Limiting Disclosure in Hippocratic Databases

Research Paper ID #174

ABSTRACT

Preserving data privacy is of utmost concern in many sectors, including e-commerce, healthcare, government, and retail, where individuals entrust others with their personal information every day. Often, the organizations collecting the data will specify how the data is to be used in a privacy policy, which can be expressed either electronically or in natural language. We describe a data model for enforcing these limited disclosure rules in a relational database at cell-level granularity. We then present a practical and efficient architecture and algorithms for implementing this model. 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 [7]. The authors of [10] proposed the vision of a "Hippocratic" database, a database system that is responsible for maintaining the privacy of the personal information it manages. The authors propose a framework for managing privacy sensitive information distilled down from the private data handling practices that are being demanded internationally, and mandated through legislation such as the United States Privacy Act of 1974 (Fair Information Practices), the EU Privacy Directive, which took effect in 1998, the Canadian Standard Association's Model Code for the protection of Personal Information, the Australian Privacy Amendment Act of 2000, the Japanese Personal Information Protection Laws of 2003, and others. The framework is based on ten broad principles central to managing private data responsibly.

A vital principle among these is "limited disclosure," which the authors define to mean that the database should not communicate private information outside the database for reasons other than those for which there is consent from the information donor[1]. A straightforward solution would be to implement this enforcement at the application, middleware, or mediator level, as is done in Tivoli Privacy Manager[11] and the TIHI security mediator[26]. However, this approach leads to privacy leaks when applied to cell-level privacy enforcement, as we discuss in Section 2.5.

An ideal solution to the limited disclosure problem would flexibly protect donor information without leaks, and would incur minimal privacy "checking" overhead when processing queries. Because of the time and expense required to modify existing application code, such a solution would require minimal change to existing applications.

In this paper, we introduce a mechanism for enforcing limited disclosure. The main idea behind our solution is storing privacy policies and user choices inside the database, and rewriting incoming queries to reflect these privacy semantics. Specifically, we intercept an incoming query, and augment the query as necessary to reflect both the privacy policy and the donor's preferences. We that our design meets the desiderata just mentioned.

1.1 Related Work

This paper builds on work in the area of data security, which can largely be grouped into the areas of discretionary access control, role-based access control, and mandatory access control [23]. Discretionary access control allows a database to grant and revoke access privileges to individual database users. In this case, the access control privileges typically refer to entire tables or views. Role-based access control allows a database to grant this type of privilege not to an individual user, but the the user's group, or role [25]. In the mandatory access control model, there is a single set of rules governing access to the entire system, and individual users are not allowed to grant or revoke access privileges.

A well-known model of mandatory access control, the Bell-LaPadula model of multilevel secure databases, defines permissions in terms of objects, subjects, and classes [13]. Each object is a member of some class, for example "Top Secret,' "Secret", and "Unclassified," and in this model, the classes typically form a hierarchy. Multi-level databases also allow for the possibility of polyinstantiation, where there exist data objects that appear to have different values to users with different classifications [16]. These formalizations have been further refined by [19] and [20], and a schema decomposition allowing element-level classification to be expressed

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

as tuple-level classification is described in [22].

The Oracle 8i product implements some of these ideas in its "Row Level Security" (also known as "Virtual Private Database") feature, which allows specification of security policies at the row level, and augments incoming queries with additional predicates to reflect the security policy[1]. Multi-level security was also implemented in Sybase's Secure SQL Server[4] and Informix OnLine/Secure[3], and work was done to benchmark row-level classification in multi-level secure database systems[18]. The notion of "reformulating" queries for security was also alluded to by[26], and [8] uses a query rewrite mechanism to control access to federated XML user-profile data.

In some ways, our ideas can be viewed as an adaptation of the ideas and semantics of multi-level and role-based access control. Our problem considers the task of assigning (purpose, recipient) pairs (the subjects) access to data cells (objects), which are grouped into data categories (classes). However, the privacy problem requires an additional degree of flexibility. In the limited disclosure problem all data assigned to a particular category does not necessarily have the same access semantics because of conditional rules, like opt-in and opt-out choices. This leads to more complex permissions management. However, our problem also allows for an important key simplification, as we need not allow polyinstantiation of data.

As far as we could determine, the only implementation of a DBMS with cell-level access control was done by SRI in the SeaView system[16], but a performance evaluation was never published[5]. Numerous 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][21], but they are application-specific in their design and do not extend a DBMS for general use. We provide a high-performance cell-level solution to the limited disclosure problem that extends a DBMS with support for limited disclosure, and can be deployed to an existing environment without modification of existing applications, and we study the performance implications of such a system. The wide use of fine-grained security by applications offers additional evidence that extending a DBMS with this capability is overdue.

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 [9], and [27]). 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 donors' preferences and choices, and we assume that additional mechanisms such as query admission control and audit trails [9] are in place to guard against the inference problem.

1.2 Paper Organization

This paper is organized as follows. First, we introduce the limited disclosure problem as it relates to a relational database. We then describe several a limited disclosure models for relational data and their semantics. We describe a basic implementation architecture for limited disclosure and some optimizations to this architecture. Finally, we evaluate the performance of our implementation, and point out topics of future research.

2. LIMITED DATA DISCLOSURE

One of the defining principles of data privacy, limited data disclosure is based on the premise that information donors should be given control over who is allowed to see their personal information, and under what circumstances. For example, patients entering a hospital must provide some information at the time of admission. The patient understands that this information may only be used under certain circumstances. The doctors may use the patient's medical history for treatment, and the billing office may use the patient's address information to process insurance claims. However, the hospital may not give patient address information to charities for the purpose of solicitation without consent.

Frequently, an organization will define a *privacy policy* describing such an agreement. Comprised of a set of rules, the privacy policy is a contract between the individual providing the information and the organization collecting the information. Data items are classified into categories. We assume for simplicity that these categories are mutually exclusive. For each category of data, the rules in the privacy policy describe the class of individuals who may access the information (the *recipients*), and how the data may be used (the *purposes*). The policy may specify that the data items belonging to a category may be disclosed, but only with "opt-in" consent from the donor. The policy may also specify that data items belonging to a category will be disclosed unless the donor has specifically "opted-out" of this default. There is much existing work regarding electronic privacy policy definition[6][12][15].

