follows:

Definition 2: (Query Semantics) Consider a query Q that is issued on behalf of some purpose-recipient pair Pj and that refers to table T. Query Semantics is enforced as follows:

1. Every table T in the FROM clause is replaced by T_{Pj}, defined as follows:

$$\{r | \exists t \in T \quad \wedge \forall i, 1 \leq i \leq n \\ (r[i] = t[i] \ if \ eval(t[i,j]) = true, \\ r[i] = null \ otherwise)\}$$

2. Result tuples that are null in all columns of Q are discarded.

For example, suppose we were to project the `Name` and `Age` columns from the Patients table. Using query semantics, the result of this query would be the table on the right of Figure 5; using table semantics, we would obtain the table on the left.

Note that in the query semantics model, different project lists in otherwise identical queries might yield different numbers of rows depending on the column(s) projected. This slight departure from the norms of conventional SQL may result in substantial performance gains, but the semantic tradeoff should nonetheless be carefully considered.

3.3 Representing a Prohibited Value as Null

In SQL, null is a special value meant to denote "no value" [11]. For this reason, it is intuitive to use null to represent a prohibited value. Adopting the semantics of SQL queries run against null values for prohibited values is desirable for several reasons:

- Predicates applied to null values, such as `X > null`, do not evaluate to true. Similarly, null values do not join with other null values. Predicates applied to privacy-enforced tables will thus behave as though the prohibited cells were not present.
- Null values do not affect computation of aggregates, so an aggregate is actually computed based only on the values available to the purpose and recipient.
- Many applications are written to withstand nulls in the query results. Such applications can be privacy-enabled without requiring expensive rewriting.

However, we also carry over the well-known semantic anomalies inherent in the use of null values [11]. For example, the SQL expression `AVG(Age)` is not necessarily equal to the expression `SUM(Age)/COUNT(*)`. One might expect that an expression such as `SELECT * FROM Patients WHERE Age > 50 OR Age <= 50` will return all tuples in `Patients`, but it might not do so in the presence of nulls.

Replacing prohibited values with nulls also makes some practical assumptions. While it is not its intended use, null may sometimes be used with application-specific semantics. For example, an application may treat a null value in the `Phone` column as an indication that a patient has no phone; the use of null for prohibited fields might conflict with such usage. The alternative to using nulls is to introduce a new special data value, *prohibited*, carrying special semantics with regard to SQL queries, but such an approach would require substantial augmentation to the query processing engine and add new semantic complexities.

4 Implementation

This section details the implementation of the architecture introduced in Section 2. We first describe the mechanisms used to store privacy meta-data, including rules and conditions as implemented using the policy meta-language. We then describe two algorithms for modifying queries to incorporate privacy enforcement. Finally, we describe an optimized implementation for enforcing opt-in and opt-out choices.

4.1 Privacy Meta-Data: Rules and Conditions

The disclosure rules from an electronic privacy policy are stored inside the database as the *privacy meta-data*. These tables capture the purpose and recipient information (Figure 6), as well as disclosure conditions (Figure 7). When a purpose P, recipient R, and data column D of table T appear in a row of the policy

RuleID	Policy	Purpose	Recipient	Table	Column	CondID
R1	P1	Insurance	Billing Office	Patients	Phone	-
R2	P1	Solicitation	External Charity	Patients	Name	C5
R3	P1	Solicitation	External Charity	Patients	Phone	C3
R4	P1	Treatment	Hospital Nurses	Patients	Disease	C1
R5	P1	Treatment	Hospital Nurses	LabResults	Diagnosis	C2
R6	P1	Solicitation	External Charity	Patients	P#	C4
R7	P2	Insurance	Billing Office	Patients	Address	-
R8	P2	Insurance	Billing Office	Patients	Phone	-

Figure 6: Policy Rules Table

CondID	Predicate
C1	"EXISTS (SELECT NurseID FROM Nurses WHERE Patients.floor = Nurses.floor AND $USERID = Nurses.NurseID)"
C2	"EXISTS (SELECT NurseID FROM Nurses, Patients WHERE Patients.floor = Nurses.floor AND Patient.P# = LabResults.P# AND $USERID = Nurses.NurseID)"
C3	"EXISTS (SELECT Phone_Choice FROM PatientChoices WHERE Patients.P# = PatientChoices.P# AND PatientChoices.Phone_Choice = 1)"
C4	"EXISTS (SELECT ID_Choice FROM PatientChoices WHERE Patients.P# = PatientChoices.P# AND PatientChoices.ID_Choice = 1)"
C5	"EXISTS (SELECT Name_Choice FROM PatientChoices WHERE Patients.P# = PatientChoices.P# AND PatientChoices.Name_Choice = 1)"

