Trustworthy Vacuuming and Litigation Holds in Long-term High-integrity Records Retention

Ragib Hasan
University of Illinois at Urbana-Champaign
201 N Goodwin Avenue
Urbana, Illinois, 61801
rhasan@illinois.edu

Marianne Winslett
University of Illinois at Urbana-Champaign
201 N Goodwin Avenue
Urbana, Illinois 61801
winslett@illinois.edu

ABSTRACT

Relational databases are periodically vacuumed to remove tuples that have expired. During the discovery phase of litigation, plaintiffs ask defendants for access to information related to their case. The requested information is then subject to a *litigation hold*, which means that the information cannot be deleted. Vacuuming exposes a database to a new threat – adversaries can try to thwart database auditing mechanism by masquerading an illegal tuple deletion as a vacuuming operation, and delete an unexpired tuple, or a tuple under a litigation hold.

In this paper, we provide a generic framework for auditing vacuuming, augment existing database integrity audit mechanisms to support vacuuming, formalize the notion of a litigation hold, and identify key issues in the context of database systems for long-term, high-integrity records retention. Then, we propose several schemes for efficiently implementing trustworthy litigation holds. Finally, we evaluate the efficiency and tradeoffs of the different schemes using a series of experiments.

1. INTRODUCTION

Litigation is an inevitable part of doing business. At any given time, large companies are involved in an average of 147 lawsuits [3]. During a lawsuit, the plaintiff and defense often try to unearth potentially damaging information relating to the opposite party. In the days of paper-based recordkeeping, it was prohibitively costly, in terms of time and money, for litigants to go through millions of pages of paper documents belonging to the opposing party in the case. But today, 92–99% of all business records and data are created, processed, and stored as electronic records [16], and searching through large volumes of data has become very easy and cheap. Therefore, lawyers have become more likely to request a litigation hold on the opposing party's records. At the same time, electronic records can also be modified or deleted easily, and a company facing a lawsuit has high incentives to find and remove any material that may be adverse to it in the lawsuit.

In 2006, the Federal Rules of Civil Procedure (FRCP) [6] were amended to set the standard for such electronic data discovery. The amendments also specify the legal responsibility of the litigants, and set punishment for destroying information related to the litigation. Litigants in civil cases are entitled to request documents and data relevant to the case from the opponents [16]. All electronically stored data is subject to subpoenas. The FRCP and other precedents from several legal cases do allow some exceptions to the law, but only under certain circumstances. For example, information subject to lawyer-client confidentiality rules cannot be subject to a litigation hold. However, in most cases, failure to protect records related to the case from deletion under a data retention policy is considered to be an act of bad faith. Such actions, therefore, can cause the judge and the jury to make adverse inference, i.e., to assume that the destroyed records would have reflected badly upon the organization, and find a verdict against the company at fault.

An example of the importance of litigation holds can be found from the following real-life civil lawsuit. Zubulake v. UBS Warburg is a landmark case that defined the importance of litigation holds and the consequences of non-compliance [1]. Laura Zubulake filed a gender discrimination lawsuit against UBS. The judge instructed UBS to retain all data, documents, and email related to the case, until the case was over. However, later it emerged that UBS deleted important email and other data despite the hold order. This failure to preserve the data subject to the hold order cost UBS the case. It resulted in severe monetary sanctions against UBS, which ultimately had to pay $29.3 million in fines. The ruling of the case also expedited the reformation of the FRCP, and introduction of rules related to litigation holds on electronic records.

A litigation hold can be explicitly set by a judge, explicitly requested during the discovery phase by the opposing counsel, or be implicit – a company is supposed to automatically place a hold on the relevant records when it anticipates a lawsuit. Therefore, companies have strong incentives to be able to prove that they have placed litigation holds appropriately. Both the plaintiff and the defense may want a proof that the records placed under the litigation hold were not removed from the database or other repository where they reside during the time the litigation hold was in effect. For example, the plaintiff might want a proof or assurance that the records it receives during the discovery phase are indeed the records that were in the system when the hold was placed. The defense may also have its own incentives – it may want to show that when it was notified of the lawsuit, it duly placed a litigation hold, and did its part in preserving the records. As many corporate civil lawsuits involve millions of dollars, there is a strong financial incentive for a malicious insider to subvert the litigation hold mechanism. Therefore, we need a scheme for provably secure litigation holds.

In this paper, we focus on the specific problem of implementing trustworthy litigation holds in relational databases designed for high-integrity long-term retention [14] – a *retention database* for short. These retention database systems use a transaction-time database to store all versions of tuples. Normal SQL queries are translated to a temporal query by the transaction-time layer. So, regular user applications and queries can run without any changes. In such a database, modifying an existing tuple causes a new version of the tuple to be inserted. SQL deletes result in insertion of a special deletion record, but older versions are retained. Each organization maintains a set of *retention policies*, as mandated by law and/or corporate strategy. Each database tuple has a specific retention period, defined by a retention policy. After the retention period ends, the tuple is shredded from the database. The shredding is performed by a periodic process which vacuums all the tuples that have expired after the last shredding event.

In such a database, existence of a litigation hold on a tuple will prevent it from being vacuumed even if it has expired. An adversary can attempt to remove tuples under a litigation hold in two ways: first, she can try to remove a tuple under a hold before it has expired. For example, the adversary can bypass the DBMS and remove the tuple from the DB files by editing it in a binary editor. However, researchers have developed trustworthy auditing schemes to detect any such attempts by an adversary to tamper with the current database content [14]. The other option is to remove expired tuples still under a hold, by subverting the vacuuming process. In this paper, we address this particular threat of making vacuuming trustworthy for retention databases.

Contributions. The contributions of this paper are as follows:

1. We formalize the notion of litigation holds in databases for long-term high-integrity records retention, and define the threat model.
2. We present four schemes for implementing litigation holds. These schemes address the challenges of fast placement, removal, and auditing, as well as having a small footprint.
3. We provide an refinement to vacuum-aware audit scheme the Log-consistent Database Architecture (LDA), to make it efficient.
4. We present a generic scheme for verifying vacuuming events for compliance with expiration and litigation hold policies.
5. Through experiments with TPC-C data, we demonstrate the speed and space tradeoffs of the four schemes and show which scheme should be chosen in a variety of common situations.

The rest of the paper is organized as follows: in Section 2, we present background information related to DBMS architectures for long-term high-integrity records retention. We provide a threat model for database vacuuming and litigation holds in Section 3, and present the semantics for setting and removing litigation holds in Section 4. Next, we describe four schemes for implementing litigation holds in Section 5. We provide a generic model for vacuuming in Section 6. We discuss and evaluate performance in Section 7. Finally, we conclude in Section 9.

2. BACKGROUND

2.1 Litigation holds under US civil law

In the United States, the Federal Rules of Civil Procedure (amended in 2006) [6] defined the nature and scope of a litigation hold. According to rule 34(a) of FRCP, all electronically stored information (ESI) is subject to discovery by the litigating parties. A plaintiff is required to preserve and retain such information. The advisory committee on rules noted that, "When a party is under a duty to preserve information because of pending or reasonably anticipated litigation, intervention in the routine operation of an information system is one aspect of what is often called a *litigation hold*. Among the factors that bear on a party's good faith in the routine operation of an information system are the steps the party took to comply with a court order in the case or party agreement requiring preservation of specific electronically stored information" [6]. FRCP also defines the scenarios when a party becomes obligated to preserve information: "A preservation obligation may arise from many sources, including common law, statutes, regulations, or a court order in the case" [6].

FRCP Rule 37(f) prevents an organization from deleting expired tuples via regular cleanup mechanisms, which implies that ordinary data retention and shredding policies should not be applied to the data under a litigation hold.