A solution to the problem of limited disclosure would ensure that the rules contracted in these privacy policies are enforced. More specifically, each query issued to the database would be issued in conjunction with a particular purpose and recipient. The database would prohibit the outflow of data, except when the privacy policy includes a rule permitting disclosure of the data to the appropriate purpose and recipient. In our hospital example, a query issued for the purpose of "solicitation" and recipient "external charity" would only reveal the personal information of those patients who provided consent.

2.1 Limitations of Tuple-Level Enforcement

Consider a table containing patient information, as shown in Figure 1. The data items "Name" and "Age" have been grouped into the data category "Personal Information." Similarly, "Address" and "Phone" have been included in the "Address Information" category. 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 2 shows the choices made by the patients.

With row-level enforcement, clearly Alice's record should be visible to charities for solicitation, and Bob's record should be invisible. However, there is a problem with the records of Carl and David. In this case, we must either filter information that is actually permitted, or we must disclose information that is prohibited. In the following sections, we first describe, and then formally define, three models of cell-level enforcement.

2.2 Strict Cell-Level Enforcement

The above problem can be solved by defining a model of

Patient#	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 1: Full data table of patient information.

Patient#	ID Info	Personal Info	Address Info
1	√	√	√
2	×	×	×
3	√	×	√
4	√	√	×

Figure 2: Patient choices for disclosure of information to charities for solicitation.

Patient#	Name	Age	Address	Phone
1	Alice Adams	10	1 April Ave.	111-1111
-	-	-	-	-
3	-	-	3 Cricket Ct.	333-3333
4	David Daniels	40	-	-

Figure 3: Privacy-enforced table of patient information, using strict cell-level enforcement.

Patient#	Name	Age	Address	Phone
1	Alice Adams	10	1 April Ave.	111-1111
3	-	-	3 Cricket Ct.	333-3333
4	David Daniels	40	-	-

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

Name	Age
Alice Adams	10
-	-
David Daniels	40

Name	Age
Alice Adams	10
David Daniels	40

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

cell-level enforcement. One way of defining such a model would be to "mask" prohibited values using the *null* value. We assign each (*purpose, recipient*) a view of each table, T, in the database. Each view contains precisely the same number of tuples as the underlying table, but prohibited data elements are replaced with *null*. The view corresponding to our hospital example is given in Figure 3. We term this model *Strict Cell-level* enforcement, and we provide a formal definition in Figure 6.

2.3 Table Semantics Limited Disclosure Model

The strict cell-level model is attractive because of its simplicity. However, if we want the privacy enforced data tables to be consistent with the relational data model, we must also ensure that the primary key is never null.

For this reason, we define another cell-level model, which we term *Table Semantics* enforcement. Here, we assign each *(purpose, recipient)* pair a view over each table in the database, and as before, prohibited cells are replaced with null values. However, in this case we allow both entire tuples and individual cells to have privacy semantics. The privacy semantics of the primary key are used to indicate the privacy semantics of the entire tuple. If the primary key is prohibited, then the entire tuple is prohibited. When we apply this model to a table, the result is that we filter prohibited tuples from the result set, and then we replace any remaining prohibited cells with the *null* value, as is done

---

Consider a list of m purpose-recipient pairs $P =< P_1, P_2, ..., P_m >$.

Every table T with n data columns is (conceptually) extended with $m * n$ columns that record opt in/out "choices". Each choice represents a decision by the donor of the data record to allow or disallow access to a given column 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 "choice" column that contains the donor's choice for data column i and purpose-recipient pair j. For any set of columns S in a table T, we use "$t[S]$ *allnull*" to denote that all columns in S are null in tuple t of T. Similarly, "$t[S]$ *nonenull*" denotes that every column in S is non null in tuple t.

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

$$\begin{aligned}\{r|\exists t \in T \quad & \wedge \forall i, 1 \leq i \leq n \\ & (r[i] = t[i] \ if \ t[i,j] = \text{“}Allowed\text{”}, \\ & r[i] = null \ otherwise)\}\end{aligned}$$

Definition 2. Table Semantics Enforcement Let T be a table with n data columns and let K be the set of fields that constitute the primary key of T. For a given purpose-recipient pair Pj, the table T is seen as T_{Pj}, defined as follows:

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

Definition 3. Query Semantics Enforcement Consider a query Q that is issued on behalf of some purpose-recipient pair P_j 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}, i.e., the version of T under Strict Cell-Level semantics.
(2) Result tuples that are null in all fields of Q are discarded.

Figure 6: Definitions for Strict Cell-level, Table Semantics, and Query Semantics enforcement

in[16]. The resulting table of patients from our hospital example is shown in Figure 4, assuming that Patient# is the primary key. We formally define this model of enforcement in Figure 6.

In SQL, *null* is a special value meant to denote "no value" [14]. Intuitively, it makes sense in our problem to use null as a placeholder when a value is not available to a particular purpose and recipient. Adopting the semantics of SQL queries run against null values is desirable for several reasons:

- Predicates applied to null values, such as `X > null`, will not evaluate to true. Because null values are defined this way, predicates applied to privacy enforced tables will behave as though the prohibited cells were not present.
- Similarly, null values do not join with other values. Thus the results of a join query issued to one of the privacy enforced tables will produce results as if the null cells were not present.
- Null values do not affect computation of aggregates, so an aggregate computed over a privacy enforced table is actually computed based only on the values available to the purpose and recipient.

There are some well-documented semantic anomalies inherent in the use of null values [14]. For example, the SQL expression `AVG(Age)` is not necessarily equal to the expression `SUM(Age)/COUNT(*)`. An expression such as `SELECT * FROM Patients WHERE AGE > 50 OR AGE <= 50`, which might be expected to return all tuples in `Patients`, may not do so in the presence of nulls.

Replacing prohibited values with nulls makes some assumptions about the practical meaning of the *null* value. While it is not its intended use, in practice *null* may carry implied semantic meaning. In our hospital example, a null value in the Phone column may indicate that a patient has no phone. To alleviate this problem, one might consider defining a new data value, *prohibited*, carrying special semantics with regard to SQL queries, to act as a placeholder.

2.4 Query Semantics Limited Disclosure Model

The table semantics model defines a view of each data table for each *(purpose, recipient)* pair, based on the associated privacy semantics. These views combine to produce a coherent relational data model for each *(purpose, recipient)* pair, and queries are executed against the appropriate database version.

