Records Retention in Relational Database Systems

Ahmed Ataullah Ashraf Aboulnaga Frank Wm. Tompa

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{aataulla, ashraf, fwtompa}@cs.uwaterloo.ca

ABSTRACT

The recent introduction of several pieces of legislation mandating minimum and maximum retention periods for corporate records has prompted the Enterprise Content Management (ECM) community to develop various records retention solutions. Records retention is a significant subfield of records management, and legal records retention requirements apply over corporate records regardless of their shape or form. Unfortunately, the scope of existing solutions has been largely limited to proper identification, classification and retention of documents, and not of data more generally.

In this paper we address the problem of managed records retention in the context of relational database systems. The problem is significantly more challenging than it is for documents for several reasons. Foremost, there is no clear definition of what constitutes a business record in relational databases; it could be an entire table, a tuple, part of a tuple, or parts of several tuples from multiple tables. There are also no standardized mechanisms for purging, anonymizing and protecting relational records. Functional dependencies, user defined constraints, and side effects caused by triggers make it even harder to guarantee that any given record will actually be protected when it needs to be protected or expunged when the necessary conditions are met. Most importantly, relational tuples may be organized such that one piece of data may be part of various legal records and subject to several (possibly conflicting) retention policies.

We address the above problems and present a complete solution for designing, managing, and enforcing records retention policies in relational database systems. We experimentally demonstrate that the proposed framework can guarantee compliance with a broad range of retention policies on an off-the-shelf system without incurring a significant performance overhead for policy monitoring and enforcement.

Categories and Subject Descriptors

H.2.7 [**Database Administration**]: Security, integrity, and protection; K.5.2 [**Legal Aspects of Computing**]: Governmental Issues - Regulations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'08, October 26–30, 2008, Napa Valley, California, USA.
Copyright 2008 ACM 978-1-59593-991-3/08/10 ...$5.00.

General Terms

Legal Aspects, Management

Keywords

Privacy, records retention, legal compliance, business records, relational systems

1. INTRODUCTION

Storage, management, and efficient access to data have traditionally been of prime importance for businesses. Any loss of records has meant the loss of valuable business intelligence. However in the past decade, increasing awareness of privacy related issues and the introduction of several pieces of legislation mandating strict records retention periods have forced organizations to move away from the "store everything" paradigm. Customers demand that their personal information not be indefinitely retained by businesses, and government legislations require that certain types of records, such as those related to taxation, be preserved for specified periods of time. This puts database systems in an awkward situation of having to store business records in tuples and tables that are relationally intertwined (e.g., through foreign keys) but with a diverse set of policies applicable to them.

The problem is further exacerbated by the fact that there is no clear notion of a "business record" in a database system, which means that there is no meaningful way of protecting records and deleting them after they are no longer required.[1] For example, a typical physical document such as an invoice may be identified by a row in one table, but its contents in a database with a normalized schema may be spread across various tables. Various departments of the business may consider selected attributes of the invoice or its related line item tuples as sensitive information and may want to expunge or protect them as necessitated by law. With conflicting requirements arising through the intertwining of complex records, the problem of mediating these requirements and determining whether they can be enforced becomes a significant challenge. As more and more policies are integrated from different departments, the problem of maintaining a consistent and enforceable set of data retention rules can easily become unmanageable.

Not complying with legally mandated records retention policies can lead to criminal charges against individuals or corporations, and this greatly increases the gravity of the problem. Even accidentally destroying records and being

[1]It is important to note that in this paper we distinguish between "records," which can be thought of as "business documents," and "tuples" that are stored in relations.

unable to reproduce them in civil litigation scenarios can lead to the imposition of heavy fines [14]. At the same time retaining certain records for too long can be a source of liability or a violation of published records retention policy. Since records management solutions for relational databases do not exist, organizations that choose to enforce retention policies on relational data are forced to do so in an ad-hoc manner. Legislation is interpreted by a privacy officer who is aware of the database schema, and then data deletion specifications are handed down to a programmer. Deleting records when they have expired is most likely done through the use of scheduled SQL execution tasks. To implement protective data retention measures, for example those legislated by the Sarbanes-Oxley Act [1] mandating corporate financial records to be maintained for specified periods, an elaborate set of logging (or replicating) triggers is the only workable solution for database administrators.

Such an approach to hand-coding a corporate retention policy in a database system has several shortcomings. First, it is only viable if there is a single corporate privacy officer providing the interpretation of data retention rules. In scenarios involving large, complex and possibly federated schemas, it is unlikely that one administrator will be responsible for understanding and interpreting legal records retention requirements for all functional areas of the organization. Second, administrators are not able to verify the effect of their policies automatically. For example, batched execution of data purging queries may attempt to delete records that are protected by data retention triggers. Policy conflicts are nearly impossible to detect beforehand when dealing with arbitrarily written queries. Consequently, ad-hoc conflict resolution becomes inevitable. The task of managing and maintaining a large number of scheduled tasks and retention triggers is not only challenging, but can also represent a significant overhead cost in business situations with evolving schemas.

In this paper we present a complete end-to-end solution for managed data retention in relational database systems. Our key contribution is that of answering three fundamental questions pertaining to systematic records retention in database systems:

- What is a business record in the context of a records retention policy in a relational database system?
- What does it mean to delete (or protect) a particular record?
- How can privacy officers and database administrators be sure that policy actions described on what they deem to be records will be meaningful and correct?

Our proposed framework answers the above questions, and it builds an expressive and flexible solution for records retention that can be implemented on top of existing mainstream database management systems.

2. RECORDS IN DATABASES

Information in a database that should be protected or deleted under a policy must be identified by subject experts in formats that are meaningful to them. Since each policy maker can interpret individual pieces of data differently, we propose a *view based* records management system.

DEFINITION 1. *A **relational record** is a logical view over a fixed relational schema.*

Note that this definition is suitably expressive, as it gives policy makers the ability to define a record for policy enforcement at any level of granularity. Although this definition obscures the distinction between a single record and a collection of records, we believe that this is the most suitable method of identifying valuable business information that caters to the needs of all users. Such a framework allows each user to specify his or her own interpretation of a given record and the valuable information contained therein. Once important records are identified by users, retention policies can then be systematically implemented on the relevant view definitions.