According to Rule 26(f) of FRCP, opposing parties must organize a discovery conference to decide a plan for discovery. Based on the plan, information must be disclosed within the next 30 days (or any schedule set by court) per Rule 26(a)(1)(D).

2.2 Databases for long-term, high-integrity records retention

Government regulations have mandated the protection of integrity of business and medical records. In the United States, Federal and state regulations such as the Sarbanes-Oxley Act [21], Gramm-Leach-Bliley Act [7], and Securities and Exchange Commission rule 17a-4 require businesses to retain all routine records for specific multi-year retention periods. Non-compliance with these regulations can bring large fines as well as prison sentences for the company's officers. As a result, compliance-related software systems have proliferated in industry, as businesses seek turnkey solutions to compliance.

While most existing efforts have focused on non-structured data such as email and files, most business records are stored in structured form in relational databases. Mitra et al. presented the design of an architecture for databases for long-term, high-integrity records retention [14]. Their threat model translates the business-level threats targeted by Sarbanes-Oxley and SEC rule 17a-4 into IT-level threats. Their threat model assumes that users commit database records correctly, but after some time, regret their own actions, and wish to modify or delete records that are in the database. Users may also regret that certain information was not inserted into the database, such as a large purchase order that would have improved the bottom line during the previous financial reporting period. In this case, a user may try to insert backdated information into the database. The adversaries in their threat model are insiders who may have superuser privileges and who have incentives to undetectably modify or delete database tuples, or retroactively insert them. Mitra et al. provided a scheme where an auditor can detect any tampering attempt. This depends on the use of write-once read-many (WORM) storage devices, which are the standard file servers used in US public companies to satisfy retention requirements for email and other unstructured business documents.

Databases must be periodically vacuumed in order to shred expired records [18]. The vacuuming process looks at the retention policies, and using them it chooses which tuples need to be shredded from the database. A shredding action vacuums the tuple, i.e., removes it from the database as well as from the database index or any other location where the tuple may be mentioned or referenced.

In Mitra et al.'s Log-consistent Database Architecture (LDA) [14], a transaction-time database is used. Insertions of new tuple versions as well as vacuuming of expired tuples are recorded immutably in a special *compliance log* kept in a write-once-read-many or WORM storage device. Using a special cryptographic hash function, auditors perform an audit of the database by first computing a hash over the tuples currently in the database, and

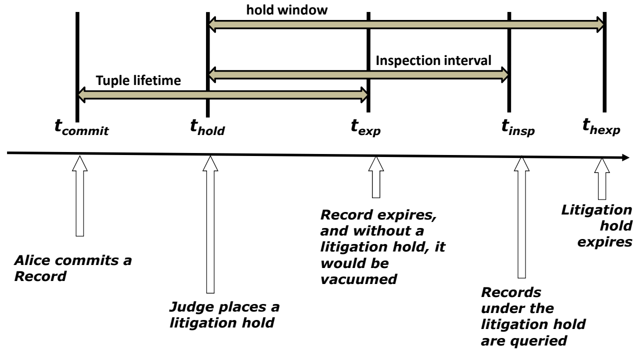
hold window
Inspection interval
Tuple lifetime
t_{commit}
t_{hold}
t_{exp}
t_{insp}
t_{hexp}
Alice commits a Record
Judge places a litigation hold
Record expires, and without a litigation hold, it would be vacuumed
Records under the litigation hold are queried
Litigation hold expires

Figure 1: Threat model parameters, and different events

comparing this to the hash of the union of tuples in the compliance log and the tuples present in the database at the time of the last audit. If the two hash sums are the same, then the database passes its audit and the auditor puts a signed snapshot of the current database state on the WORM storage server, for use at the next audit.

In LDA, a scheme for trustworthy shredding of records from databases for high-integrity records retention was presented. In this scheme, a periodic vacuuming process vacuums expired tuples. It runs as an ordinary transaction, except that it bypasses that transaction time layer. The vacuuming process log all the vacuumed tuples into the compliance log. During the next audit, the auditor checks an expiry relation (containing expiration times for various relations), and verifies whether the tuples recorded in this log had indeed expired at the time of the vacuuming event. The database audit process also takes into account that the shredded tuples are no longer in the current state of the database, and therefore the audit process is modified accordingly. However, LDA does not support litigation holds, and the vacuum-aware audit scheme as described in [14] is very inefficient as it requires the auditor to recompute the hash of the previous database snapshot (as opposed to using the stored hash from the last audit).

3. MODEL

The nature and scope of litigation holds on relational data depends on assumptions made about the system, operational, and adversarial models. In this section, we discuss these models. We start by looking into the system model.

3.1 System and operational model

Litigation involves events that have already taken place. Thus the records subject to litigation holds are also historical in nature, and will be subject to the usual records retention requirements of the business. We assume that the DBMS used for this purpose is a *records retention database* as defined in Section 1 (i.e., running on top of a transaction-time database). Each relation has an retention period, and therefore each tuple has an expiration time, after which it can be vacuumed from the database. We assume that the vacuuming process is not trustworthy, and can be subverted by the adversary. We also assume that the organization maintains write-once-read-many or WORM storage servers.

Such WORM servers are already widely deployed, and export a narrow file system interface. While most WORM servers use magnetic disks to store data, a firmware enforces the write-once semantics. Once a file has been closed, the WORM guarantees that it will be immutable for the rest of its retention period (as set by the administrator). The WORM servers sold by vendors today provide strong immutability guarantees – even administrators cannot overwrite a file stored on a WORM storage server, shorten its retention period, or delete an unexpired file once it has been committed to the server. (The WORM manuals actually instruct administrators to be careful in configuring the system, as any mistakes will render the WORM unusable.) We also trust that the WORM server uses a trustworthy and tamper-proof clock. Vendors already provide such functionality (e.g., the Snaplock compliance clock from NetApp).

During a civil lawsuit, opposing counsel intends to get access to the database records related to the case. Once the records are identified in the e-discovery phase, they are placed under a litigation hold, indicating that these records are not to be removed from the database, even if they expire. After e-discovery, the defense usually has up to 30 days to produce the held content for the disclosure phase. The records under the hold can also be subpoenaed any time during the litigation.

3.2 Adversarial model

We assume a strong insider adversary with superuser privileges in the system. Malory, the adversary, can impersonate any other local user, can delete, move, or create content on the magnetic storage used for ordinary database files. Therefore, he can tamper with the database by removing the tuples he does not want to retain. However, an auditor will be able to detect such tampering (e.g. using an existing audit scheme such as LDA [14]), and this will lead to adverse inference. Instead, we focus on the threat where Malory subverts the litigation hold itself.

We trust that existing files on the WORM storage servers cannot be subverted by Malory. This trust assumption is reasonable because WORM storage servers are designed to withstand exactly such attacks even by superusers. We also trust that when auditors run a query, she gets a correct answer. This is also reasonable, because if Malory modifies the DBMS engine, he will easily be caught by the auditor if she checks the hash of the code to detect any tampering with the DBMS software. Also, during an audit, the auditor can run her own software on the database directly, rather than relying on software supplied by Malory.

3.3 Threats

We address threats to the tuples that are under a litigation hold. As a superuser, or with help from a bribed superuser, Malory can of course remove such tuples. However, our goal is to ensure that any such attack will be detected by an auditor during the next audit. Going through the DB interface to remove such tuples will leave enough evidence that will show up in a future audit. We focus in particular on the trustworthiness of vacuuming, i.e., we want to ensure that if during vacuuming, Malory attempts to remove tuples that have expired but are still under a hold, it will be detected by the auditor.