An alternative to this approach is to do enforcement based on the query itself. Unlike table semantics, here we remove prohibited data from a query's result set based on the *purpose, recipient*, and the *query* itself. We call this the *Query Semantics* enforcement model. For example, using our hospital table, 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. We formally define the query semantics model in Figure 6. Because this model filters records in response to the issued query, and we do not aim to define a version of the underlying relation for each purpose and recipient, a tuple in the query result set may include a null value for an attribute that is part of the primary key in the underlying schema.

This model benefits from the same properties of null values discussed above. However, these semantics cause some anomalies in certain cases. For example, `COUNT` aggregates may observe different numbers of records depending on the column. For example, if the Salary attribute is provided based on a condition, and the Name attribute is provided unconditionally, `COUNT (Name)` and `COUNT (*)` will likely observe higher counts than `COUNT (Salary)`.[2] In some cases these slight semantic departures buy substantial performance gains, as we show in our experimental results, but the semantic tradeoff should be carefully considered.

2.5 Application-level Limited Disclosure

There are several possible approaches to implementing application-level privacy enforcement. One such approach is to first retrieve the requested data from the database, and then apply the appropriate enforcement before returning the data to the user. In a cell-level enforcement scheme, this approach leads to significant difficulties.

For example, consider a query involving a predicate over a privacy-sensitive field: `SELECT * FROM PATIENTS WHERE DISEASE = Hepatitis`, and a patient who chose to disclose his name, but not his disease history. An application-level enforcement scheme might do the following to execute this query: First, the application would issue the query to the database, and retrieve the result set. Then, the application would go through each of the resulting records, and based on the privacy semantics, replace prohibited cells with null. However, this approach is flawed. In the previous example, the query results would contain the patient's records, with the Disease field blocked out. Unfortunately, this allows anyone to conclude from looking at the results that this patient has Hepatitis, even though he had chosen not to share this information.

This type of leakage is not a problem in the table semantics or query semantics model because data values that are not visible to a particular purpose and recipient are removed prior to query execution.

An alternative approach might select all of the Patient data from the database (in our example, this would include all patient records, not just those with a particular disease), and apply the predicate in the application. However, this leads to significant performance problems as it must fetch data unnecessarily from the data. 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.

3. IMPLEMENTATION ARCHITECTURE

We have developed a database architecture for efficiently and flexibly enforcing limited disclosure rules. The basic components of this architecture are the following:

- **Policy definition** Privacy policies must be expressed electronically, and stored in the database where they can be used to enforce limited disclosure.
- **Query modifier** SQL queries entering the database

[2]Thank you to `<removed>` for pointing out this anomaly.

should be intercepted, and augmented to reflect the privacy semantics of the purpose and recipient issuing the query. The results of this new query will be returned to the issuer.

- **Privacy meta-data** This is where we store the additional information that will allow us to determine the correct privacy semantics of an incoming query.

- **Data and Choice Tables** The data is stored in relational tables in the database. User choices (opt-in and opt-out) must also be stored in the database.

In our prototype, privacy policies are defined using P3P [15], and the privacy meta-data is stored in the database as ordinary relational tables. We implement the prototype enforcement module as an extension to the JDBC driver, where queries are intercepted and rewritten to reflect the privacy semantics stored in the privacy meta-data. In our implementation, queries are issued via an HTTP servlet, forcing the use of the secure driver.

There are two ways to determine the purpose and recipient associated with a query. The first possibility is to extend the syntax of a SQL query to include this information. For example, `SELECT * FROM Patients FOR PURPOSE Solicitation RECIPIENT External_Charity`. The second possibility is to infer this information based on the application context, similar to the approach implemented in [1]. Because the first method requires extensions to the query language and modification to existing applications, we elected to use the second option, though the rest of our implementation is compatible with either alternative. Our query interceptor infers the purpose and recipient of the query based on the issuing application. The context of each application must be specified, and in our prototype, we store the context information in an additional database table. We then use this information to tag incoming queries with the appropriate privacy semantics based on the issuing application.

An overview of this architecture is given in Figure 7. In the future, the query interception and modification component may be moved into the database's query processor without changing the general approach. By the same token, the privacy meta-data could be moved to an external mediator database, which would be responsible for intercepting and rewriting the query, as long as the user choices remain in the same database as the donor data. In the following sections, we first describe the basic implementation, showing that it can be applied to any of the limited disclosure models described above. We then describe model-specific adjustments and optimizations.

3.1 Architecture Overview

We store the disclosure rules from a specified privacy policy inside the database, as the *Privacy Meta-data*. These tables capture the purpose and recipient information, as shown in Figure 8, as well as conditions of the form `attribute <opr> value`, which are used to resolve conditional access, such as opt-in and opt-out choices. When a purpose P, recipient R, and data category D appear in a row of the policy table, this indicates that D is available to recipient R for purpose P. If this row contains condition values, it means that P and R may access D, but with restrictions as indicated by the condition. For example, the rules described in Figure 8 indicate that address information is always provided to the billing office for the purpose of processing insurance claims, but address information is provided to external charities for solicitation only on an opt-in or opt-out basis. These tables also capture the identification of the privacy policy corresponding to each rule. We also store mappings of data columns to the broader categories used by privacy policies, as shown in Figure 9.

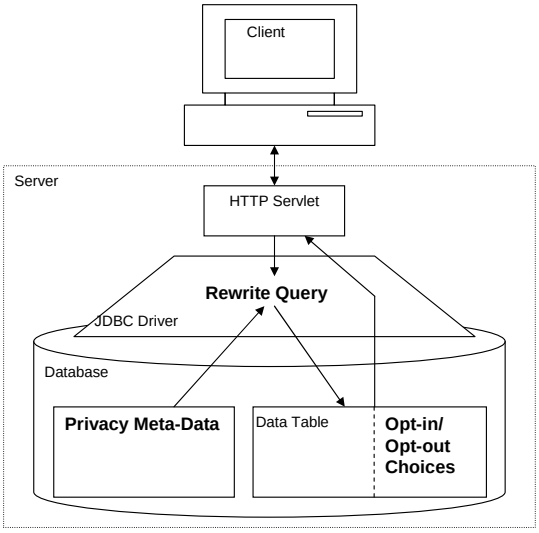

Figure 7: Implementation architecture overview

In addition to storing the data disclosure rules, we must provide a mechanism for storing user choices. In the basic architecture, we store these values in additional choice columns appended to the data tables themselves.