To delimit the scope of the obligations on the data stored by an organization in a relational database, we propose that all records be explicitly declared. For example, if a business has very specific retention obligations on the taxation information contained in paid invoices issued to Jones Corporation, it can declare a record as follows:

```
DEFINE RECORD R1 AS
SELECT *
FROM Invoice NATURAL JOIN LineItem
WHERE Recipient = 'Jones Corp.'
AND Paid = true
```

This example of a record declaration captures the data contained in all paid invoices issued to Jones Corp. Observe that relational records are much more flexible than traditional physical records (such as a single invoice or telephone bill) and can encapsulate a collection of related traditional physical records in a very compact definition. Furthermore, using views provides a declarative means for policy makers to specify policy-relevant pieces of data that may be spread across numerous physical records.

An important requirement for any records retention framework is to accommodate the notion of temporality. The vast majority of data retention policies are time driven and warrant that records be destroyed after (or protected for) a fixed time after an event. We therefore propose the use of a temporal function **NOW**, which will denote the current system-wide time. Such a function is presently implemented in all modern database systems, and it is simply treated as a special timestamp. Support for comparative operators, date-time functions, and even its use in view definitions is also widespread. An example of a time-varying record would be to add a temporal predicate to the Jones Corporation record, R_1, to identify those invoices and line items that were paid within the last five years:

```
DEFINE RECORD R2 AS
SELECT *
FROM Invoice NATURAL JOIN LineItem
WHERE Recipient = 'Jones Corp.'
AND Paid = true
AND Years(NOW - PaidDate) < 5
```

It must be clarified that our model for records and policies only considers the current state of the database. Here we do not deal with enforcing retention constraints on backup (offline) copies of the database or on tuples in older snapshots. Our aim is to ensure that the active database is always in a compliant state with respect to retention: all records that warrant protection are not affected by user initiated transactions and no records physically exist that have outlived their maximum retention period.

3. RETAINING RECORDS

In the context of business documents such as invoices on which tax has been collected, preservation typically implies that the invoices are not destroyed, not modified, and that no new details are added. The natural equivalents of these protective requirements in databases are to ensure that certain insertions, deletions, and updates are rejected. We propose the use of a *retention condition* to express these protective requirements in user defined records.

DEFINITION 2. *A **retention condition** is a boolean formula on the attributes of a record to denote tuples in the record on which a retention policy must be enforced.*

If the policy applies to all tuples in the record, the retention condition is simply the logical value *true*. We denote policies that prevent modification of records as *protective retention policies*. For these policies, we propose two levels of protection, namely *update* and *append*. A protective retention policy on a record is defined as a pair Ret_p:(*protective retention condition*, *level of protection*). For a record under update protection, a DML statement is aborted if it modifies (deletes or updates) tuples in the record for which the protective retention condition is true. Similarly, for a record under append protection, a statement is aborted if it inserts tuples in the record for which the protective retention condition is true.

A protective policy on all invoices issued to Jones Corporation and paid within the last five years (R_2) could be Ret_{p_1}:(*ItemTotalTax* > *0*, *update*) and a simple syntax for defining such a policy is as follows:

```
DEFINE PROTECTIVE POLICY
ProtectItemTotalTax ON R2 AS
(ItemTotalTax > 0, update)
```

The effect of the above policy would be that any update leading to a change in a taxable amount for a tuple in the record will be rejected if the initial taxable amount was greater than zero.

Update protection by itself is not adequate for avoiding phantom inserts. For example an employee could create new line items for older invoices with non-zero tax amounts. Append protection ensures that such inserts cannot happen, and the record cannot increase in size because of user initiated transactions. A policy such as (*ItemTotalTax* <> *0*, *append*), which can be instantiated using a similar syntax as described above, will reject all DML statements that lead to a new tuple being inserted in the record for which the taxable amount of the line item is non-zero.

Several retention policies can be enacted on a single record definition. For example, independent update and append policies can be placed at the same time to make sure that no change to a record can take place.

Note that changing the record definition slightly can have significant consequences in terms of these protection levels. For example consider the following record:

```
DEFINE RECORD R3 AS
SELECT Sum(ItemTotalTax)
FROM Invoice NATURAL JOIN LineItem
WHERE Recipient = 'Jones Corp.'
AND Paid = true
AND Years(NOW - PaidDate) < 5
```

R_3 adopts a different, summarized view of the record that was examined in R_2 and focuses on the total tax collected on behalf of Jones Corporation in the last five years. It is also an example of a record that may not correspond to a physical document, but on which retention policies can be enforced. The intent of a policy protecting R_3 could be to ensure that the total tax owed (paid in the past on behalf of Jones Corporation) is never changed in the database. An important difference in protecting R_3 is that append protection for such a record will have no effect. By definition, R_3 will always contain a single tuple, therefore the cardinality of this view can never increase.

It is important to recognize that because of the flexibility in record definitions, many different records can be created over the same data, each with its own retention objectives. In fact, a significant benefit of our approach of using views to define records is that policy makers can leverage existing definitions of documents derived from a relational database. Queries used to generate day to day documents such as invoices and sales reports can be taken directly from applications and used as the view definitions for defining records that are used to enforce retention policies.

We note that protection for records is only meaningful (and enforceable) in the context of user initiated transactions. Consider the following example of a record and an append protection policy:

```
DEFINE RECORD R4 AS
SELECT *
FROM Invoice
WHERE Days(NOW - CreateDate) > 5

DEFINE PROTECTIVE POLICY
TemporalPolicy ON R4 AS
(true, append)
```

In this example, a user has defined a record as "all invoices created more than 5 days ago" and attempted to enforce append protection for the contents of the record. In such a situation, as time passes tuples representing new invoices will automatically slide into the view and become part of the record definition. Since we are unable to abort the passage of time, we cannot protect records against trivial temporal alterations to the view, and we consider this as acceptable behavior that does not violate the retention policy. Details on how such trivial temporal alterations can be detected [5] are omitted due to space considerations.