Figure 7: Conditions table

table, this denotes a rule, which indicates that D is available to recipient R for purpose P. If this row contains a condition, it means that (P, R) has access to D, but with restrictions as indicated by the condition. For example, the rules described in Figure 6 indicate that, under policy *P1*, `Phone` information is always provided to the billing office for the purpose of processing insurance claims (Rule R1), but it is provided to external charities for solicitation only conditionally, on an opt-in or opt-out basis (Rule R3)

4.2 Query Modification

We have two algorithms for query modification that are described in this section, one using case-statements, and the other using outer-joins.

4.2.1 Case-Statement Modification

The first query modification algorithm augments incoming queries with case statements to enforce the rules and conditions expressed in the privacy meta-data. Consider, for example, a data table `Patients`, containing an attribute `Phone`. Under the privacy policy that is in place, the `Phone` attribute is made available to charities for the purpose of solicitation on an opt-in basis, as is the primary key, `P#`. Suppose the query `SELECT Phone FROM Patients` is issued for this recipient and purpose. Using the table semantics model, this query can be rewritten to resolve the condition as follows, where the choice condition on `Phone` is used to perform cell-level enforcement, and the condition on `P#` is used for record filtering:

```
SELECT
CASE WHEN EXISTS
    (SELECT Phone_Choice
    FROM PatientChoices
    WHERE Patients.P# = PatientChoices.P#
    AND PatientChoices.Phone_Choice = 1)
THEN Phone ELSE null END
FROM Patients
WHERE EXISTS
    (SELECT ID_Choice
    FROM PatientChoices
    WHERE Patients.P# = PatientChoices.P#
    AND PatientChoices.ID_Choice = 1)
```

The basic modification algorithm is given in Figure 8.[6] Through a series of simple lookup queries to the privacy meta-data tables, the algorithm resolves the appropriate rules, and modifies the query appropriately. The given algorithm filters records in accordance with the table semantics model, but the filtering method for query semantics applies predicates similarly.

An optimization not reflected in the above pseudo-code is available when the original query includes a predicate on an indexed column. Consider for example the query `SELECT Phone, Name FROM Patients WHERE Phone = 222-2222` over a data table in which there is an index on `Phone` and `Name` is non-indexed. For simplicity, in this and subsequent query modification examples we refer to the predicate implementing condition "C3" for table `Patients` as `C3`, etc. We also disregard row filtering and translate the query to:

```
SELECT Phone, Name
FROM (SELECT CASE WHEN C3
    THEN Phone ELSE null END,
    CASE WHEN C5
    THEN Name ELSE null END
    FROM Patients) AS q1(Phone, Name)
WHERE q1.Phone = 222-2222
```

This query cannot use the index on `Phone` because the reference to `Phone` is embedded inside a case-statement. We fix this problem by pulling the indexed data attribute and the corresponding choice out to the predicate, where the index can more easily be used,

[6]The query translation techniques are described to simplify exposition, not in their optimized form. We assume that the queries are further rewritten by the query optimizer.

Modify_Query
Input: Query string Q to be modified, unique policy identifier PID, Purpose P, Recipient R
Output: Query string Q', reflecting the privacy semantics of P and R under policy PID
Method: For each table t referenced by query Q, (1) replace the reference to t with a sub-query that reconciles the semantics associated with each column in the table based on the policy and conditions, and (2) add a predicate that filters rows where at least one attribute of the primary key is forbidden. *GetPolicy* and *GetCondition* are functions of PID, P, R, t, and column c that query the privacy meta-data tables. *GetPolicy* resolves the privacy semantics of a column for the given purpose and recipient, and returns a value of "allowed", "prohibited", or "condition." *GetCondition* retrieves the appropriate conditional predicate. $Eval(condition, row)$ is a boolean function that evaluates a particular condition predicate over an individual row. We use bracket <> notation to distinguish commands executed at query runtime from those executed at query modification time.