Another threat may be that, when the litigation hold is placed during the e-discovery phase, Malory can try to prevent some or all tuples from falling under the hold. For example, Malory can present a partial view of the database to whoever is placing the hold. To protect against this, any litigation hold scheme must either require that the hold be placed by a trusted auditor, or else provide a way to verify that the hold was placed correctly.

Malory can try to tweak the DBMS server clock to set it in future during vacuuming. This can cause the vacuumer to incorrectly vacuum tuples that are yet to expire.

Finally, Malory can attempt to subvert the vacuuming process and the auditor by tampering with the retention period information, and other associated organizational policies. For example, suppose that the retention period for a business record is 5 years. However, Malory may want to change it to 1 year only, before the next vacuuming. This will cause all 1 year-or-older tuples in this relation to appear as expired, causing them to be vacuumed.

3.4 Threat model parameters

We illustrate the threat model in terms of the following events, as shown in Figure 1. t_{commit} is the time when Alice commits a tuple to the database. The tuple expires at time t_{exp}. At time t_{hold}, Judge Judy places a litigation hold on the tuple. Note here that t_{exp} must be after t_{hold} (i.e., $t_{exp} > t_{hold}$, because otherwise the tuple has already expired and will not fall under the hold by law, as long as it has been shredded. At time t_{insp}, Bob, the opposing counsel, asks Alice for a copy of all the records that fell under the litigation hold. At this point, Alice has to present all such tuples. Bob also wants a proof that these are indeed the tuples that fell under the hold. Here, t_{insp} can be before or after t_{expt}. If $t_{insp} < t_{exp}$, then the tuple has not yet expired, and therefore, an audit of the database will prove that this tuple has not been removed. If $t_{insp} > t_{exp}$, then the tuple has expired by the inspection point, but due to the litigation hold, Alice is required to retain it. Therefore, Alice has to prove that she has retained the tuple even though it has expired.

4. LITIGATION HOLDS IN RELATIONAL DATABASES

While the notion of litigation hold is well known in law, its application to relational data is comparatively new. In this section, we formally define a litigation hold in the context of relational databases.

4.1 Litigation holds

We model a litigation hold in the case of a database based on the requirements specified in FRCP:

For a database D, we define a *litigation hold* (LH) as follows:

$$LH = \langle q_{(t_1,t_2)}, t_s, t_e, D \rangle,$$

where $q_{(t_1,t_2)}$ is a temporal query over D (in the time interval $t_1 - t_2$), and t_s and t_e are the litigation hold start and end times respectively. Here, t_1 and t_2 represent the two extreme endpoints of the time epoch the query $q_{(t_1,t_2)}$ examines. For example, if $q_{(t_1,t_2)}$ asks for all patients who had a cancer diagnosis in 1994–1996 and had a diagnosis of diabetes within the four years immediately preceding their cancer diagnosis, then $t_1 = 1990$ and $t_2 = 1996$. By definition of litigation holds, the times t_1 and t_2 must be prior to t_s, i.e., the time when the litigation hold starts. So, we have the following constraints here: $t_1 \leq t_2 \leq t_s < now$, and $t_s < t_e$.

Under FRCP, the result of running the temporal query q should remain invariant between t_s and t_e. We define $View(q_{(t_1,t_2)}, t_{hq})$ as the view of the database using the query $q_{(t_1,t_2)}$ on the database D at time t_{hq}. We define $TH(q_{(t_1,t_2)}, t_{hq})$ as the set of tuples which contribute to the different columns of $View(q_{(t_1,t_2)}, t_{hq})$.

FRCP requires that the data under the hold must remain in the database during the duration of the hold. We can interpret this in two ways. We define *weak litigation hold compliance* as follows: during the time interval $[t_s, t_e]$, $View(q_{(t_1,t_2)}, t_{hq})$ remains constant whenever $t_s \leq t_{hq} \leq t_e$. In other words, for any time t_{hq} between t_s and t_e, $View(q_{(t_1,t_2)}, t_{hq}) = View(q_{(t_1,t_2)}, t_s)$.

We say that a database has *strong litigation hold compliance* if $TH(q_{(t_1,t_2)}, t_{hq})$ is the same whenever $t_s \leq t_{hq} \leq t_e$. In other words, for any time t_{hq} between t_s and t_e, $TH(q_{(t_1,t_2)}, t_{hq}) = TH(q_{(t_1,t_2)}, t_s)$.

The difference between strong and weak litigation hold compliance is that in weak compliance, we only require the materialized view at hold setup time to be protected, allowing expiration of non-related columns of the tuples that contribute to the view. In strong compliance, we require entire tuples to be retained.

4.2 Issues

As the law does not precisely define the semantics of a litigation hold, several issues have to be considered. Here, we focus on the following issues related to litigation holds, in the context of relational databases.

Which tuples should we hold? Essentially, a litigation hold query result is a view whose columns may have come from different tuples. Should we assume that all attributes of the tuples that contributed to the query results are subject to the litigation hold? Or, are only the contributing columns under the hold? Note that Rule 26(f)(3) stipulates that "documents must be produced in their original form", so if a tuple contributes to a query, we cannot shred its non-contributing attributes (i.e., change those attribute values to null values), as that might be considered a modification of the data by the opposing lawyers. On the other hand, if the lawyers only want to see the contents of the view, then non-contributing attributes of the underlying tuples can be removed without affecting the view.

What is the granularity of expiration? When a tuple has several attributes, do all of them expire at the same time? For example, a tuple may have financial information, such as total sales value in dollars, which the company may have to keep for a certain time as mandated by law. But suppose the same tuple also has the attributes "name" and "address". Do these attributes have the same expiration time as the sales value? The granularity of expiration has not been clearly defined anywhere. LDA [14] used the model where the lowest unit of specifying an expiration time was a tuple.

What operations to permit on held data? The next issue is whether we should allow ordinary user queries on data subject to a litigation hold. For tuples that have not yet expired, this is not an issue, as they are still in the database and would have appeared in a query even in the absence of a litigation hold. But for data that has expired, but is kept in the database only because of the litigation hold, the question remains – should this data appear in the answer to ordinary user queries on the database? Since the data has expired, it will not appear in the answers to non-temporal queries in any case. In some cases, it is not desirable that the expired data not appear in temporal query results either. In such cases, we may require the translation of a temporal SQL query by the transaction-time layer to include a new condition – namely to look into only unexpired data.

Do litigation holds expose companies to more problems? Finally, depending upon the specific mechanism of marking the tuple under a litigation hold, there is a possibility that future litigation may impose a hold on the previously held information. For example, suppose that Acme Inc. was sued by Trouble Inc. and therefore stored information about the related litigation holds in its storage. Later, DoubleTrouble Inc. sues Acme, and wants access to that stored litigation hold data of the previous case. Under current law, such hold requests are perfectly legitimate, as FRCP allows "all" electronically stored information to be discovered by litigating parties. Ideally, the information stored in response to a litigation hold would not expose the company to further litigation.

Next, we look at the semantics of setting a litigation hold.

4.3 Setting litigation holds

We propose that when a litigation hold is to be set during a discovery conference per FRCP Rule 26(f), the litigating parties define the set of holds, by specifying the temporal queries to be used and the duration of the hold. Then, the resulting set of LH specifications is given to a trusted third-party auditor.

The auditor is given access to the database. There can be a specific auditor role which the auditor can assume. We assume that a litigation hold query executes correctly, and returns correct query results. To ensure that this is true, the auditor can run the query in his or her own software and hardware environment. For less stringent guarantees, the auditor can simply perform an audit of the DBMS (using LDA's [14] audit scheme) right before submitting the litigation hold query.