The basic enforcement mechanism intercepts and rewrites incoming queries to incorporate the privacy semantics stored in the privacy meta-data tables, as well as the user choices. The mechanism uses case-statements to resolve choices and conditions, and applies additional predicates to filter prohibited records from the result set. The query rewrite algorithm is a straightforward SQL implementation of the enforcement definition.

Consider, for example, a data table *Patients*, containing an attribute *Phone*. Under the privacy policy that is in place, the Phone attribute is included in the *Address* category, which is made available to charities for the purpose of solicitation on an opt-in basis. The user choices for Address information are stored in column Choice_1. The choices for the primary key of the patients table, ID, are stored in column Choice_2. Suppose the following query is issued for this recipient and purpose:

```
SELECT Phone FROM Patients
```

This query can be rewritten to resolve this particular condition as follows, using the table semantics model:

```
SELECT
CASE WHEN Choice_1 = 1 THEN Phone ELSE null END
FROM Patients AS q1(Phone)
WHERE Choice_2 = 1
```

Similar rewriting techniques resolve the privacy semantics of both allowed and prohibited categories. The rewriting algorithm is given in Figure 10, and the algorithm for resolving conditions is given in Figure 11. The *Resolve_Category()*, *Resolve_Policy()*, and *get_Condition()* functions mentioned

Purpose	Recipient	PID	Category	Choice Table	Choice Col.	Choice Opr.	Choice Val.
Insurance	Billing Office	P1	Address	-	-	-	-
Solicitation	External Charity	P1	Address	Patients	Choice_1	=	1

Figure 8: Sample policy table from the privacy meta-data, showing two sample rules.

PID	Table_Name	Column_Name	Category
P1	Patients	Name	Personal
P1	Patients	Age	Personal
P1	Patients	Address	Address
P1	Patients	Phone	Address

Figure 9: Sample data categories table from the privacy meta-data showing the mappings of data columns to the data categories used by the policies.

```
Rewrite (Query Q, PolicyID PID, Purpose P, Recipient R)
{
    for each table T referenced by Q

      nested_Select = "(SELECT "
      for each column c ∈ T
        cat = Resolve_Category (T, c)
        pSemantics = Resolve_Policy (PID, P, R, T, cat)
        if (pSemantics == FORBID)
          nested_select += Rewrite_Null(T)
        else if (pSemantics == CONDITION)
          cond = get_condition (PID, P, R, T, cat)
          nested_select += Rewrite_Cond (T, c, cond)
        else //(pSemantics == ALLOW)
          nested_select += c

    nested_select += " FROM T) AS q1("
    for each column c ∈ T
      nested_select += c + ", "
    nested_select += ") )"
    nested_select += filterRows(Q, T, PID, P, R)

    //Replace reference to T in query Q with
    //new Select statement
    Replace(Q, T, nested_select)
}
```

Figure 10: Basic algorithm for rewriting queries for privacy enforcement

in the algorithms are implemented as simple queries to the privacy meta-data tables. When the policy store table contains no rule corresponding to a particular *purpose* and *recipient*, the *Resolve_Policy()* function evaluates to FORBID. If the policy table contains an appropriate rule, but the values of the condition columns are null, then *Resolve_Policy()* evaluates to ALLOW. Otherwise, it evaluates to CONDITION. The *FilterRows()* function removes prohibited rows from the result set, as indicated by either the table semantics (Figure 12) or query semantics (Figure 13) model.

3.2 Privacy Enforcement using Views

Based on the previous section, an alternative architecture becomes apparent in the case of table-semantics enforcement. In this case, it is possible to achieve the same enforcement using views, while circumventing the overhead of rewriting incoming queries. This simplifies the architecture greatly by capturing all of the information from the meta-data tables described in the previous architecture in a single table mapping *(purpose, recipient)* pairs to privacy views of each table, as shown in Figure 14. These views can be defined using the same case-statement mechanism we described above, and at most we need to define one view for each *(purpose, recipient, policy)* combination.

```
Rewrite_null(Table T) {
    return "null"
}
Rewrite_Cond(Table T, column c, Condition cond) {
    //Resolve Conditions stored as
    //Columns in the data table
    return "CASE
    WHEN " + cond.cond_column +
    cond.cond_opr + cond.cond_value +
    "THEN c
    ELSE null
    END"
}
```

Figure 11: Case statements for resolving privacy semantics of data attributes, including choices stored as columns within the data table

These views may be constructed once at policy installation time, in which case we no longer need to store the privacy policy table or the category table. Alternatively, we may continue to store this information and lazily construct and cache these views as each is requested. In either case, we intercept incoming queries, and based on the purpose and recipient information, redirect them to the appropriate view.

There is a complication to this approach when we consider application queries with predicates over indexed data columns. Consider for example the following query over a data table in which SSN is an indexed data value, and the disclosure of SSN is governed by some choice stored in Choice_2. Name is a non-indexed data value, and disclosure of Name is governed by Choice_1. For simplicity, we ignore primary-key based filtering in this example:

```
SELECT SSN, Name
FROM Participants
WHERE SSN = 222-22-2222
```

In this case, the query is translated to:

```
SELECT SSN, Name
FROM (SELECT CASE WHEN CHOICE_2 = 1 THEN SSN ELSE null END,
    CASE WHEN CHOICE_1 = 1 THEN Name ELSE null END
    FROM Participants) AS q1(SSN, Name)
WHERE q1.SSN = 222-22-2222
```

Unfortunately, executing this query in DB2 causes us to discard the index on SSN because the reference to SSN is buried inside a case-statement. To fix this problem, we can pull the indexed data attribute and the corresponding choice out to the predicate, where the index can more easily be applied [3]:

[3]Thank you to <removed> for pointing out this fix.

```
FilterRows (Query Q, Table T, PolicyID PID, Purp P, Recip R)
{
    pKey_Cats = unique categories in primary key

    f = ""
    if (∃k ∈ pKey_Cats such that
          Resolve_Policy(PID, P, R, T, k) == FORBID)
       //Attach an unsatisfiable predicate
       f += "WHERE false"
    else
       for each k ∈ pkey_Cats such that
             Resolve_Policy(PID, P, R, T, k) == CONDITION
          cond = get_condition(PID,P,R,c)
          f += "WHERE " + cond.cond_column +
             cond.cond_opr + cond.cond_value
          if more categories in pKey_Cats
             f += "AND"

    return f
}
```