$Q' = Q$
for all Tables t referenced by Q **do**
  $C[\,] \leftarrow$ column list of t
  for $i = 0$ to length of C **do**
    if $GetPolicy(PID, P, R, t, C[i])$ = allowed **then**
      $C'[i] \leftarrow C[i]$
    else if $GetPolicy(PID, P, R, t, C[i])$ = prohibited **then**
      $C'[i] \leftarrow null$
    else
      /* Replace with a column function that resolves the condition on a per-row basis */
      $condition \leftarrow GetCondition(PID, P, R, t, C[i])$
      $C'[i] \leftarrow$ <**if** $Eval(condition, row)$ **then** $C[i]$ **else** $null$ **endif**>
    end if
  end for
  $K[\,] \leftarrow$ the primary key columns of t
  /* Define a predicate function *predicate* to filter individual rows based on the semantics of the primary key attributes */
  $predicate \leftarrow$ <$\neg(\exists k \in K$ such that $Policy(PID, P, R, t, k)$ is forbidden) $\wedge$ $\neg(\exists k \in K$ such that $\neg Eval(GetCondition(PID, P, R, t, k), row)$>
  Create selection subquery t' with projection list C'; Append predicate function *predicate* to t'
  Replace reference to t in Q' with t'
end for
return Q'

Figure 8: Basic algorithm for modifying queries for privacy enforcement using table semantics.

but the meaning of the query is still consistent with the desired semantics[7]:

```
SELECT Phone, Name
FROM (SELECT Phone,
    CASE WHEN C5 THEN Name ELSE null END,
    FROM Patients) AS q1(Phone, Name)
WHERE q1.Phone = 222-2222 AND C3
```

4.2.2 Outer Join Modification

An alternative modification mechanism implements the Table Semantics and Query Semantics enforcement models using the left outer join and full outer join operators, respectively. Consider the query `SELECT Phone FROM Patients` from the previous section. This query can be rewritten as follows to reflect the table semantics enforcement model:

```
SELECT Phone FROM
(SELECT P# FROM Patients WHERE C4) AS t1(P#)
LEFT OUTER JOIN
    (SELECT P#, Phone
    FROM Patients
    WHERE C3) AS t2(P#, Phone)
ON t1.P# = t2.P#
```

[7]Thank you to Jerry Kiernan and Ramakrishnan Srikant for pointing out this fix. Note that this optimization also requires modification of the correlated sub-query in condition predicate C3, and it necessitates a dynamic rewriting mechanism which cannot be applied if we simply define a view for each purpose-recipient pair.

The modification algorithm for table semantics is a SQL implementation of the following relational algebra expression; we omit pseudo-code. Consider some query Q. Each table T referenced by Q contains some columns, $a_1 \ldots a_n$. Let k represent the primary key of T, and for simplicity assume that the primary key is comprised of just one column. We replace Q's reference to T with the following, where "$\ltimes$" denotes the left outer join operator:

$$[\sigma_{k=\text{“}Allowed\text{”}}(\Pi_k(T))] \ltimes_{\$1=\$1} [\sigma_{a_1=\text{“}Allowed\text{”}}(\Pi_{k,a_1}(T))] \ltimes_{\$1=\$1} \cdots \ltimes_{\$1=\$1} [\sigma_{a_n=\text{“}Allowed\text{”}}(\Pi_{k,a_n}(T))]$$

We have a similar algorithm for query semantics. Consider a query Q which projects a set of columns from some set of tables. For each such table T, let $p_1 \ldots p_n$ denote the columns of T projected by Q, and let k be the primary key of T. Again assume the primary key contains just one column. We replace the reference to T by Q with the following, where "$\times$" denotes the full outer join operator:

$$[\sigma_{p_1=\text{“}Allowed\text{”}}(\Pi_{k,p_1}(T))] \times_{\$1=\$1} [\sigma_{p_2=\text{“}Allowed\text{”}}(\Pi_{k,p_2}(T))] \times_{\$1=\$1 \vee \$3=\$1} \cdots \times_{\$1=\$1 \vee \$3=\$1 \vee \ldots} [\sigma_{a_n=\text{“}Allowed\text{”}}(\Pi_{k,a_n}(T))]$$

The SeaView system took a similar approach in constructing cell-level access control, recovering multilevel relations from the underlying relations using the left outer join and union operators [18].