Here, we describe an SQL-style language syntax for setting litigation holds.

SET HOLD $name$ AS q FROM t_s TO t_e

Here $name$ is a freeform label that identifies the hold, and q is the hold query. The DBMS should only allow this command for setting a hold, if it is given by a principal acting in the auditor role. This command causes the hold LH to be applied to the database D, and suspends the expiration times as specified in the retention policies for the affected data.

4.4 Renewing litigation holds

Litigation holds should be renewed from time to time if necessary. For example, it is not often known how long a particular civil litigation will last. A case may continue longer than originally thought, necessitating the renewal of the hold.

In our model, a hold LH can be renewed by an auditor using the following syntax:

EXTEND HOLD $name$ UNTIL t_e

This extends the end time of the litigation hold LH to t_e.

4.5 Removing litigation holds

Removal of litigation holds can occur in several ways. A litigation hold can expire because the end time t_e has passed. Or, the case may be over before t_e, so the company may want to remove the hold. Expired litigation holds could be removed automatically by a periodic process, but that introduces a new vulnerability – Mallory can try to convince that process into removing valid, unexpired holds. So, we require that litigation holds must be removed explicitly by a principal acting in the auditor role.

We define the following syntax:

DROP HOLD $name$

This causes the litigation hold LH on D to be removed. If the tuples affected under the hold are still in the database, the vacuuming process would now be allowed to vacuum them.

5. ARCHITECTURE

In this section, we define several schemes for supporting litigation holds. The goals of the design of the schemes are as follows:

- Space overhead: A hold should have a small footprint, and support overlapping holds with minimal space overhead.
- Placement time: The hold placement time should be small.
- Vacuuming: Vacuuming should not be adversely impacted by litigation holds.
- Querying: Querying for held data should be fast and efficient.
- DB Kernel: The holds should not require changes to the database kernel.

Through these schemes, we want to ensure that tampering with litigation holds will be difficult. Some of our schemes will prevent any tampering of the holds, while other schemes will not prevent the adversary from tampering with the hold, but will ensure that any such tampering will always be detected by an auditor during the next audit.

We model a litigation hold scheme in terms of the following: a hold definition step (used to set up a hold), a hold query method (used during vacuuming or audits to query for held tuples), a hold audit process (used to verify compliance with litigation holds), a hold renewal process, and a hold removal process. We assume that during an audit, the auditor can obtain the list of vacuumed tuples in a trustworthy manner by consulting log files on WORM. We elaborate on the auditing framework in Section 6.

5.1 Finding tuples that contribute to a view

To set a hold, we need to figure out which tuples *contribute* to a hold. Holds are defined by lawyers in natural language, and that descriptions of what to hold are most naturally expressed in terms of business objects. Such a hold can be defined as a query on the underlying relations. Placing a hold may require us to figure out which tuples actually contribute to the query results.

To determine which tuples contribute, we build upon the work by Cui et al. in [8]. Cui et al. showed that contributing tuples for a Select-Project-Join (SPJ) view can be determined through a single relational query over base relations. This lineage derivation scheme involves splitting the original query into queries for base relations. With some refinements, the same technique was extended to aggregation views as well.

As an example of Cui et al.'s scheme [8], suppose that we have the base relations ITEM(item_id, item_name, item_quantity), WAREHOUSE(store_id, warehouse_name, zipcode), and INVOICE(store_id, item_id, quantity, price, date). And we have the following view:

```
CREATE VIEW info61820 AS
SELECT warehouse_name, item_name, quantity
FROM WAREHOUSE, ITEM, INVOICE
WHERE INVOICE.store_id = WAREHOUSE.store_id AND IN-
VOICE.item_id = ITEM.item_id AND WAREHOUSE.zipcode="61820"
```

Suppose that in the materialized view for this, we have the row ⟨"Lowes","Box", 25⟩. To determine the actual tuples that contribute to this row, we can apply Cui et al.'s split operator to split the original query used in the view, and convert it into queries to the base relations, as shown below:

```
SELECT * FROM WAREHOUSE WHERE zipcode="61820"
AND warehouse_name="Lowes"
SELECT * FROM ITEM WHERE item_name = "Box"
SELECT * FROM INVOICE WHERE quantity=25
```

Note that FRCP does not define whether it is enough to enforce the hold on the materialized view, or whether we have to keep the original tuples. So, in the remainder of this paper, we provide schemes for both of theseoptions (i.e., store materialized view, or store contributing tuples).

5.2 Four schemes for litigation holds

Here we propose four schemes for litigation holds. Once a query or a set of queries is chosen, Audrey, a trusted auditor, is given the query to run on the database of the defendant. After running the litigation hold query, Audrey adds a temporal SQL clause to the query, indicating that it was run on the database at a particular time. Then she signs the litigation hold query and stores it into the WORM. The litigation hold problem is to ensure that during the lifetime of the hold, the answer to the hold query remains the same, or the tuples contributing to it remain the same, even if the tuples expire and the vacuumer runs.

Broadly, we can divide the schemes into two categories: in the first group of schemes, the tuples constituting the query result, or a projection of those tuples are stored in a file on the WORM server, with the file's expiration time equal to the hold end time. The actual tuples can then be vacuumed from the database when they expire. The second group of schemes does not copy the tuples to WORM; instead they require the vacuuming process to check for holds before removing an expired tuple.

Next, we discuss the details of each of the schemes. For each scheme, we describe the hold setup process, vacuuming scheme, hold extension process, and the audit process. We also discuss the advantages and disadvantages of each scheme.

5.3 Scheme 1(a): Storing all query results

The simplest scheme involves storing the entire query result into the WORM. Note that, the query result may consist of attributes (i.e., columns) from many tuples.

Setup. The auditor stores the litigation hold query result in a file in the WORM. The auditor also computes a cryptographic hash of the query result, signs the hash, and stores it in the WORM. The expiration time of the file containing the query results and the hashes is set to the expiration time of the hold.

Vacuum. No change to the vacuum process is needed. When any tuple in the database contributing to the hold query result expires, it can be vacuumed from the database as part of the regular shredding process.

Audit. At the disclosure phase, the defendant simply presents the file containing the hold query result and the signed hash of the result to the plaintiff or the judge. The defendant cannot modify the query result without detection as it is signed by the auditor. Auditing such a litigation hold query is also very simple – the auditor verifies the signature on the hash of the query result stored on WORM.

Extension. To extend a hold, the expiration date of the file containing the query result needs to be extended to the new end date. Today's WORM servers provide an operation for this purpose.

Removal. When a hold is set, the file containing the hold query results is given an expiration time based on the hold duration. So when the hold ends, the file of results will also reach its expiration time. Then it can simply be removed from the WORM by the WORM administrator (since the file has expired, the WORM permits its deletion).

Advantages. Under this scheme, no change to database shredding/vacuuming is needed. The vacuumer can remove all expired tuples from the database. The vacuumer does not need to consider the holds. Audits also do not need to change; audit schemes such as LDA audit can proceed without considering the existence of litigation holds.

Disadvantages. There is a potential of a large overhead in terms of space. There might be several litigation holds issued from the same or different lawsuits. These litigation hold query answers may have substantial overlap. In that case, there may be several hold result files with a lot of common data that need to be stored on WORM. Also, searching, sorting, and querying the tuples will become difficult (to do that, the results would have to be exported to a database). It is also not very clear whether this will satisfy the legal requirement of keeping data "in original form", per FRCP. Finally, the view itself can be subject to a later litigation hold. As we are specifically marking the data related to the case and storing them on WORM, future lawsuits have the option to place a hold on the WORM data.