Figure 12: Algorithm for filtering prohibited records using the table semantics model of enforcement

```
FilterRows (Query Q, Table T, PolicyID PID, Purp P, Recip R)
{
    data_Cats = unique categories projected by Q

    f = ""
    if (∃c ∈ data_Cats such that
          Resolve_Policy(PID, P, R, T, c) == ALLOWED)
       //no filtering
    else if ¬∃c ∈ data_Cats such that
          Resolve_Policy(PID, P, R, T, c) == CONDITION)
       f += "WHERE 0 = 1"
    else
       for each c ∈ data_Cats such that
             Resolve_Policy(PID, P, R, T, c) == CONDITION
          cond = get_condition(PID, P, R, c)
          f += "WHERE " + cond.cond_column +
             cond.cond_opr + cond.cond_value
          if more categories in data_Cats
             f += "OR"

    return f
}
```

Figure 13: Algorithm for Filtering prohibited records using the query semantics model of enforcement.

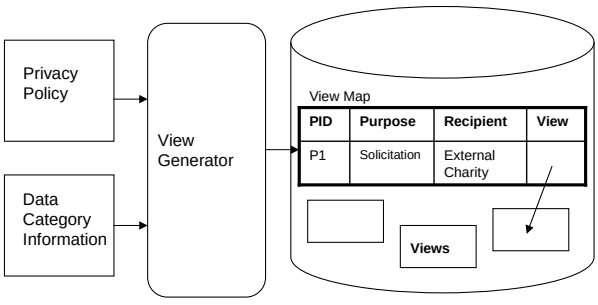

Figure 14: Alternative architecture maps *(purpose, recipient)* pairs to views of each table.

```
SELECT SSN, Name
FROM (SELECT SSN,
    CASE WHEN CHOICE_1 = 1 THEN Name ELSE null END,
    Choice_2
    FROM Participants) AS q1(SSN, Name, Choice_2)
WHERE q1.SSN = 222-22-2222 AND q1.Choice_2 = 1
```

As this optimization is based on the query itself, it cannot be incorporated into the view definition. We may only pull the choice out to the predicate when the query includes a predicate on the particular attribute.

3.3 Alternative Rewrite Algorithm

An alternative to the case-statement rewrite mechanism implements the Table Semantics and Query Semantics enforcement models using the left outer join and full outer join operators respectively.

Consider the same query we translated using the case-statement algorithm in Section 3.1, with privacy semantics as described previously:

```
SELECT Phone FROM Patients
```

This query can be rewritten as follows to reflect the table semantics enforcement model:

```
(SELECT
ID WHERE Choice_2 = 1) AS t1(ID)
LEFT OUTER JOIN
    (SELECT
    ID, Phone WHERE Choice_1 = 1
    FROM Patients AS q1(Phone)
    WHERE Choice_2 = 1) AS t2(ID, Phone)
ON t1.ID = t2.ID
```

The translation algorithm for table semantics is a SQL implementation of the following relational algebra expression; we omit the full SQL algorithm in the interest of space. Consider some query Q; each table T referenced by Q contains some attributes, $a_1 \ldots a_n$. For simplicity, we assume these attributes belong to separate categories. 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 each category contains just one column, and 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:

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

It is worth noting that in DB2 the outer join rewrite algorithm cannot be applied to queries of the form `"SELECT FOR UPDATE"` because of the join operators involved. This is similar to the fact that, in general, views joining multiple tables are not updatable. However, in this case, there is a straightforward translation from the view update to a table update, so in the future the database system could be extended to handle this case.

The SeaView system took a similar approach in constructing cell-level access control [16]. In the SeaView system, multilevel relations existed only at the logical level, as views of the data. They were actually decomposed into a collection of single-level tables, which were physically stored in the database. The multi-level relations were recovered from the underlying relations using the left outer join and union operators. However, there are important performance implications in choosing to use an outer join rewrite algorithm for limited disclosure, as we discuss later in the paper.

3.4 Alternative Choice Storage

The basic architecture takes a simple approach to storing user choice values by appending additional columns to the data table T. We refer to this as the *internal storage* design. If the number of choices increases as the application evolves over time, modifying the application schema may be costly in this approach. In addition, a space overhead is paid per application record.

We now consider alternative approaches for storing choices *externally*[4], in tables other than data tables. To some extent, these alternatives represent a tradeoff between query efficiency and ease of deployment [5].

The *multiple external 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 tuple of T for which the donor of the data opted in for the ith choice. Thus, if the data table T is extended with n choices to yield the internal design T', table $C_i = \pi_{key}(\sigma_{choice_i=1}(T'))$ for choice i.

The choice tables involved in V_C correspond to choices for columns mentioned in the query, partitioned into *key* and *non-key* choices. Key (resp. non-key) choices are those that involve key (resp. non-key) attributes in T. A tuple of T is not visible unless the donor has opted in for all key choices; therefore, the corresponding choice tables are combined using joins in the view V_C. A non-key choice determines whether the corresponding field is visible (assuming that the tuple 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. The case statements in the query modification algorithm test whether the choice field (generated from the outer join of the corresponding external table) is 1 or *null* to determine whether the contents of a field are visible.

The *single external table* design replaces the multiple external tables with a single table C_C. The schema contains two fields: the key for table T and a choice field whose values lie in the range $1...n$ where n is the number of choices. If the donor of a record with key k has opted into the ith choice, the record $< k, i >$ is in C_C. The basic query modification algorithm is similar to that in the external multiple table design; the main difference is that selections on the second field of C_C are used to generate subsets of C_C that correspond to the tables C_i in the multiple table design.

The external choice store design is particularly attractive in two instances. First, deployment requires little or no modification to existing data tables. Second, it offers more flexibility for handling conditions more complex than simple choices. However, the clear downside is the cost incurred by performing additional joins for each query. When comparing multiple versus single external designs, multiple consumes less space but at the expense of using the catalog significantly more.

While the discussion of storage alternatives has been focused on external designs, we also considered internal design alternatives. For example, instead of storing a 0 to denote an opt-out, a *null* value could have been used. Additionally, the collection of choice fields can be abstracted as a set-valued attribute. The schema of the application table T can be appended with a set-valued attribute S_C whose values are drawn from the domain of choices. If a record r in T has a choice C_i opted-in, $r.S_C$ must contain C_i.