4.3 Optimized Implementation of Opt-In/Opt-Out Conditions

We have described how query modification can be used to handle general data-driven conditions. In this section, we describe how the database can provide high-performance support for the important special class of conditions that are expressed as simple opt-in or opt-out choices.

There are several possible approaches to storing choice values. The simplest approach, termed the *internal* design, appends additional columns to the data table (one per choice), where it stores the binary choices (1 denotes consent). While this approach may be satisfactory in some cases, it is preferable to avoid schema modification when adding privacy support to an existing database environment. For this reason, we explore options for storing choice values *externally*, in tables separate from the data tables.

The *external multiple table* design uses one table per choice. The schema of each external choice table consists of a foreign key that references table T. The table C_i corresponding to choice i contains one row for each row of T for which the data subject provided consent for the ith choice. Thus, if the data table T were extended with n choices to yield the table T', table $C_i = \pi_{key}(\sigma_{choice_i=1}(T'))$ for choice i.

These external choice tables can be used to enforce limited disclosure using either the table semantics or query semantics enforcement model. We describe in detail only the implementation of table semantics, but query semantics follows handily. Let V_C refer to a view of the choice tables corresponding to the columns in table T. Let *key* choices refer to those that involve a key column in T, and let *non-key* choices refer to those that do not. A row of T is not visible unless the subject has opted in for all key choices; therefore, the corresponding choice tables are combined using joins to produce the view V_C. A non-key choice determines whether the corresponding column is visible (assuming that the row is visible according to the key choices). This is enforced by using left-outer join to add each non-key choice table to V_C. V_C is then joined with T, and the following condition is tested by the modified query (using either modification algorithm): If the choice field (generated from the outer join of the corresponding external table) has a value other than *null*, then the condition is satisfied and the data returned; otherwise the data is forbidden and replaced with *null*.

The definition of view V_C is illustrated below for an example data table T with a key column K, a choice (C_0) on this column, and choice C_1 on non-key column A. For the multiple external table design, V_C is defined as:

```
SELECT C0.key, C1.key
FROM C0 LEFT OUTER JOIN C1 ON C0.key = C1.key
```

Alternatively, the *external single table* design replaces the multiple external tables with a single table C_C. This design essentially stores the choice columns described for the internal design in a separate table which can be joined with the data table. The schema contains the key for data table T and n choice columns. The basic query modification algorithm is similar to that of the internal design, though the data and choice tables must first be joined.

5 Performance Evaluation

We describe here the results of experiments studying the performance of our architecture and of query modification as a method of enforcing limited disclosure. The primary focus of these experiments is to measure the performance of enforcing unconditional policy rules and those containing opt-in and opt-out conditions, as these are by and large the most common cases. Our experiments are intended to address the following key questions:

- **Overhead of Privacy Enforcement** What is the overhead cost introduced by privacy checking? We address this question through an experiment that factors out the impact of choice selectivity, incurring the cost of checking privacy semantics, but gaining nothing from filtering prohibited tuples from the result set.
- **Scalability** We test the scalability of our modification mechanism in terms of database size and application selectivity. We vary both the percentage of users who elect to share their data for a particular purpose and recipient (*choice selectivity*)[8], and the percentage of the records selected by an issued query (*application selectivity*).
- **Impact of Filtering** In both the table and query semantics models, there are cases where tuples are filtered entirely from the result set of a query. We perform an experiment to show the performance gain due to this filtering.
- **Choice Storage** We compare the performance of the internal, external single, and external multiple choice table designs for storing choices.
- **Enforcement Model** We describe the performance implications of choosing between Table Semantics and Query Semantics models.
- **Modification Algorithm** We compare the performance of the case-statement and outer join modification algorithms for different opt-in and opt-out scenarios.

There are several distinct sources of performance cost in our architecture. There is some cost incurred by rewriting queries, but this cost is small, and constant in the number of columns. Moreover, this step is likely

[8]Except where otherwise noted, our experiments use cell-level enforcement, but make the simplifying assumption that access to all columns in the data table is based on a single opt-in/opt-out choice. This means that every record is either fully visible or fully invisible; however, for the case-statement rewrite mechanism we still perform cell-level enforcement by evaluating a case statement over each column.