Analysis. The space overhead of this scheme depends on the size of the litigation hold query result. The execution time overhead for hold placement is the time required to write the query result to the WORM device, therefore it is also dependent on the query result size. The execution overhead for vacuuming is zero. The litigation hold audit overhead is simply the time to verify the signature on the stored query result, therefore it is also dependent on the size of the query result.

5.4 Scheme 1(b): Storing all tuples contributing to query results

In the next scheme, we require storing all the tuples in that contribute to the query result, in a file stored on the WORM.

Setup. The auditor runs the litigation hold query, and then applies Cui et al.'s algorithm to decompose the query into queries on base relations. Thus, the auditor finds the list of contributing tuples. The auditor stores all of these tuples in a file in the WORM. The auditor also computes a cryptographic hash of these tuples, signs the hash, and stores it in the WORM. The expiration time of the file containing the query results and the hashes is set to the expiration time of the hold. Whether we hold all the contributing tuples, or only the contributing columns of those tuples is a separate and interesting question, as we discussed in Section 4.2.

Vacuum. Like scheme 1(a), no change to vacuum process is needed. When any contributing tuple expires, it can be vacuumed from the database as part of the regular shredding process.

Audit. The audit of the litigation holds in this scheme is the same as scheme 1(a). Similarly, database integrity audits (e.g., LDA-style) are not affected,

Extension. Extending a view is done in the same manner as scheme 1(a).

Removal. This is the same as the removal scheme of 1(a).

Advantages. The advantages are the same as scheme 1(a). We have one additional advantage here: since the contributing tuples are stored rather than just the query result, this scheme fulfills the legal requirement of storing data "in original form", as required by FRCP.

Disadvantages. Like scheme 1(a), we face the same overheads due to overlapping holds. Since we are storing the contributing tuples in their entirety, we incur more space overhead than scheme 1(a). The space overhead ranges from, at a minimum, the same as for 1(a), up to something much larger. The worst case would be the case of aggregate queries. For example, if the query is to find the SUM of an attribute of 100,000 tuples, then for scheme 1(a), we will only need to store a single value, the sum. But for scheme 1(b) (as well as the following two schemes), all 100,000 tuples will have to be held.

There is also a chance of inadvertently disclosing sensitive information as the tuples are all stored on the WORM. For example, suppose that the original litigation hold query was just about all invoices from a particular store. Also, suppose that one of the contributing tuples is from the Customer relation. In this case, we store the tuple from the Customer relation in its entirety. So, all columns of the tuple from the Customer relations will be stored on WORM, and be potentially exposed to the other party, even though the litigation hold query only required keeping the customer name, and not the customer's other information. Thus, there is a possibility that this scheme might inadvertently disclose sensitive data to the other party in litigation. The additional fields of the Customer relation will also be subject to future litigation holds, even after the original tuples expire.

Analysis. The space overhead of this scheme depends on the size of the litigation hold query result, and ultimately, the number of the tuples contributing to the query results. The execution time overhead for hold placement is the time required to write the contributing tuples to the WORM device, therefore it is also dependent on the size of the hold query result and the contributing tuples. Like the previous scheme, the execution overhead for vacuuming is zero. Time overhead to extend a hold is the same as scheme 1(a). The litigation hold audit overhead is simply the time to verify the signature on the stored tuples, therefore it is also dependent on the size of the contributing tuples. Finally, the removal overhead is essentially the time needed to delete the expired file of tuples from the WORM, and therefore very small.

5.5 Scheme 2: Hold counter

In this scheme, we introduce a new relation named the hold counter relation. The schema of this relation is as follows: $\langle\, tuple\_ID$, $hold\_ID \,\rangle$. The relation can be stored in the same database, or can be put on a separate transaction-time database containing only this relation. The tuple ID field contains the tuple ID of the held tuple. The mapping between litigation hold IDs and the corresponding litigation hold queries should be maintained on an append-only file on the WORM.

Setup. The scheme works as follows: when there are no previous litigation holds. the hold counter relation is empty. Whenever a new litigation hold is being set, the auditor computes the set of contributing tuples (using the same scheme as in 1(b)), and then for each such tuple, inserts a row into the hold counter relation. The new row contains the tuple ID of the contributing tuple, and the ID of the new hold.

Vacuum. Whenever the vacuuming process is presented with a candidate for vacuuming, it should query the hold counter table to check if any row exists with the same tuple ID. If the query result is empty, then the tuple is not under a hold, and can be safely vacuumed. If the query result is non-empty, then the tuple is under one or more holds, and therefore the vacuumer should skip it.

Audit. For each vacuumed tuple, the auditor issues a temporal query to the hold counter table (i.e., a query to be evaluated at the time of vacuuming). If the query result shows that the tuple was under one or more litigation holds at the time of vacuuming, then it shows that the tuple should not have been vacuumed, i.e., the vacuuming violates the litigation hold.

Extension. Extending a hold in this scheme requires no action, since the hold counters are removed explicitly (see next).

Removal. When a hold expires, the auditor needs to issue a query to remove all tuples from the hold counter relation that match the hold ID of the expired hold.

Advantages. There is no need to store the (potentially large) query result or tuples on WORM. Even in case of multiple litigation holds, we need to store only a small amount of data (i.e., 1 row containing the tuple ID and the hold ID) for each hold a tuple is subject to, in the hold counter table. Querying such a hold counter table is also fast if the table is small or there is an index on the tuple ID. Finally, the space overhead is independent of the size of the query result, or the sizes of the contributing tuples.

Disadvantages. Since this scheme considers contributing tuples, we face the same problem as Scheme 1(b) for aggregation queries, where the query result is small, but the list of contributing tuples is large. The vacuumer and the auditor need to be aware of the holds, so both of these processes need to consult the hold counter table when vacuuming, or checking vacuuming operations. Also, since the hold counter table will be placed on regular database storage, it may be subject to tampering by the adversary. However, we show next that any such tampering can be detected, and the hold counter scheme is trustworthy.

Analysis. The space overhead for this scheme is related to the average number of holds on each tuple in the original database. If the average number of holds on each tuple is n_h, then the space overhead is $O(N \cdot n_h)$, where N is the total number of tuples in the database. The vacuuming overhead for a given tuple is small, and is constant, as all the vacuuming process needs to do is to query the hold counter table with the tuple ID of the tuple to be vacuumed.

We argue next that the hold counter scheme is trustworthy, even though it is kept in regular database storage. We can use an LDA-style audit [14] to check if the hold counter table has been tampered with. The LDA audit described in [14] stores a special compliance log on WORM. Even though the adversary can tamper with the database content, she cannot tamper with the compliance log stored on WORM (because of the immutability properties of WORM devices). Therefore, an auditor can detect the presence of tampering.

So, the first task the auditor needs to do during an audit of vacuumed tuples is to perform an LDA audit on the hold counter relation. If the relation has not been tampered with, then the auditor can use it to verify the vacuuming. Note that the vacuumer does not need to perform such an audit. We assume that the vacuumer will be run periodically by the DBMS administrator, and therefore, we do not trust that it will behave honestly. The hold counter scheme will not prevent a malicious vacuuming process (run by an adversary) from vacuuming tuples still under a litigation hold. Rather, what we guarantee is that such activity will be detected during the next audit. Such detection of tampering will also result in adverse inference against the organization in the litigation.

5.6 Scheme 3: Bloom filters

In this scheme, we utilize Bloom filters – an efficient probabilistic data structure which can be used to check membership in constant time [4]. Bloom filters do not have any false negatives, but may have a small amount of false positives. The key idea is to create one Bloom filter per hold, and use them to check for memberships during vacuuming and audits.