Our framework for protecting records resembles integrity constraints on arbitrarily defined views. Consequently we are able to benefit from the support for views provided by typical database systems and many of the theoretical results in the area of static query analysis.

Protecting records from modifications can be addressed using methods from efficient maintenance of materialized views. The notion of detecting whether a modification to a database will impact a particular view applies to both problems. Specifically, given a record definition and a policy, we identify precisely the view that needs to be monitored for relevant updates as follows:

DEFINITION 3. *For a given record definition R and a retention condition for a policy C(Ret), a **critical view** C_v is defined as $C_v = \sigma_{C(Ret)}R$*

Critical views specified by protective retention policies are denoted as *protective critical views*. A critical view is a

subset of a record that contains rows on which policy actions need to be enforced. Consequently, protecting the contents of critical views is reduced to identifying modifications to a base table that will change the contents of a protective critical view and rejecting such statements. In our running example, the record and policy pair R_2 and $ProtectItemTotalTax$ generate the critical view specified by C_{p_1} as follows:

C_{p_1} =
```
SELECT ItemTotalTax
FROM Invoice NATURAL JOIN LineItem
WHERE Recipient = 'Jones Corp.'
AND Paid = true
AND Years(NOW - PaidDate) < 5
AND ItemTotalTax > 0
```

Updates and mechanisms for monitoring the effect of updates, such as alerters, assertions and triggers, have long been studied in the database community. A wide array of techniques, such as static analysis of updates to determine relevance for a view [7] and incrementally maintaining and monitoring the impact of updates on views [8] can be adopted in our context. However, the efficiency of any selected approach depends on a large number of factors specific to real-world data retention obligations. We provide a discussion of these factors and highlight the key decision making criteria that can lead to an efficient retention policy monitoring system. Our tests (Section 6) were developed with these principles in mind, and they demonstrate that the correct choice of event detection mechanism is critical for a view based records retention system.

The most important design consideration leading to efficient detection of policy violations is how we constrain the record definitions themselves. With the flexibility of defining records using arbitrary views with the full expressiveness of SQL comes the risk of having to incur the worst case cost of full re-computation of the record at every update. Thus, to ensure that our policies are practical and can be efficiently enforced, we need to restrict the record definition language. Our examination of data retention laws has led us to conclude that policies are typically defined on what law-makers would consider practical business records, which typically map in our framework to sets of related but simple parameterized queries. We observed that in most business schemas these "practical" records (or the data contained therein) can be expressed as *conjunctive queries*. Consequently, we restrict ourselves in this paper to the analysis of records specified as conjunctive queries with support for aggregation.

When enforcing policies, we should consider overlap among them. For example, given an update that is relevant for critical view C_{v_1} and is irrelevant for critical view C_{v2} in all database instances, there is clearly no need to check for violations on C_{v_2} for the update in question. This problem of determining exclusive relevance has been addressed in the context of query disjointness, and Elkan gives an efficient decision procedure to detect disjointness of conjunctive queries [10]. Generally, if updates can easily be checked for relevance against a large number of views, then the task of capturing violations becomes significantly easier. The complexity of the schema and the density of policies over a given set of relations also play a critical role in determining policy overlap. If the majority of data retention policies are clustered over unrelated relations, then there will certainly be less overhead in determining the impact of a particular update on all protected records.

We can also benefit from various static optimizations of policy actions on records. Given a set of record (view) definitions and a set of protective retention policies, we propose to derive an optimized set of policies with the same protective characteristics. One such optimization is that of eliminating redundancies in protective requirements that can naturally arise because of the hierarchical nature of business records. For example, at an abstract level, the data contained within a monthly sales report will typically be a subset of the data contained in the relevant yearly sales report. Therefore, when dealing with a large number of record definitions and policies, we may be able to exploit such correlations and check several policy violations against similar records.

Another property of a wide array of business records is that of temporal stability: as a record becomes older it is less likely to be modified. In many transactional business applications, records can be expected to be in the "active state" for a fixed period of time after their creation and then gradually achieve temporal stability. It may be beneficial in these situations to identify active records in order to isolate the passive or protected records. Temporal stability can also be used in conjunction with the fact that business records are often referenced with monotonically increasing identifier values. For example, if Invoice #500 was the first invoice created on Jan 01, 2007, and Invoice #1000 was the last invoice created on Dec 31, 2007, then it is very likely that all invoices created in the year 2007 lie in that particular range of identifiers. In such cases maintaining a few pointers can substantially reduce the overhead involved in monitoring whether a particular update will affect the contents of a record. Mechanisms to infer such correlations automatically or the ability for users to specify them must be supported by the underlying database system for these optimizations to be useful (e.g., range partitioning in Oracle and DB2).

4. DELETING RECORDS

So far we have focused on protecting records as long as retention conditions are met. The flip side of records retention is to ensure that sensitive information is removed from a database as soon as it has outlived its purpose.

The two options available to achieve compliance with policies that dictate maximum retention periods for records are destruction and transformation. Deleting a sensitive business record altogether is not a strategy that is widely practiced, nor is it suitable in many situations. Instead, techniques such as partial deletion and anonymization are preferable, as they ensure that records retain their business value and yet pose no liability for the organization.

We denote policies that mandate removal (deletion or anonymization) of records as *destructive retention policies* and extend our view based records management framework to support such policies. In this section, we develop a formalization for deletion and anonymization of user defined records over a fixed relational schema.

We define a destructive retention policy on a record as a pair Ret_d:(*destructive retention condition*, $\alpha(R)$). A *destructive critical view* is defined similar to a protective critical view, as a conjunction of the destructive retention condition and the record definition. In contrast to their protective counterparts, however, destructive critical views contain a subset of the rows of the record that need to be expunged from the database. In a *retention compliant state* all destructive critical views are empty, reflecting that there

is no sensitive information in the database that needs to be deleted or anonymized. Monitoring destructive critical views is also very similar to monitoring protective critical views, except that we only pay attention to whether the view is empty or not. If any destructive critical view in a database is non-empty, then the database is in a *retention unsafe state*, and some action needs to be taken to remove the critical rows from the destructive critical views. *Destructive actions*, denoted by $\alpha(R)$, are associated with each destructive retention policy, and on detection of a non-empty destructive critical view, the execution of the specified actions is intended to make the critical view empty, taking the database into a retention compliant state.