The enforcement condition can then be expressed in terms of the *containment* operator. Given a record r and a set-value S, the containment operator returns r if every member $s \in S$ also satisfies $s \in r.S_C$. Assuming the `CASE` statement can be extended to test for containment in $r.S_C$, the rewrite algorithm could utilize such a set-valued approach. With regards to storage, external and internal alternatives are applicable to set-values as well. Furthermore, for internal alternatives, representations such as bit-maps and techniques such as compression can be leveraged.

4. PERFORMANCE EVALUATION

We performed extensive experiments to study the performance of our architecture and of query modification as a method of enforcing limited disclosure. 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. In the worst case, we incur the cost of checking privacy semantics, but we do not gain any performance by filtering prohibited tuples from the result set.

- **Scalability** We test the scalability of our rewrite algorithm 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*)[6], and the percentage of the records selected by an issued query (*application selectivity*).

[4]The delineation between internal and external storage alternatives is based on a similar classification in [24] used in the context of database support for set-valued attributes.

[5]In this section, we ignore any potential incompatibility with the `FOR UPDATE` command mentioned previously.

[6]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. In the table semantics model, this assumption does not influence execution time. If the primary key is allowed, then we fetch the tuple and process a case statement for each cell. For the query semantics model, the number of independent "optable" columns only influences performance insofar as it influences the number of tuples retrieved, so it is possible to assess the performance of "multi-catagory" tables using a single category evaluation. The number of independent data categories in a table *does* influence the performance of the outer join algorithm, as it dictates the number of joins necessary. We discuss this issue in Section 4.2.4.

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

Figure 15: Benchmark dataset and choice values are stored in the same table.

- **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 impact of this filtering on performance.
- **Enforcement Model** We study the performance implications of choosing the Table Semantics or Query Semantics enforcement model.
- **Rewrite Algorithms: Case vs. Outer Join** We briefly compare the performance of the case-statement and the outer join rewrite algorithms.
- **Views vs. Complete Query Rewrite** We discuss the tradeoff between defining and caching privacy views and performing complete query rewrite for table semantics enforcement. We measure the cost of completely rewriting queries in our Java prototype implementation. We also discuss the implications of materializing the privacy-preserving view.
- **Choice Storage** We discuss the implications of choosing among the various modes of choice storage.

There are several distinct sources of performance cost in our architecture, which we isolated in our performance experiments.

- **Query Rewrite** Our implementation intercepts and rewrites queries. This component includes indexed lookup queries to the privacy meta-data. The cost of rewriting a query is constant in the number of columns and categories in the underlying table schema, and relatively small compared to the cost of executing the queries themselves.
- **Query Execution** The cost of executing the rewritten query includes some amount of I/O, CPU processing, and the cost of returning the resulting data to the application.

4.1 Experimental Setup

We evaluate the performance of our architecture and enforcement algorithms using a synthetically-generated dataset, based on the Wisconsin Benchmark[17]. The synthetic data schema is described in Figure 15. All experiments were run on a single 750 MHz processor Intel Pentium machine with 1 GB of physical memory, using DB2 UDB 8.1 and Windows XP Professional 2002. The buffer pool size was set to 50MB, and the pre-fetch size was set to 64KB. All other DB2 default settings were used, and the query rewrite algorithms were implemented in Java.

To measure the cost of rewriting queries, we used the system clock. To measure the cost of executing queries, we used the DB2batch utility. Each query was run 6 times, flushing the buffer pool, query cache, and system memory between unique queries. The results given below represent the warm performance numbers, the average of the last 5 runs of each query. The size of the data table is 5 million records, except where otherwise noted.

4.2 Experimental Results and Analysis

4.2.1 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 selection queries, with predicates applied to non-indexed data columns. We report the results for our table semantics privacy enforcement model, but the trends are similar for query semantics. We assume, as described previously, that all columns in the table belong to a single data category, with a single choice value. To measure the overhead cost of enforcement, we consider the worst case scenario as described above, where the choice selectivity is 100%, so we incur all the cost of privacy processing, but do not see the performance gains of filtering.

Figure 16 shows the overhead cost of executing queries rewritten for privacy enforcement over tables containing 1 million and 10 million records. The graphs show the total execution time for queries with various application selectivity levels, and of the same queries rewritten using the case-statement rewrite algorithm. In all of these examples, the query plan is a sequential scan. The rewritten queries show the overhead of processing the additional case statement for each cell. Figure 17 shows the CPU time used in executing these same queries, in particular the extra cost of processing the additional case statements.

Because the figures show the warm performance numbers, the results of queries over the 1 million-tuple table can largely be processed from the buffer pool. In the case of the 10 million-tuple table, however, the size of the table exceeds the size of the buffer pool and the query processing incurs disk I/O. 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. As the application filters fewer tuples, the CPU cost increases, but because the queries are executed as sequential scans, the I/O cost does not change, explaining Figures 16 and 17. The total cost increases when we increase the table size from 1 million to 10 million records, but this cost is dominated by the I/O.

4.2.2 Implications of Filtering due to Choice Selectivity

In cases with choice selectivity less than 100%, the rewritten queries perform significantly better because, through the use of a choice index, they need to read fewer tuples. In this experiment, the application query selects all 5 million records in the table. However, the rewritten queries vary the

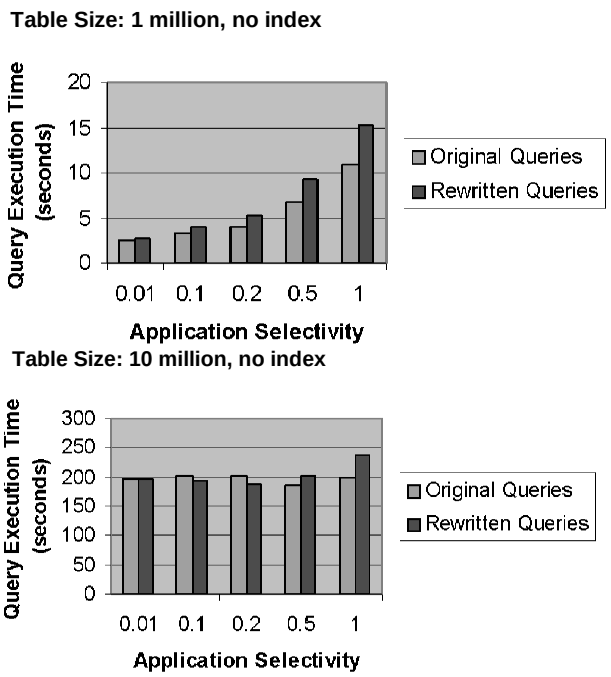

Figure 16: Total performance overhead of table semantics enforcement using case-statement rewrite with choice selectivity = 100%