Column	Description
Unique2 (int)	Primary key, Sequential order
Unique1 (int)	Candidate key, random order
Onepercent (int)	Values 0-99, random order
Tenpercent (int)	Values 0-9, random order
Twentypercent (int)	Values 0-4, random order
Fiftypercent (int)	Values 0-1, random order
stringu1 (32-byte str)	Unique character string
stringu2 (32-byte str)	Unique character string
Choice_0 (int)	Values 0-1 (1% = 1), indexed
Choice_1 (int)	Values 0-1 (10% = 1), indexed
Choice_2 (int)	Values 0-1 (50% = 1), indexed
Choice_3 (int)	Values 0-1 (90% = 1), indexed
Choice_4 (int)	Values 0-1 (100% = 1), indexed

Figure 9: Benchmark dataset and choice columns.

circumvented altogether for pre-compiled queries. For these reasons, we focus on the cost of query execution.

5.1 Experimental Setup

We evaluate performance using the synthetically-generated dataset described in Figure 9, which is based on the Wisconsin Benchmark [13]. All experiments were run using DB2 UDB 8.1. The operating system was Microsoft Windows 2000 Server, Service Pack 4. The hardware consisted of a dual-processor 1.8Ghz AMD machine with two GB of memory and four 80 GB IDE Western Digital hard drives. The data was spread across two of the disks. The buffer pool size was set to 50MB, the pre-fetch size was set to 64KB, and the level of optimization was set to 5. All other DB2 default settings were used.

To measure the cost of executing queries, we used the DB2batch utility. Each query was run 6 to 11 times, flushing the buffer pool, query cache, and system memory between unique queries. The results below give the average warm performance numbers for each query. With 95% confidence, the margin of error for the reported numbers is less than ±5%.

5.2 Experimental Results and Analysis

5.2.1 Overhead and Scalability

Our first set of experiments measures the overhead cost of performing privacy enforcement and the scalability of our algorithms to large databases. To measure this cost, we consider simple queries that select all records from the data table. We report the results for our table semantics enforcement model, but the trends are similar for query semantics. We consider the worst case scenario, where the choice selectivity is 100%. Here we incur all the cost of privacy processing, but we do not see the performance gains of filtering. The application selectivity was kept fixed at 100%.

Figure 10 shows the overhead cost of executing queries modified for privacy enforcement over tables containing 1, 5, 10, and 15 million records. The graph shows the total execution time for unmodified "`SELECT *`" queries, as well as queries modified to incorporate choices stored using the internal and external multiple approaches. The queries incorporating internal choices

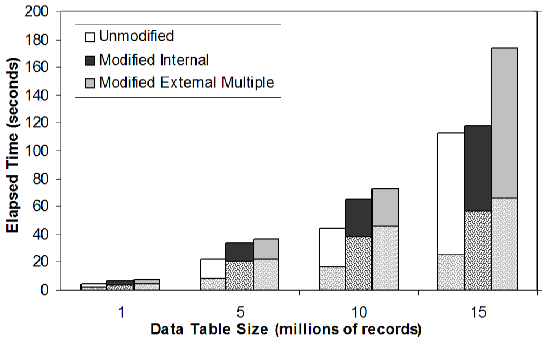

Figure 10: Scalability and performance of modified and unmodified queries, choice selectivity = 100% and application selectivity = 100%. CPU time is the bottom portion of each bar, and the full bar represents the total query execution time.

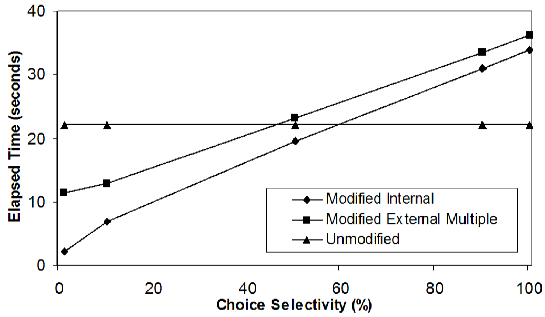

Figure 11: Comparing original and modified queries (5 million records, application selectivity = 100%)

incur the overhead of processing the additional case statement for each cell, and the external also incurs the cost of joining with the choice table. Note that the join is performed quickly as both the data and choice tables are clustered on the join key.