One approach to moving a database into a retention compliant state through destructive actions is to associate a stored procedure with every destructive policy, and to execute the procedure when a non-empty destructive critical view is detected. This technique essentially offers unbounded flexibility in the actions that can be performed to purge outdated records, but verification and ensuring correctness of actions performed by arbitrarily written procedures over all database instances is undecidable. Consequently, we focus on a restricted class of destructive actions and thereby present a *provably correct* mechanism for policy enforcement.

4.1 Correctness of Destructive Actions

We now introduce the first formal requirement for any destructive action on critical views to be provably correct.

DEFINITION 4. *A destructive policy P is **weakly correct** if upon any tuple becoming part of the destructive critical view V_d, the destructive actions specified by P, denoted by $\alpha(R)$, will ensure that V_d will become empty.*

Weak correctness implies that the invocation of the destructive actions specified by the policy must remove critical tuples from the critical view. Weak correctness by itself does not provide any guarantees for termination of policy actions. This is because actions of a particular destructive policy may introduce tuples in critical views of other destructive policies. Consequently, the notion of non-invasive policies is presented so that we can reason about policy execution patterns.

DEFINITION 5. *A destructive policy P is **non-invasive** with respect to another policy P' if the destructive actions specified by P, denoted by $\alpha(R)$, cannot affect the critical view of P'. P is called invasive with respect to P' if $\alpha(R)$ has the potential to change the contents of the critical view of P'.*

This definition specializes the definition of *relevant updates* presented by Blakeley et al. in the context of materialized views [7]. If $\alpha(R)$ of P cannot impact the contents of the critical view of P', then $\alpha(R)$ is irrelevant to the critical view of P' and P is non-invasive with respect to P'.

An important observation is that the definition of invasive policies does not restrict P' to being a destructive retention policy, and the notion of invasive policies can be used to detect delete-protect conflicts (Section 4.4). The definition also encompasses side effects that could be caused through destructive actions, for example through foreign key constraints with cascading deletes.

Invasiveness among policies need not arise directly because of data shared among records. As an example, consider two relations X_1 and X_2, each with a single attribute called *id*, and two record definitions, R_1 which selects all *id's* in X_1 that are not in X_2, and R_2 which selects all *id's* in X_2 that are not in X_1. Note that, by definition, R_1 and R_2 are always disjoint ($R_1 \cap R_2 = \emptyset$) but the definitions of R_1 and R_2 are inseparable. Performing insertions or deletions on either of these records has a direct impact on the other. Consequently, we must derive sufficient conditions to guarantee consistency of execution among the destructive actions of several destructive retention policies. The following requirement ensures that a policy will not only expunge all tuples from its critical view but also not lead to the insertion of new tuples in the critical views of other policies.

DEFINITION 6. *A destructive policy P is **strongly correct** if it is weakly correct and either (i) P is non-invasive with respect to other destructive policies or (ii) the destructive actions specified by P can only delete tuples from the critical views of other destructive policies.*

Finally, using the notion of invasiveness and strong correctness, we can offer an algorithm to determine whether a given set of destructive policy actions can be correctly enforced. We rely on constructing a *policy interference graph*, which can also be used to visualize the impact of actions performed by each destructive policy on contents of critical views of other policies:

DEFINITION 7. *A **policy interference graph** is a directed graph constructed using a set of policies $P = \{P_1, P_2, ..., P_n\}$ as nodes and edges from P_i to P_j when P_i is invasive with respect to P_j.*

LEMMA 1. *A set of destructive policies is guaranteed to be terminating if all cycles in the relevant policy interference graph involve strongly correct policies.*

The above lemma specifies a sufficient condition for avoiding circular enforcement of destructive retention policies. This result is similar to the results from trigger termination and cyclic execution of rules in database systems [4]. If all interfering actions caused by policies only cause removal of records from destructive critical views, and the number of tuples in a database is finite, then indefinite execution of destructive actions is not possible.

4.2 Trading Flexibility for Decidability

It is important to note that the correctness criteria that we developed for our proposed view based records retention framework is general enough for all record definitions and all possible destructive actions that expunge tuples from critical views. Unfortunately, determining correctness for arbitrary record definitions and destructive actions is undecidable. Therefore, we restrict ourselves to simple destructive actions on updatable conjunctive views. We argue (again) that business records can usually be expressed as conjunctive queries, and the destructive actions required to expunge or anonymize these records can be relatively simple. Such actions include setting a particular attribute value to a default value or to null, or deleting a tuple altogether; we denote these as *simple destructive actions*.

Before presenting a decision procedure for correctness, we introduce the notion of an *extended retention condition* and

a *post-condition* of the actions performed by a destructive retention policy. An *extended retention condition* is the boolean formula that defines the destructive critical view, constructed as the conjunction of the predicates of the record and the destructive retention condition of the policy, as described earlier. The *post-condition* of a destructive action is a boolean formula on the tuples of the critical view that must hold on a critical tuple (i.e., a tuple in the critical view) after the action is successfully applied to this tuple. For updatable conjunctive views, if the action involves deleting the tuple from the critical view, the post-condition is trivially false, since the deleted tuple will not exist in the record. On the other hand, if the action involves updating the value of one or more attributes in the critical tuple, say setting a_i to v_i for $i = 1 \ldots n$, then the post-condition is $a_1 = v_1 \wedge a_2 = v_2 \wedge \ldots \wedge a_n = v_n$.