Setup. Like schemes 1(b) and 2, the auditor first finds the list of contributing tuples. Then, the auditor creates a Bloom filter for the contributing tuples. The Bloom filter is signed by the auditor and both the filter and the signature are stored on WORM in a file. The expiration time of the file containing the Bloom filter is equal to the hold expiration time.

For a given number of tuples and a fixed false positive probability, the two parameters in building a Bloom filter are the number of hash functions, and the size of the Bloom filter. Based on this, we can compute the size of the Bloom filter. For example, for 4 hash functions and 2.5*10^{-5} false positive probability, we need 32 bits per inserted tuple. This will cause a false positive only for 1 in 25,000 tuples. We can also choose to have a fixed size for all of our Bloom filters, in which case the false positive probability will vary. We can save space by using compressed Bloom filters as described in [15].

Bloom filters can also be merged by taking a bitwise OR of the filters. This increases the false positive rates. What the auditor can do here is to store the individual Bloom filters per hold, as well as maintain a global filter. This will help optimize vacuuming.

Vacuum. During vacuuming, the vacuuming process checks the unexpired Bloom filters to see if a given tuple falls under any litigation holds. An expired tuple may have been originally inserted into the database *after* some Bloom filters (and the corresponding holds) had been placed. So, for each expired tuple, we only check the Bloom filters (i.e. holds) created after the commit time for the transaction corresponding to that tuple – we ignore all holds placed before the commit timestamp of that tuple. For matching against multiple Bloom filters (all having the same number of hash functions), we only need to hash the tuple key once per hash function, and then match it together with all the unexpired Bloom filters. To make this faster, we can also match against the union of similar sized Bloom filters (or the global Bloom filter, which is the union of all existing unexpired Bloom filters). If there is no hit in the global Bloom filter, then the tuple is not subject to a hold. If there is a hit, then the vacuumer can check the tuple against the individual per-hold Bloom filters to check if the tuple falls under any of them.

Audit. The auditor examines the vacuumed tuples, and checks whether each was a member of any Bloom filter stored in the WORM and still active at the time of the shredding. To root out false positives, the auditor can do other checks after getting a hit in a stored Bloom filter with a given tuple. This might involve checking the litigation hold query specification, and then seeing if this tuple will be included in the query result. Depending upon the nature of the query, this may be very complex. Therefore, it is best to use large Bloom filters to reduce the error rate to very low. For each 4.8 bits per tuple added to a Bloom filter, we can reduce the error rate by a factor of 10.

Extension. To extend a hold, the expiration time of the file containing the Bloom filter corresponding to the hold needs to be extended to the new hold end time. Like scheme 1(b), depending on the WORM mechanism, this can be done by extending the file's expiration time.

Removal. When the litigation hold ends, the file on WORM containing the Bloom filter should expire (since its expiration time is equal to the hold end time). Then the expired hold can be deleted by the WORM administrator. All global filters must be recomputed at that point.

Advantages. Space overhead of a Bloom filter is much lower than many other schemes. For 1% error rate, we need only 9.6 bits per tuple, regardless of the size of the tuple. Membership tests also takes a constant time (dependent on k – the number of bits in the bloom filter). Also, WORM stores only the Bloom filter, not the actual tuples, so accidental disclosure of sensitive information is prevented. If opportunistic opponent lawyers subpoena Bloom filters to examine the contents, a Bloom filter by itself will reveal little about the data because of the one way nature of the hash functions.

Disadvantages. In this scheme, we need to check each tuple against the existing Bloom filters. Since companies face many lawsuits at a time, there may be many Bloom filters existing at any time. So, the cost to check against each Bloom filter adds up. As shown above, we can optimize this by maintaining a global Bloom filter, and only in case of a hit on the global filter, we can check the individual per-hold Bloom filters. The global filter must be recomputed each time a hold is dropped.

The other big disadvantage is the presence of false positives. While we can design filters to have a very low false positive probability, with a large number of tuples in the database, we are bound to have false positives. What we need to do in case of false positives depends on the nature of the data and the organizational policy. One can choose to keep the tuple – this means the tuple was retained past its expiration even though it was not under a hold. Detailed checks to determine whether this tuple actually belonged to the hold (corresponding to the Bloom filter where the tuple found a match) will only slow down the vacuuming operation. On the other hand, many privacy laws require mandatory deletion or shredding of tuples beyond its lifetime, so in such cases retaining a false positive tuple may be illegal. In such situations, another scheme must be used.

Analysis. Hold placement time is the time to create a Bloom filter. This is dependent on the number of tuples under the hold. The vacuuming process involves a check for each expired tuple in each Bloom filter. For each tuple, we incur the cost of hashing the tuple key using the k hash functions. Using the various optimizations in the hash process, we can reduce this time. Auditing also involves a Bloom filter membership check. Removal of a hold is simply the time needed to expire a file from the WORM and recompute any global filters. This scheme cannot be used when laws mandate shredding of expired tuples, due to the scheme false positives.

5.7 Discussion

The four schemes described above have different pros and cons associated with them. Here we compare the schemes in different scenarios:

Size of query result. When the query result is small, but is computed from a large number of tuples, storing the result only using scheme 1(a) would incur the lowest space overhead. For example, in the scenario where we have an aggregation query (e.g., sum of all orders), then storing the result will involve storing only a single value. Storing all tuples used to compute the sum will cause a huge overhead if the scope of the query is large. Storing a Bloom filter will cost slightly less, but we still will require a lot of space.

If the size of the query result is very large, then the hold counter approach or Bloom filter approach becomes cheaper than the first two approaches. For example, if the result involves 100,000 tuples, then we store only 400,000 bytes for the Bloom filter (using 32 bits per tuple). But for the first two approaches, we have to store the entire 100,000 tuples or a projection of them, which imposes a lot of space overhead.

Audit and vacuum overheads. The first two approaches of storing results or tuples cost zero overhead in vacuuming. Tuples that expire can be shredded during the vacuuming process. For both the hold counter and Bloom filter approaches, we incur the additional cost of checking the hold counter or Bloom filter when examining expired tuples. The cost may be smaller in case of a hold counter as we only need to query the hold counter relation once, no matter how many holds there are, whereas in a Bloom filter, we have to check against all existing Bloom filters.

Querying held data. Since in the hold counter scheme and the Bloom filter scheme, the tuples stay in the database, it is easy to run different types of queries on them. However, in the first two schemes (store hold query results, and store contributing tuples), we need an additional step of exporting the result or the contributing tuples to a new database.

Overlapping holds. If there are overlapping litigation holds, then the first two schemes of storing contributing tuples or results will incur large overheads. In case of Bloom filters, we also would need to store multiple Bloom filters (unless we merge them using a bitwise OR at the cost of larger error probability). But in such a case, the hold counter approach works the best, as we incur only a small additional cost per new hold on each tuple.

We can of course use a hybrid approach, adapting the scheme based on the nature of the query. The initialization phase is common for all four schemes. So, we can design a hybrid scheme that uses the common initialization phase, and then based on the nature of the query, chooses one of the four schemes. This increases the workload of the vacuumer, as it has to check both the hold counter relation and any Bloom filters to determine if a tuple belongs to any holds (the first two schemes do not impact the vacuumer). If the number of holds is large, with multiple overlaps, the hold counter scheme may perform better than a Bloom filter, because the new Bloom filter will mean one additional filter to check for any expired tuple.

6. TRUSTWORTHY VACUUMING

In this section, we present a generic framework for auditing the vacuuming operation in a database. We start with the formal definition of vacuuming, and then explain the auditable vacuuming problem.