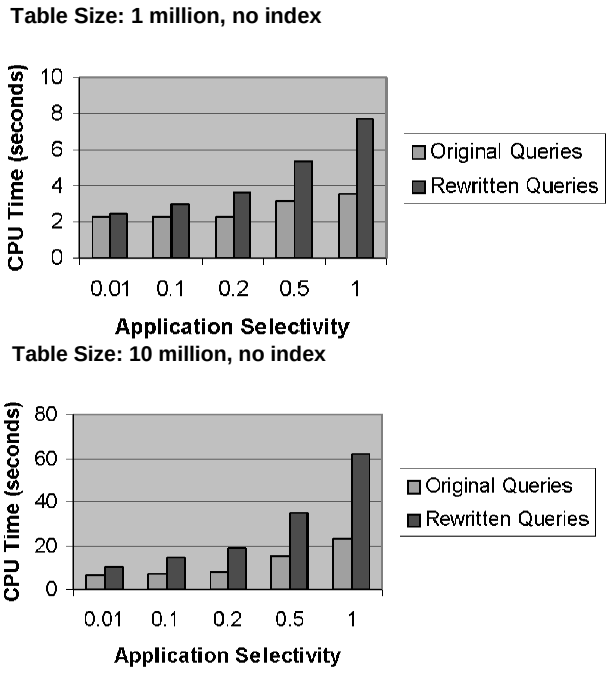

Figure 17: CPU overhead of table semantics enforcement using case-statement rewrite with choice selectivity = 100%

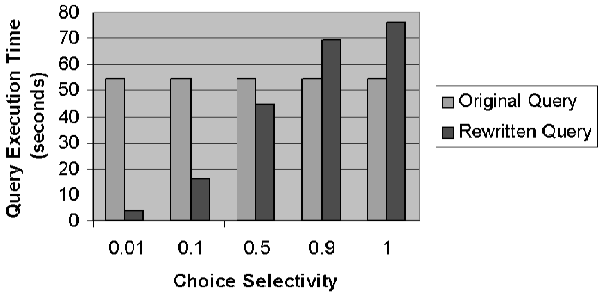

Figure 18: Comparing the cost of executing rewritten and original queries for varying choice selectivity; Application selectivity = 100%

choice selectivity. Note that in our experiment, the queries with a choice selectivity of .01, .1, and .5 used the index on the choice column; the others did not.

As can be seen from Figure 18, the performance gain is considerable for low choice selectivity. When the choice selectivity is near 100%, we incur the cost of privacy checking, but do no benefit from choice selectivity. Still, the cost of enforcement is quite low.

4.2.3 *Performance Differences Among Enforcement Models*

There is a clear performance distinction between the table semantics and the query semantics privacy models, which becomes clear when we consider a table comprised of columns belonging to different data categories, with independent privacy rules.

In the table semantics model, a tuple is filtered from the result set if the primary key is forbidden. In this case, if the underlying table schema is defined as suggested in Section 2.3, and a record is made visible if any of its attributes are visible, then it is convenient to think of the independent choice selectivities for all of the projected columns combining to form the *effective choice selectivity*. If we consider some table, T, containing x categories, such that the choice selectivities for the categories are independent of one another, the effective selectivity can be determined by $1 - \prod_{i=1}^{x}(1 - s_i)$, where s_i is the choice selectivity corresponding to category i.

This is not the case when we consider the query semantics model. Here, the effective choice selectivity is not determined by the underlying table schema; instead it is determined by the selectivities of only those columns projected by the query. In many situations, this leads to substantial performance gain, as we need to read and return fewer tuples.

However, in some situations, this performance gain may be offset because the query semantics rewrite algorithm yields a query that is less likely to use indices on the choice columns. For instance, if our query projects two columns belonging to two separate categories, in the query semantics model, the filtering predicate might include a disjunction of the form, `WHERE Choice_0 = 1 OR Choice_1 = 1`. We observed that when executing the above predicate, the optimizer does not make use of the indices on either `Choice_0` or `Choice_1` even though the combined selectivity of the two choices is low. Our conjecture is that the choice indexes were not incorporated in the query plan because of the disjunction in the

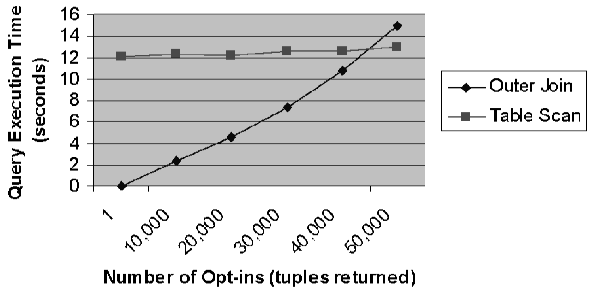

Figure 19: Comparing case statement executed as a sequential scan and outer join rewrite algorithms for indexed choice values.

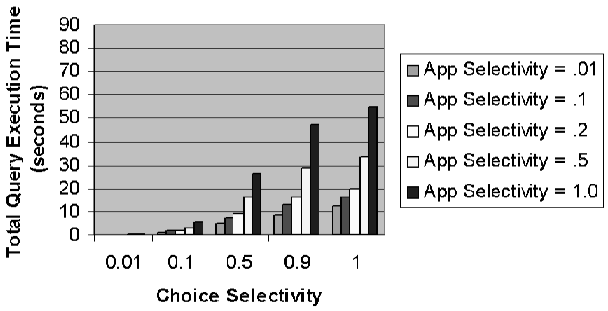

Figure 20: Performance of queries executed over a privacy-preserving materialized view.