LEMMA 2. *A destructive retention policy with an extended retention condition R_p and a destructive action having post-condition $\alpha_p(R)$ is weakly correct if and only if $R_p \wedge \alpha_p(R)$ is unsatisfiable.*

The above lemma reduces the problem of verifying correctness of policies to that of satisfiability of conjunctive predicates, and it states that enforcement is guaranteed if and only if $R_p \wedge \alpha_p(R)$ is unsatisfiable. To illustrate the usability of this result, consider the following schema and a simple destructive retention policy that monitors completed sales orders that were placed more than a year ago:

Order(OID, Delivered)
Transaction(Txn, OID, TxnDate, CreditCard)

where singly underlined attributes are keys and doubly underlined attributes are foreign keys.

Assume that the administrator has defined a record and a destructive policy as follows:

```
DEFINE RECORD OldTxns AS
SELECT OID, Txn, TxnDate, CreditCard
FROM Order NATURAL JOIN Transaction
WHERE Delivered = true
AND CreditCard IS NOT null

DEFINE DESTRUCTIVE POLICY
AnonymizeCreditCard ON OldTxns AS
(Months(NOW-TxnDate)>12, SET CreditCard = null)
```

Observe that the extended retention condition R_p on this record and policy pair is ($Months$(**NOW** - TxnDate) > 12 $\wedge$ Delivered $=$ *true* $\wedge$ CreditCard IS NOT *null*) and all rows in the critical view must satisfy this condition. As a particular transaction for a delivered order becomes a year old, the destructive critical view will become non-empty, and consequently $\alpha(R)$ will be triggered. In this case the destructive action is simply to set the CreditCard attribute to null, and it is obvious that no tuple can simultaneously satisfy both the post-condition and the extended retention condition: the CreditCard attribute cannot be null and non-null at the same time. The unsatisfiability of $R_p \wedge \alpha_p(R)$ proves that $\alpha(R)$ will always expunge tuples from the critical view.

Note that more general destructive actions can also be handled by this correctness condition. For example if $\alpha(R)$ is changed to $SET\ CreditCard = Anonymize(CreditCard)$, and if the post-condition of $Anonymize(CreditCard)$ always invalidates the extended retention condition, then the destructive action is, again, verifiably correct.

If $R_p \wedge \alpha_p(R)$ is satisfiable, the process of proving this fact can be used to help administrators debug incorrect destructive actions. For if $R_p \wedge \alpha_p(R)$ is satisfiable, there will exist at least one possible tuple in the critical view that can serve as a counterexample in which performing the destructive action will not remove the tuple that violates the policy from the critical view. This counterexample serves to illustrate that the destructive action is not correct. At the same time, presenting this counterexample to the administrator may help in debugging this incorrect destructive action.

The implications of being able to statically verify that a record's life cycle will always terminate in destruction (removal from the critical view) are substantial. This proof of correctness of destructive policy actions provides the highest level of assurance for automated records retention compliance. The only downside is that, in the general case, checking for satisfiability is NP-complete. This is an improvement over the case of arbitrary record definitions and destructive actions, in which static verification is undecidable, but we still need to pay the cost of solving an NP-complete problem. We note that in most practical situations where the number of predicates in the extended retention condition will be limited, modern SAT solvers will be able to prove correctness for individual policies efficiently. Furthermore, the cost of verifying correctness has to be incurred only once (offline) for any given set of policies and destructive actions. Restricting the expressiveness of records to select-project-join views also allows us to verify non-invasiveness of actions and avoid cyclic execution of policy actions, since the problem of isolating policy interference is equivalent to statically determining relevance of actions on destructive critical views [7].

4.3 Integrity Preservation

Yet another challenge in providing support for data retention policies is to ensure that actions performed by retention policies never compromise the integrity of the database.

DEFINITION 8. *A destructive policy P is **integrity preserving** if its destructive actions $\alpha(R)$ when applied to any valid instances of the database will always lead to a valid instance of the database with respect to all integrity constraints.*

The notion of integrity preservation for a destructive action is distinct from the notion of correctness defined earlier. For example, the action can set a primary key value to null, which would satisfy the correctness criteria stated in Lemma 1, but would violate integrity. Conversely, a destructive action can be integrity preserving without being correct. For example, the action can simply do nothing, which would leave the integrity of the database intact but would not eliminate the tuples from the critical view. Once again, statically verifying integrity compliance is undecidable in general, but the class of records specified by updatable conjunctive views and simple destructive actions on them can be restricted further to be made integrity preserving, as described in the remainder of this section.

4.3.1 Primary Key and Uniqueness

Modifications to primary key and unique attributes cannot be checked for uniqueness without executing additional queries. Similarly, we can not statically determine the non-existence of foreign key references to a primary key value

being deleted. Hence, for proving correctness of destructive actions on such attributes independently of the database instance, we must restrict these actions to deletions. Furthermore, if the action requires eliminating a primary key value, the only option that can be statically proven to be integrity preserving is to delete the entire tuple from the critical view, and only if this deletion is guaranteed to be non-invasive (through possible cascading delete foreign key constraints) for all protected critical views (Section 4.4).

4.3.2 Foreign Keys

Modifying foreign key attributes is slightly less restrictive, and if the foreign key attribute is nullable then it can always be set to null. However, in cases where null is not suitable, we can overwrite the foreign key value with a different but valid one. To motivate a scenario where this may be useful, let us assume that the CreditCard attribute in the example of Section 4.2 was a foreign key reference and that customers in our database are uniquely identified by their credit card numbers. An extended schema and a different destructive policy is described below:

Order(OID, Delivered)
Transaction(Txn, OID, TxnDate, CreditCard)
Customer(CreditCard, CustomerName)

```
DEFINE DESTRUCTIVE POLICY
AnonymizeCustomer ON OldTxns AS
(Months(NOW-TxnDate) > 12,
SET CreditCard = 1111-1111-1111-1111)
```

Note that this policy assumes the existence of an artificial customer (say John Doe) identified by the fictitious credit card number that is being used for anonymization. However, to ensure correctness of this policy, the foreign key value (the tuple for John Doe) must exist in the Customer relation at the time of policy instantiation, and the system must guarantee that the statically used primary key will itself never be modified or deleted. This requirement ensures that the destructive action remains valid with respect to the integrity constraint throughout the lifetime of the policy. This constraint on the artificial customer (statically used primary key value) can itself be modeled as a protective retention policy as follows:

```
DEFINE RECORD Customers AS
SELECT CreditCard
FROM Customer

DEFINE PROTECTIVE POLICY
ProtectJDoe ON Customers AS
(CreditCard = 1111-1111-1111-1111, update)
```

4.3.3 Triggers and Check Constraints

With modern database systems supporting triggers and check constraints of arbitrary complexity, it becomes significantly harder to determine statically the consequences of actions taken in response to policy violations. For example, if our attempts to remove outdated records are rejected by a user programmed trigger, we will be unable to offer compliance guarantees unless the trigger is removed (or suspended) and the destructive policy actions repeated. Suspending trigger invocation altogether while retention actions take place is a simple solution, but then extra care must be taken in programming retention actions.