Because the figures show the warm numbers, the queries over the smaller tables can largely be processed from the buffer pool and system memory. In the case of the 15 million-record table, however, the query processing becomes I/O-bound. Thus, in the case of the former, the cost is dominated by the CPU time spent processing the case statements, whereas in the latter, the cost is dominated by I/O. Here the relative cost of performing privacy checking is reduced as the database size grows. We observe an increase in elapsed time for the external strategy and the 15 million-record table because of the extra I/O cost of reading the indexes on the data and choice tables.

Overall, the overhead cost of privacy-checking is small, particularly for the internal choice storage design. Because the cost of privacy-checking is largely CPU-based, privacy enforcement scales well to larger queries in which I/O dominates the cost.

5.2.2 Impact of Record Filtering

In cases with choice selectivity less than 100%, queries modified to reflect the table semantics or query se-

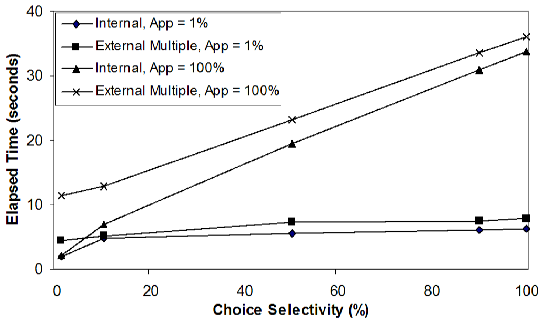

Figure 12: Comparing the internal and external multiple strategies (5 million records table)

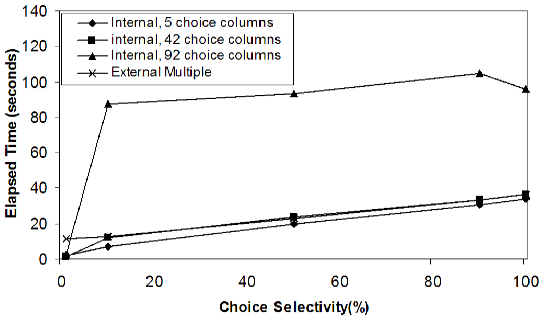

Figure 13: Comparing storage strategies for varying number of choice columns (5 million records, application selectivity = 100%)

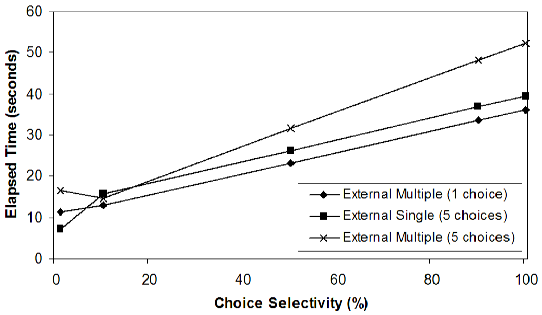

Figure 14: Comparing external multiple and external single for multiple choices (5 million records, application selectivity = 100%)

mantics enforcement model perform significantly better that the original queries. Because the choice values are indexed, these queries avoid reading tuples that are filtered from the result set. Figure 11 compares the performance of original and rewritten queries over a table containing five million records. In our experiment, the queries with choice selectivity 1% and 10% used the index on the choice column; the others did not. The performance gain is considerable when the choice selectivity is 1% or 10%.

5.2.3 Comparing Choice Storage Methods

There are a number of performance issues to consider when choosing a choice storage design. In particular, when using an external design, it is necessary to join the data table with the choice table when processing a query. Using our original experimental dataset, we found that the internal approach performed better than the external multiple approach (Figure 12). However, on further investigation, we found that the cost of performing this join may be offset when the number of choice columns stored internally is large. In addition to database design considerations, this performance trade-off should be considered when choosing a choice storage design.