6.1 Definitions

Vacuuming in a transaction-time database corresponds to the systematic removal of expired database content and its supporting structures. In a transaction-time database, there are no deletions of tuples during regular transactions. Vacuuming is the only legal operation that can remove old versions of tuples. We can generalize the vacuuming process as a policy compliance problem as follows:

DEFINITION 1. *Vacuuming is a process $Vacuum(D, \mathcal{I})$, which takes a database D and $\mathcal{I}$ (information about expiration times of the tuples or the base relations, and existing litigation holds) as input, and produces a new database D' as output, where tuples in D' are a subset of the tuples in D, and D' is produced by shredding expired tuples from D while adhering to the expiration and litigation hold constraints.*

A *vacuuming event* $V(t_{vac})$ is a session of vacuuming initiated at time t_{vac} by the vacuuming process (vacuumer). A trustworthy vacuuming scheme is one where any vacuuming action can be verified for expiration and litigation hold policy compliance by an auditor at a later time.

6.2 A Framework for Auditing Vacuuming

Here, we propose a scheme to audit vacuuming events. Our scheme depends on a trustworthy method of finding the transaction times. To illustrate our scheme, we use the Log-consistent Database Architecture (LDA) as described by Mitra et. al. [14]. However, the scheme is generic enough so that any trustworthy scheme for finding the transaction times will be useful here. Recall that, in LDA, a special compliance log (where all tuple insertions and vacuuming are recorded) is stored in the WORM.

To initiate a vacuuming audit, the auditor first needs to perform an integrity audit (e.g., as done in LDA). A vacuuming event is valid if it did not vacuum any unexpired tuples, and it did not remove any expired tuples that are under one or more litigation holds. Therefore, the auditor needs a trustworthy method of knowing the expiration times of the tuples, and the set of litigation holds. The former can be found from the retention policies for the base relations for the tuples. For example, certain business records must be kept for 7 years according to SOX, so the expiration time for such tuples is set accordingly. This information about the expiration times for different relations can be stored on a file on WORM.

The main challenge in auditing a vacuuming event is to ensure that the vacuuming did not violate the existing expiration and litigation holds at the time of vacuuming. To audit this, the first task is to determine the time t_{vac} of the vacuuming event. We do not trust the vacuumer, so any statements made by the vacuumer about the vacuuming event time cannot be trusted. In LDA, the auditor can determine t_{vac} in a trustworthy manner by looking at the compliance log. The commit time of the transaction that vacuumed a tuple gives us the time t_{vac} for that tuple. In case of segmented compliance logs, the auditor can needs to identify the compliance log file segment containing for the tuple. The timestamp of the log file segment is the vacuuming event time t_{vac} for the tuple. Note that, there is a margin of error here, equal to twice the regret interval parameter of LDA.

We do however require that the vacuumer notes the start times for vacuuming operations in an append-only file on WORM. At the start of an audit, the auditor checks this file containing the start times for vacuuming operations since the last audit, and looks up the corresponding compliance log file segments of LDA. While we do not trust the vacuumer, the adversary gains nothing by lying about the vacuum start times. This is because if the adversary lies about the vacuum operation start times, then the auditor will detect that when she performs an integrity audit (where she needs to match the previous database state plus the newly inserted tuple and sans the vacuumed tuple with the current database state). If the claimed vacuuming operation start times and the timestamp on the corresponding log file do not match within a margin of one regret interval, then the auditor must consider it as a tampering attempt. Also, if the adversary does not note a vacuuming operation start time in the file containing the list of vacuuming operations, then the corresponding tuples will not be included in the tuple completeness check (described below). So, the adversary gains nothing by misstating or not stating the vacuuming operation start times.

Once the auditor figures out t_{vac}, she then accesses the policy repository to fetch the policies effective at that time, and the litigation holds and expiration information from the vacuuming information repository. The policy repository contains a set of files containing the set of policies for a given time period. The last write times of these policy files indicate which policies were in effect during a given vacuuming event. To check policy compliance at time t_{vac}, the auditor locates the policy file corresponding to that time, and reads the policies. The policy files should be signed by the auditor. If the policies rarely change, then the policies can be built into the vacuumer.

The auditor then accesses the vacuuming information repository and reads the supporting information. For example, the expiration times of different relations, and the litigation holds in effect, and other related information.

We present the algorithm for auditing vacuuming events below. Here we assume that $\mathcal{L}$ is the no-crash version of the compliance log, as stored in the WORM in the LDA architecture.

```
Audit ($\mathcal{L}, \mathcal{I}$)
 1: Let $\mathcal{L} = \{L_1, L_2, \cdots, L_n\}$ be the LDA compliance log (consisting of chronologically ordered log files $L_1 \cdots L_n$)
 2: Let $\mathcal{I}$ be the set of supporting vacuuming information, also sorted chronologically
    {Find the list of vacuuming events in $\mathcal{L}$}
 3: $V \leftarrow findVacEvents(\mathcal{L})$
    {For each vacuuming event, check policy compliance}
 4: for all $vacEvent \in V$ do
 5:   $t_{vac} \leftarrow getTimeStamp(vacEvent)$
      {Get hold information and retention periods at time $t_{vac}$}
 6:   $I_{t_{vac}} \leftarrow getInfo(\mathcal{I}, t_{vac})$
 7:   for all $tuple \in vacEvent$ do
 8:     if (hasExpired($tuple, t_{vac}, getExpirationTime(tuple, \mathcal{I}_{t_{vac}})$)==FALSE) then
 9:       return AUDIT_FAIL
10:     end if
        {check if tuple was under a hold}
11:     if (wasHeld($tuple, t_{vac}, getHold(tuple, \mathcal{I}_{t_{vac}})$)==TRUE) then
12:       return AUDIT_FAIL
13:     end if
14:   end for
15: end for
16: return AUDIT_PASS
```

6.3 Efficient vacuum-aware audits

During a regular audit in LDA, the auditor checks if the final database state is equal to the last database snapshot plus the new tuples noted in the compliance log. In terms of the hashes, this means that the hash of the tuples in the final database state should be equal to the hash of the tuples in the previous database snapshot plus the set of tuples inserted since then (as noted in the compliance log). Slightly abusing notation we can write this tuple completeness check of LDA as, $H(D_f) = H(D_o \cup L)$, where D_f is the final DB state, D_o is the previous DB snapshot, and L is the set of tuples that were part of the committed transactions as found in the compliance log.

However, the tuple completeness check does not hold when there are vacuuming events. This is because the new database state does not include tuples that have been vacuumed since the last audit. The old database snapshot stored on WORM contained the vacuumed tuples, so the union of the tuples in the old snapshot and the newly inserted tuples (as found from the compliance log) do not equal the tuples in the current database state. To take into account the effect of vacuuming, LDA provides the following refinement: the auditor finds the list of vacuumed tuples from the "shredded tuple" records of the compliance log. Then the auditor finds the corresponding pages in the previous database snapshot, and recomputes the hash of those pages, and ultimately recomputes the hash of the previous snapshot. Unfortunately, this refinement is inefficient, as it requires the auditor to recompute the hash of the previous database instance, rather than using the hash value from the previous audit (stored on WORM).

To make this efficient, we propose to enhance the LDA auditing scheme as follows: we redefine the vacuum-aware tuple completeness check as $H(D_f) = H(D_o \cup L - V)$, where V is the set of vacuumed tuples. Since we used an additive hash function in LDA, this is difficult to compute. We can rewrite the above equation as follows: $H(D_f \cup V) = H(D_o \cup L)$. In other words, the auditor simply checks if the union of the tuples in the final database state and the vacuumed tuples equal the union of tuples in the previous database snapshot, and the newly inserted tuples found in the compliance log. By using this tweak, we can use the old database snapshot hash stored during the last audit, rather than recomputing it.