4.4 Delete-Protect Conflict Detection

The final requirement for proving correctness for a given set of protective and destructive retention policies is to show that they are conflict free. Inter-policy conflicts are caused when expired data that is to be removed or modified according to a destructive policy is to be retained under a protective policy at the same time. Detecting conflicts can be easily accomplished using the framework for correctness that we described earlier.

DEFINITION 9. *A destructive policy D is **conflict free** with respect to a protective policy P if D is non-invasive with respect to P.*

If the actions specified by a destructive policy cannot impact the critical view of a protective policy then the two policies are guaranteed not to conflict with each other. As before, the problem of isolating conflicts is reduced to that of determining whether a given update (destructive policy action) is irrelevant to a given relational expression (protected critical view).

As in Section 4.2 we rely on the result that detecting irrelevant updates for select-project-join view based records is decidable (although it is NP-complete), whereas in the general case detecting conflicts is undecidable [7]. To prove total correctness for a given set of policies, we have to show that all pairs $\{P,D\}$, where P is a protective policy and D is a destructive policy, are conflict free. If conflicts are detected, the retention manager(s) responsible for the policies can be informed, so that appropriate and defensible resolutions can be devised.

5. IMPLEMENTATION

Our proposal for records retention uses active integrity constraints and actions on views. We assert that the task of periodically monitoring destructive critical views and purging records from them will not be a significant source of performance overhead in our model. This is because of the temporal flexibility available in destructive data retention requirements: as long as records are deleted by well-documented regular maintenance routines, the legal retention requirements will be met. For example, this flexibility allows us to execute daily or weekly batch jobs that enforce destructive retention policies without interfering with the online operation of the database system.

However, if a record is accidentally deleted when it is supposed to be protected by an organization, the consequences are more serious. Naturally, there is no flexibility in protecting records and every modification to a database has to be checked against the relevant policies to ensure compliance. Consequently, the cost of computing the effect of updates on protected records will be the most significant source of overhead. Thus, the aim of our experimental evaluation is twofold: first, to measure the overhead of continuous record protection for a broad mix of protective policies in a high-update and heavily regulated business scenario; and second, to determine and recommend means of minimizing this overhead using features already present in database systems.

There are two widely used mechanisms to detect events specified by arbitrary relational expressions (such as a tuple becoming critical) in database systems: incremental computation or total re-computation. More specifically, to detect changes in the contents of a critical view, we can either *materialize the view* completely and implement triggers on

Policy #	1	2	3	4	5	6	7	8	9	10	11	12
Critical Row Coverage	0.08%	1.8%	6%	2.8%	6.6%	2.6%	5%	5.3%	0.6%	13.3%	2%	100%
Type	C	C	S	C	S	S	C	C	C	A	A	A

Figure 1: A summary of the policies used for experiments. The critical row coverage represents the proportion of TPC-H Orders and LineItem tuples that were subject to a specific policy. The type denotes the complexity of the critical view: simple, on a single relation (S); conjunctive, involving multiple base relations (C); or involving an aggregate value (A).

the materialized view (a feature present in some commercial database systems such as Oracle) or *implement triggers on base relations* to detect whether the relevant critical view will be impacted by a triggering statement (possibly after execution of an additional query) [8]. A detailed examination of support for triggers on views, including an algorithm for mapping triggers on views to an equivalent set of triggers on base relations, has been presented in the literature [17].

We note that for monitoring critical views, there are two well known optimizations that can be exploited to reduce overhead. First, we can benefit from the observation that certain policies naturally favor a particular mechanism to detect violations. For example, using triggers to monitor a critical view with an aggregate value is ill-advised, since this value requires re-computation at every relevant update. Generally, the decision to materialize or to re-compute depends on the average cost incurred per relevant update, and most database optimizers can quite easily assess the cost of a typical update and re-computation query, to give a reasonable estimate of which technique will be better than the other. The second optimization relies on the observation that policies can quite often be clustered around a small number of related tables. Instead of instantiating a large number of triggers on base relations, the approach of trigger grouping [12, 17] can be used to reduce the number of triggers per table and to exploit the fact that multiple policies on similar predicates can be checked by a single trigger invocation. The result of these optimizations can lead to a significant reduction in the cost of view monitoring.

6. EXPERIMENTAL EVALUATION

Our tests use the TPC-H benchmark database representing a business scenario involving the sale of parts to customers worldwide. We developed 12 different protective data retention policies and translated them into critical views that would need to be monitored. These policies were directly derived from real-world records retention requirements imposed on TPC-H like businesses by various security, export, and taxation agencies. Examples of such policies include protecting purchase orders involving the sale of special parts like uranium fuel rods, protecting order details with particularly large monetary sums, and protecting sales tax totals for specific countries. Our policy set consists of 3 simple policies on single relations, 6 policies leading to the monitoring of conjunctive views, and 3 policies that protect an aggregate amount. We define a policy's critical row coverage as the fraction of tuples subject to protection/monitoring in the base relations. In choosing our experimental policies we ensured that all but one has a low tuple coverage, and yet every tuple is protected against modification by at least one policy (Figure 1). A detailed description of the motivations behind these policies, their parameters, and the resulting critical views is available in our full report [5].