To measure this performance trade-off, we ran experiments varying the number of choices *stored*. The number of choices stored is distinct from the number of choices *enforced* because choices might be stored for a number of different purpose-recipient pairs, but all of these choices are not necessarily enforced for each query. Figure 13 compares the performance of queries enforcing a single choice modified using the internal and external multiple storage strategies. The application selectivity was held constant at 100%. In the case where the total number of stored choices is small, the internal strategy performs slightly better than the external strategy. However, as the number of stored choices grows, the internal strategy widens the data table, causing performance to suffer. The external approach performs similarly to an internal table of width 50. (This is equivalent to our standard benchmark data table, plus 42 choice columns appended internally.) For the tables of one hundred columns and with the internal strategy the queries become I/O bound and perform poorly in comparison.[9]

As the number of choices stored increases, performance of the internal strategy deteriorates. On the other hand, as the number of choices enforced increases, the number of joins required for the external multiple strategy grows, causing performance to suffer. In this case, we can substitute the external single strategy to obtain better performance. Figure 14 shows the results of an experiment varying the number of choices enforced; the number of stored choices was held constant at five. This represents a tradeoff because the choice table will likely contain more records than a single choice table from the external multiple design, and like the internal design, the single external table could potentially be wide. To partially address this concern, it is possible to create several external choice tables, grouped by choice values.

5.2.4 Performance Differences Among Enforcement Models

As we saw in Section 5.2.2, record filtering can have a significant impact on performance. For this reason, there is a clear performance distinction between

[9]It is also possible to encode choices as a bit vector in one (or a small number of) columns. We do not consider this option because in our implementation the necessary bitwise operations precluded the use of an index on the choice values.

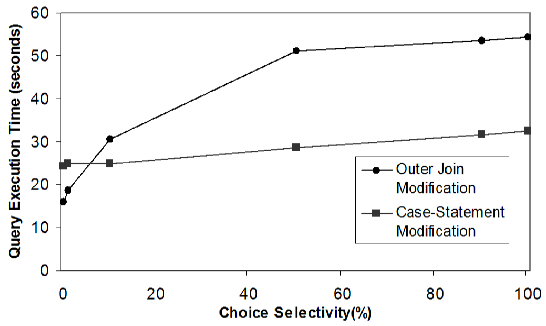

Figure 15: Comparing modification algorithms for a "sparse" choice space (internal choice storage, 5 million records, application selectivity = 100%)

the table semantics and query semantics enforcement models. In the table semantics model, a tuple is not filtered from the result set if the primary key is not prohibited, although all of the non-key columns might have been prohibited. Assume that the key columns of a table have independent privacy rules and that the primary key columns are allowed if any of the non-key columns of the record is allowed. In this case, it is convenient to think of the independent choice selectivities for all of the columns in the table combining to form the *effective choice selectivity*. If the table contains x columns, the effective selectivity can be determined by $1 - \prod_{i=1}^{x}(1 - s_i)$, where s_i is the choice selectivity corresponding to column i. Thus fewer rows are filtered as the number of columns in the table increases.

In the query semantics model, the effective choice selectivity is determined by the selectivities of only those columns projected by the query. In many situations, therefore, using the query semantics model will likely lead to substantial performance gain.

5.2.5 Comparing Modification Algorithms

In most situations, our case-statement modification algorithm outperforms the outer-join algorithm. However, there are some situations where outer join does better, particularly when the space of choice values is "sparse", meaning few data subjects have provided consent for a particular column, but many of the records must still be fetched because of other data columns. In this case, the outer join algorithm is able to make use of the index over the "sparse" choice column. Thus, it avoids processing a case-statement over each of these cells.

Figure 15 shows the performance of the outer join algorithm compared to case-statements over a table with the following privacy semantics: `Unique2` can be unconditionally disclosed, but the rest of the columns are provided based on some choice, the selectivities of which are shown in the graph.

5.2.6 Summary of Performance Results

Through our experiments, we found that in general the cost of privacy checking is small and CPU intensive. For large I/O bound queries, the relative cost is minimal, so the cost of privacy enforcement scales well. Both semantic models, Table Semantics and Query Semantics, filter records from the result set in addition to performing cell-level enforcement. This filtering leads to substantial performance gains when the choice selectivity is small. We expect that in many situations, the query semantics model would filter more records than would the table semantics model, and for this reason, we expect that it would generally show superior performance.

There is a performance tradeoff to be considered when choosing between the internal, external multiple, and external single choice storage designs. When the number of stored choices is high, the performance of the internal design suffers because of an increase in I/O due to the increase in the width of the records. When the number of choices enforced is high, the performance of external multiple suffers because of the number of required joins. Frequently, the external single design serves as an effective compromise.