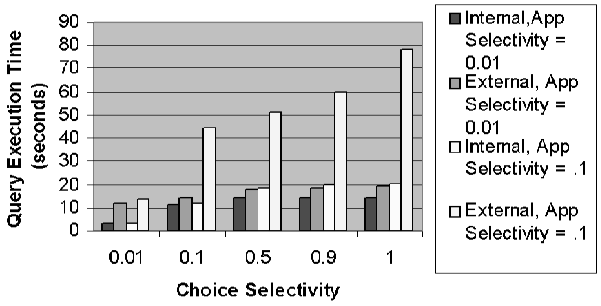

Figure 21: Performance of external, multiple table storage method for two application predicate selectivities as compared to the internal storage method.

predicate.

4.2.4 Comparing Rewrite Algorithms

In most situations, our case-statement rewrite algorithm substantially outperforms the outer-join rewrite algorithm, and for good reason. The outer join algorithm scales poorly because of the repeated and costly join operations involved. For large tables with high choice selectivity (many tuples selected), the performance was quite poor, so we have omitted those results from the paper.

However, there are some specific situations where the outer join algorithm does perform better than using case-statements. For example, we observed in the previous section that the DB2 optimizer did not use choice indexes for a query with a predicate including a disjunction of conditions. However, the outer join rewriting algorithm was more likely to be able to use such indexes.

Figure 19 compares the performance of the outer join rewritten query with a case-statement rewritten query performing a sequential scan. These are the results for a query consisting of two categories and performing query semantics enforcement, so the outer join query includes one join. A complete characterization of conditions under which the outer join rewrite algorithm should be selected over the case-statement algorithm is the subject of future work.

4.2.5 Query Rewriting vs. Views

We showed that it is possible to implement a table semantics enforcement mechanism by redirecting incoming queries to predefined privacy views, rather than entirely rewriting the incoming queries. In practice, these two methods yield identical query execution performance, except when we have to perform additional rewriting to avoid discarding a useful index, as explained in Section 3.2. In this case, the performance impacts of not using an index may be substantial.

The views implementation avoids much of the cost of rewriting queries to reflect the privacy semantics. However, this cost is constant in the number of columns, and for large tables and complex queries, small compared to the cost of executing the queries themselves. The cost of querying the privacy meta-data is negligible because these queries are implemented as simple indexed lookups. For eight columns, from distinct data categories, the average time to rewrite a query in our Java implementation averaged approximately 0.15 seconds when we pooled the privacy meta-data connections.

An alternative, feasible only as a method of optimizing performance for a few *(purpose, recipient)* pairs, is actually materializing the view. Querying the materialized view is very inexpensive, as shown in Figure 20, though we must take into account the effort needed to maintain the view as the underlying data tables are updated. For each data table, this solution requires us to store one table, which could be as large as the original data table, per *(purpose, recipient)* pair.

4.2.6 Comparing Choice Storage Methods

The following experiments address some of the tradeoffs involved in deciding which storage design to use. We focus on two factors that influence performance: the added join processing cost of external versus the extra bandwidth required by internal.

The first experiment demonstrates the overhead associated with external methods for a variety of choice and application selectivities. The 5-million record relation with schema shown in Figure 15 is decomposed into both external single and multiple table alternatives. Figure 21 shows how the performance of external single degrades with respect to the internal storage alternative as application and choice selectivity are increased. External single is not shown, as it degrades very quickly but it should be noted that its performance levels off and eventually beats multiple external for low selectivity application and choice predicates. In either case, the added flexibility of external methods would be difficult to justify at such low predicate selectivities.

The second experiment was designed to see if and when external could out-perform the internal method. For a variety of application and choice selectivities, the internal table's choices were increased to 50 and 100 columns. While the most selective choice predicate considered (1%) still favored

the internal design, the external multiple performed better at 10% choice selectivities, continuing the trend up to 50% selectivity.

5. CONCLUSION AND FUTURE WORK

Limited disclosure is a vital component of a data privacy management system. We presented several models for limited disclosure in a relational database. We then proposed a scalable architecture for enforcing limited disclosure rules at the database level. Application-level solutions are inefficient and unable to process arbitrary SQL queries efficiently and without leaking private information. By pushing the enforcement down to the database, we gain improved performance and query power, without modification of existing application code. We showed that the performance overhead of performing database-level privacy enforcement is small and scalable, and often times the overhead is more than offset by the performance gains obtained through tuple filtering.

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 linked 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. Also of interest is the problem of identity management. In our implementation, we infer the purpose and recipient based on the application issuing the query. However, there are a multitude of alternative ways of defining and obtaining this information. Lastly, we are looking at ways of efficiently maintaining an audit trail that would allow us to report all data disclosures for the purpose of guaranteeing regulatory compliance.

6. REFERENCES

[1] govt.oracle.com/ tkyte/article2/index.html.

[2] www.vignette.com.

[3] Informix-OnLine/Secure security features guide. Technical report, Informix Software Inc., Menlo Park, California, April 1993.

[4] Final evaluation report: Sybase SQL Server Version 11.0.6 and Secure SQL Server Version 11.0.6. Report CSC-EPL-96/002 and CSC-EPL-96/003, National Computer Security Center, March 1997.

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

[6] eXtensible access control markup language (XACML) version 1.0 specification, Feb. 2003. OASIS Standard.

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

[8] Privacy conscious user profile data management with GUPster. Tech. report, Bell Laboratoris, Lucent Technologies, 2003.

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

[10] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proc. of the 28th Int. Conf. on Very Large Data Bases*, Hong Kong, China, August 2002.

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

[12] R. Ashley, S. Hada, G. Karjoh, C. Powers, and M. Schunter. Enterprise privacy authorization language 1.1 (EPAL 1.1) specification. IBM Research Report, June 2003.

[13] 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.

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

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

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

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

[18] V. Doshi, W. Herndon, S. Jajodia, and C. McCollum. Benchmarking multilevel secure database systems using the MITRE benchmark. In *10th Annual Computer Security Applications Conf.*, Dec. 1994.

[19] S. Jajodia and R. Sandhu. Polyinstatiation integrity in multilevel relations. In *IEEE Computer Society Symp. on Research in Security and Privacy*, May 1990.

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

[21] N. Kabra, R. Ramakrishan, and V. Ercegovac. The QUIQ Engine: A hybrid IR-DB system. In *In Proc. Int. Conf. on Data Engineering*, Bangalore, India, March 2003.

[22] 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.

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

[24] K. Ramasamy. Efficient storage and query processing of set-valued attributes. PhD Dissertation, Univ. of Wisconsin Computer Sciences Department, July 2001.

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

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

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