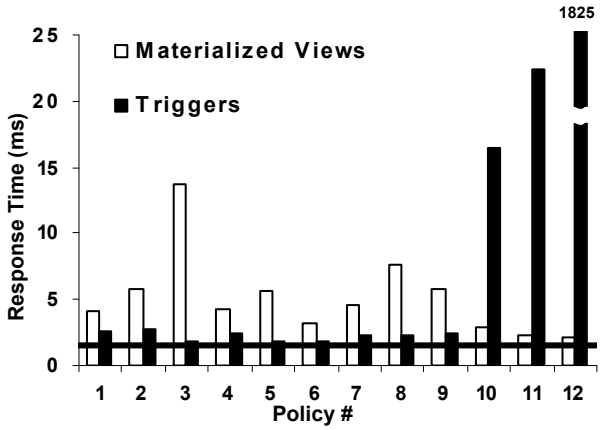

Figure 2: Average completion (commit) time for an update to a single tuple in a base relation protected under a single policy implemented using triggers over base relations and using the incremental view maintenance approach. The horizontal line represents the cost of an update without any monitoring mechanisms in place.

Our tests were conducted using a 1GB dataset on an Intel Core 2 Duo (1.8Ghz) machine with 1.5GB of RAM. All tests were performed on a warm database using DB2 v9.5 and involved issuing several thousand individual update statements on base relations that impact critical views defined by protective retention policies. We issued update statements until the confidence intervals for expected update time were very small, which typically required issuing 3,000-5,000 updates. The standard error in measured wall-clock response times in our tests was usually less than 5% of the average time to commit for an update.

Figure 2 summarizes the results of the overhead incurred by each of the 12 protection policies when they are individually implemented as a materialized view and as a set of (base relation) triggers. All updates were performed randomly over the dataset (each tuple was equally likely to be modified). Policies involving aggregated information (policies 10, 11 and 12) clearly favor incremental computation (the materialized view approach), whereas the maintenance overhead caused by materialization for other policies is far greater than simply checking the relevance of updates using triggers on base tables.

The reason for triggers individually performing better than materialized views for event detection in most policies is largely due to the TPC-H schema and data specification. For example, a purchase order in the TPC-H schema is related to only one customer, nation and region. Furthermore, the data contained in any purchase order includes at most 7 line items. Consequently, the majority of simple policy decisions

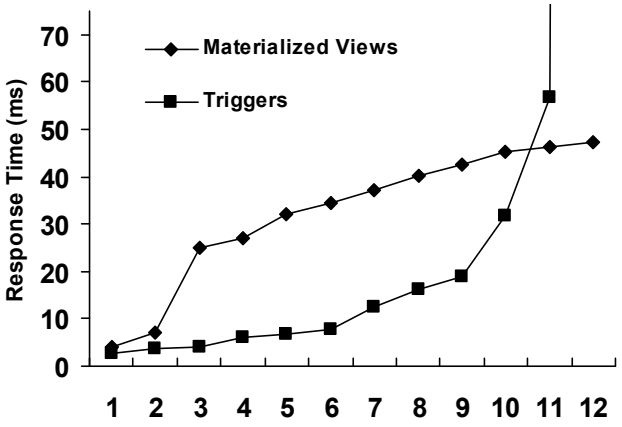

Figure 3: The scalability of incremental computation (materialized views) and total re-computation (triggers) for detecting changes in the contents of critical views. The values on the x-axis represent the total number of protective policies being enforced on the database. For example at 7, Policies 1 through 7 are all being enforced at the same time.

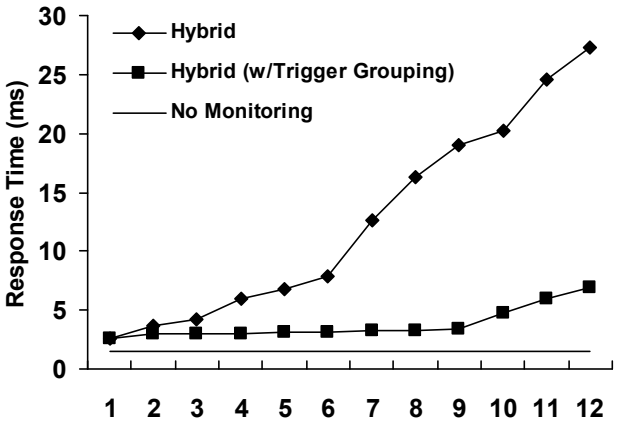

Figure 4: The scalability of the hybrid approaches to event detection on views. The values on the x-axis represent the total number of protective policies being enforced on the database. For example at 7, Policies 1 through 7 are all being enforced at the same time.

on updates pertaining to individual purchase orders can be made by examining a small number of tuples. Only when a large number of tuples has to be examined (e.g., for policies involving aggregated totals) does the active re-computation required when using triggers pay a heavy price.

Figure 3 demonstrates how both the incremental (materialized views) and total re-computation (triggers) approaches scale as more and more policies are implemented. Materialization suffers from the overhead of view maintenance whereas triggers have scalability problems arising from one policy on a view being translated into multiple triggers on base tables. It is worth mentioning that most commercial database systems have strict limits on the number of triggers that can be instantiated on a relation (typically fewer than 64). Consequently, it is very unlikely when dealing with a large number of policies that triggers would be able to accommodate the monitoring of all protective critical views. Note that the policies where triggers incur a high overhead are deliberately introduced as the last three policies in the mix. Thus, the figure misleadingly seems to favor the use of triggers for critical view monitoring. However, the total cost associated with monitoring all policies using triggers is far greater than that of using materialized views. It is clear that neither triggers nor incremental view maintenance alone can work well in general for monitoring a broad variety of records and policies.

We can instead combine the two monitoring methods into a hybrid critical view monitoring technique that attempts to take advantages of the positive aspects of both approaches. Simply choosing the correct and more suited event monitoring mechanism for each policy can substantially reduce the cost associated with monitoring a collection of critical views. Figure 4 shows that the simple hybrid strategy performs better than either technique alone. However it still suffers from having to instantiate a large number of triggers which greatly reduces its scalability. Since the TPC-H schema is relatively compact, and a large number of our policies revolve around the Orders and LineItem tables, we also recommend trigger grouping [21], which provides a significant improvement in response times by minimizing the number of trigger invocations per update on a base table.