Finally, the case-statement modification algorithm almost always outperforms the outer-join mechanism, except in certain cases, such as sparse choice spaces.

6 Conclusion and Future Work

Limited disclosure is a vital component of a data privacy management system. We proposed a scalable architecture for enforcing limited disclosure rules and conditions at the database level, and we presented several models for cell-level limited disclosure enforcement in a relational database. Application-level solutions are unable to process arbitrary SQL queries efficiently. By pushing the enforcement down to the database, we gain improved performance and query power. This mechanism can be deployed without modifying legacy application code or existing database schemas. We showed that the performance overhead of database-level privacy enforcement is small, and often times the overhead is more than offset by the performance gains obtained through record filtering.

The work reported here has broader applicability than the Hippocratic databases. Specifically, our techniques can be used in any application requiring policy-driven fine-grained access and disclosure control.

There are several important extensions to this architecture that are areas of ongoing and future work. One such extension would allow us to assign versions to privacy policies. In this case, personal data would be permanently associated with the policy in place at the time of collection. The database would then be responsible for enforcing these multiple policies as queries are issued. Another such extension would provide granular privacy enforcement for data modification commands.

7 Acknowledgements

Our thanks to Jerry Kiernan for his invaluable help in implementing query modification and discussions about EPAL and cell-level enforcement semantics, to Ramakrishnan Srikant for discussions about choice storage and query modification, and to Ameet Kini and Diana Zhou for their work on the prototype implementation.

References

[1] US Department of Health and Human Services. http://www.hhs.gov/ocr/hipaa.

[2] Vignette Corportation. www.vignette.com.

[3] Nov. 2003. Personal communications with Sushil Jajodia.

[4] The Lowell database research self assessment, June 2003.

[5] N. Adam and J. Wortman. Security-control methods for statistical databases. *ACM Computing Surveys*, 21(4):515–556, Dec. 1989.

[6] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, Hong Kong, China, August 2002.

[7] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise privacy authorization language 1.2 (EPAL 1.2). W3C Member Submission, November 2003.

[8] P. Ashley and D. Moore. Enforcing privacy within an enterprise using IBM Tivoli Privacy Manager for e-business, May 2003.

[9] D. Bell and L. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp., Bedford, Mass., March 1976.

[10] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison Wesley, 1995.

[11] D. Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan Kauffmann, San Francisco, California, USA, 1998.

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

[13] D. DeWitt. The Wisconsin benchmark: Past, present, and future. In J. Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann, 1993.

[14] P. Griffiths and B. Wade. An authorization mechanism for a relational database system. In *SIGMOD*, Washington, DC, June 1976.

[15] S. Jajodia and R. Sandhu. Polyinstantiation integrity in multilevel relations. In *IEEE Symposium on Security and Privacy*, May 1990.

[16] S. Jajodia and R. Sandhu. A novel decomposition of multilevel relations into single-level relations. In *IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 1991.

[17] N. Kabra, R. Ramakrishan, and V. Ercegovac. The QUIQ Engine: A hybrid IR-DB system. In *ICDE*, Bangalore, India, March 2003.

[18] T. Lunt, D. Denning, R. Schell, M. Heckman, and W. Shockley. The SeaView security model. *IEEE Transactions on Software Eng.*, 16(6):593–607, June 1990.

[19] A. Nanda and D. Burleson. *Oracle Privacy Security Auditing*. Rampant, 2003.

[20] X. Qian and T. Lunt. Tuple-level vs. element-level classification. In *Database Security, VI: Status and Prospects. Results of the IFIP WG 11.3 Workshop on Database Security*, Vancouver, Canada, August 1992.

[21] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2003.

[22] A. Sahuguet, R. Hull, D. Lieuwen, and M. Xiong. Enter once, share everywhere: User profile management in converged networks. In *CIDR*, Asilomar, CA, January 2003.

[23] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996.

[24] M. Stonebraker and E. Wong. Access control in a relational data base management system by query modification. In *ACM/CSC-ER*, 1974.

[25] G. Wiederhold, M. Bilello, V. Sarathy, and X. Qian. A security mediator for healthcare information. In *AMIA Conference*, Washington, DC, Oct. 1996.

[26] L. Willenborg and T. deWaal. *Elements of Statistical Disclosure Control*. Springer Verlag, 2000.