Finally, we note that our tests stress retention monitoring much more than real world scenarios would. The foremost difference is the non-uniform pattern for updates: all protected tuples are not equally likely to be modified. As noted earlier, many businesses do not actively modify temporally stable records (for example very old purchase orders). Consequently, the use of temporal or range based indexing and partitioning will substantially reduce monitoring costs. Second, we note that our tests were done using unrealistically high record coverage (each tuple was protected under at least one policy) and typical coverage can be expected to be much lower than what we tested against.

In practice, we recommend that the decision to use a particular strategy for monitoring views be made by the query optimizer after considering the expected number of policy violations, expected number of relevant updates, level of temporal stability exhibited by records, and types of views to be monitored. Given a particular workload, modern database systems are able to recommend materializations of views that can improve query performance. Much of the infrastructure to measure the cost and benefit of incremental computation of views versus re-computation costs of queries already exists. Therefore we believe that by using these existing features a "retention policy advisor" can be built such that given a set of record definitions, protective policies on records and an expected workload profile, the best monitoring mechanism can easily be determined.

7. RELATED WORK

Limited and managed data retention has long been of prime importance in the field of records management. Retention solutions for unstructured data, such as documents and email, are widely deployed in large organizations, and the importance of limited retention has also been acknowledged as a fundamental requirement in privacy aware Hippocratic databases [3]. The problem of unintentional data retention in relational database systems, even after explicit deletion by users, has been also been examined recently [18].

Unfortunately the need to have a formal framework for systematically creating, managing and enforcing records retention policies has been ignored for structured records.

The work by Garcia-Molina et al. [11] and Toman [19] in removing tuples from large fact tables in data warehouses when they are not required in answering a given set of queries is similar in spirit to ours. The problems being addressed, however, are quite different, and our framework extends the notion of removal of sensitive data to apply to records as views and takes into consideration the interaction between protective and destructive policies.

Other suggested approaches to solving the problems associated with data retention in relational databases have generally taken a very simplistic approach to the problem of managed records retention, where a piece of metadata, such as a purpose or expiry timestamp, is attached to abstractly defined facts that have retention obligations [2, 3, 16]. Such proposals also trivialize the problem to that of deleting records when they have outlived their purpose, whereas the vast majority of legal retention obligations faced by organizations are protective and not destructive. Needless to say that for provable regulatory compliance, the interaction between protective and destructive policies has to be examined carefully, and our research looks at the problem from both perspectives.

Our proposed framework for protective policies on records is related to several features present in traditional integrity constraint and materialized view mechanisms, which have been exhaustively examined in the past several decades. Efficient monitoring of records is directly related to several problems in efficient materialized view maintenance [7] and incremental evaluation [8]. The problem of policy enforcement and actions on views relies heavily on the notion of updatable views [9]. Verifying the impact of user defined actions on views [15] and assisting users in creating meaningful actions on views [13] have also been examined in the literature.

8. CONCLUSION

In this paper we present a view based framework for systematically creating and enforcing records retention policies in relational database systems. Using such a framework enables users to define records at any granularity and enforce a rich set of protective and destructive retention policies on these records. A significant benefit of our approach is that it specifies a formal criterion for correctness for any user defined record and policy action. We demonstrate that for the class of records specified by updatable conjunctive views and simple destructive actions on them, users can statically verify the effect of their policies and be certain that policy execution will *always* guarantee regulatory compliance. The requirement to use such views can be enforced syntactically. The layer of formal reasoning that we have developed can also be extended to encompass more expressive records (e.g., outer joins) and can be combined with a wide choice of anonymization functions. Our framework can be efficiently implemented using the infrastructure for triggers and materialized views that is part of almost all commercial database systems.

Acknowledgements

This work was funded by the University of Waterloo, Open Text Corporation, and the Natural Sciences and Engineering Research Council of Canada.

9. REFERENCES

[1] United States Public Company Accounting Reform and Investor Protection Act of 2002 (Sarbanes-Oxley Act) (Pub.L. 107-204, 116 Stat. 745). August 2002.

[2] Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, and Walid Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *ICDE*, 2005.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic databases. In *VLDB*, 2002.

[4] Alexander Aiken, Jennifer Widom, and Joseph M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *SIGMOD*, 1992.

[5] Ahmed Ataullah. A Framework for Records Management in Relational Database Management Systems. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, April 2008.

[6] Lars Bækgaard and Leo Mark. Incremental computation of time-varying query expressions. *IEEE Transactions on Knowledge and Data Engineering*, 1995.

[7] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 1989.

[8] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[9] Umeshwar Dayal and Philip A. Bernstein. On the updatability of relational views. In *VLDB*, 1978.

[10] Charles Elkan. A decision procedure for conjunctive query disjointness. In *PODS*, 1989.

[11] Hector Garcia-Molina, Wilburt Labio, and Jun Yang. Expiring data in a warehouse. In *VLDB*, 1998.

[12] Eric N. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J. B. Park, and Albert Vernon. Scalable trigger processing. In *ICDE*, 1999.

[13] Arthur M. Keller. Choosing a view update translator by dialog at view definition time. In *VLDB*, 1986.

[14] Bill Lipner. The million-dollar backup tape. *ComputerWorld Magazine*, August 2006.

[15] Claudia Bauzer Medeiros and Frank Wm. Tompa. Understanding the implications of view update policies. *Algorithmica*, 1986.

[16] Marco C. Mont and Robert Thyne. A System to Handle Privacy Obligations in Enterprises. In *Hewlett-Packard Internal Technical Report (HPL-2005-180)*, 2005.

[17] Feng Shao, Antal Novak, and Jayavel Shanmugasundaram. Triggers over nested views of relational data. *ACM Transactions on Database Systems*, 2006.

[18] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. Threats to privacy in the forensic analysis of database systems. In *SIGMOD*, 2007.

[19] David Toman. Expiration of historical databases. In *TIME*, 2001.