7. EXPERIMENTAL EVALUATION

Setup. To evaluate the schemes, we used MySQL 5.1.34. The DBMS was hosted in a machine with a Pentium dual core 2.8 GHz processor, 512KB L2 cache, 4GB RAM, and a 1TB hard disk. We simulated the WORM server using a Pentium 2.8 GHz single core processor, 512KB L2 cache, with a portion of its local disks exported as an NFS volume. All the machines ran on Linux with kernel 2.6.11.

Dataset. To initialize our experiments, we ran 100,000 TPC-C transactions with a 512 MB DBMS cache and a 10 warehouse TPC-C configuration. We used the STOCK relation for placement of our holds.

Queries. We created five different holds on the STOCK relation of TPC-C. Each of the holds except the last one are queries that choose three attributes from the STOCK relations – the stock ID, the warehouse ID, and the quantity. The first four holds are designed using a query as follows:

SELECT s_id, s_{w_id}, $quantity$
FROM STOCK
WHERE condition

Here based on the condition used, different number of rows are returned in the result. We designed the condition so that, the first hold, Q1, covers 1% of tuples, the second hold Q2 covers 10% tuples, the third hold Q3 is actually a combination of 5 different queries, each of which cover 10% of the tuples, and all of these holds overlap (i.e., the holds cover the same tuples). The fourth hold Q4 covers 50% of tuples, and the fifth hold Q5 also covers 50% of the tuples, but this hold differs from other holds in that the query is an aggregation query instead of a selection query (i.e., SELECT SUM($quantity$) FROM STOCK WHERE condition). For the STOCK relation obtained after running 100,000 TPC-C transactions, the total number of tuples in the relation is 1,455,652.

For the hold counter scheme, we allocated 2 bytes for the hold ID, and 4 bytes for the tuple ID, resulting in a total of 6 bytes for each hold counter entry. For the Bloom filter scheme, we used 32 bits per tuple, and 4 hash functions, giving us an error probability

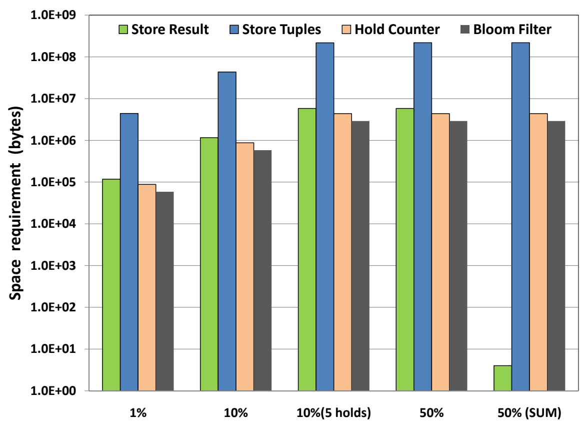

Figure 2: Space overheads.

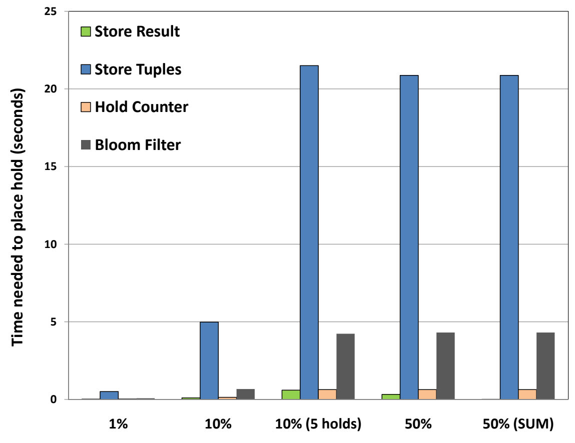

Figure 3: Hold placement time.

of 2.5*10^{-5}. For the first scheme, the view has 3 columns per row, requiring a total of 64 bits per row.

For each case, we measure the space requirements, hold placement time, vacuum overhead, and hold removal time.

Space overhead. The space requirements for the four schemes using the four queries are shown in Figure 2 (using logarithmic scale for space requirement on WORM). Holding all tuples costs almost 74 times than holding just the 3 columns of the tuples. In our setup, the Bloom filter approach requires the least amount of space.

However, for aggregation queries, the scenario changes. For Q5, the cost of all schemes except the first scheme of storing results is the same as Q4. But the space requirement of the scheme of storing results only, is drastically small – equal to the resulting value (4 bytes).

Hold placement time. The hold placement times are shown in Figure 3. Here, the scheme for storing the view results incurs the least amount of time. The cost of storing the tuples incurs the highest time overhead, as it has to store a lot of data.

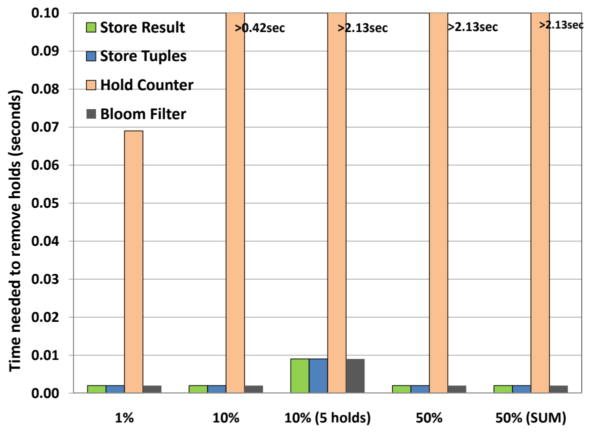

Figure 4: Hold removal time.

Vacuuming time. The vacuuming overheads for the first two schemes (storing results and storing tuples) are zero. For the hold counter scheme, it is the time to query the hold counter relation. For the Bloom filter based scheme, it is the time to read the existing Bloom filters, and to check each tuple against them. We assume that at each vacuuming session, 1% of the tuples, i.e., 14,556 tuples expire, and would have to be checked against the hold counter relation or the Bloom filters. For this part of the experiment, we also assume that the holds Q1 (a single hold) and Q3 (5 overlapping holds) are present. We precomputed the list of expired tuples, and measured the time overhead in terms of the time needed to query the hold counter table, and the time to match the tuples against the Bloom filters. Our experiments show that for a single hold, hold counter scheme requires about 0.04 seconds to check for 14,556 expired tuples, whereas the Bloom filter takes about 0.06 seconds to check these tuples against the Bloom filter. But when we have multiple holds at the same time (such as Q3), hold counters still need the same time (0.04 seconds), while the Bloom filter takes approximately 0.45 seconds to check against the five Bloom filters. It shows that the vacuuming overhead is lower for the hold counter scheme when there are many holds.

Hold removal time. We measured the time to remove a hold. Figure 4 shows the hold removal times. For the hold result, hold tuple, and Bloom filter schemes, the time to remove hold is simply the time to delete the corresponding (expired) file from the WORM. (Since the file's retention period is over by then, the WORM allows removal of the file). So, hold removal time for these schemes is extremely small. But for the hold counter scheme, removal of a hold involves sending a query to the hold counter relation to remove the rows corresponding to the removed hold. In our setup, it took MySQL 2.135 seconds to remove about 0.7 million entries from the hold counter table for Q3, Q4, and Q5. While this is higher compared to the other schemes, the absolute time is still reasonable.

8. RELATED WORK

The concept of litigation hold is new in the context of relational databases. Most of the existing research focus on forensic analysis of databases. However, litigation holds are special cases where the usual database retention policies are suspended. That is, in litigation holds, legitimate database operations on held should be suspended. The same operation would not have caused an audit failure if the hold were not in place.

Borisov et al. described a scheme for placing litigation hold on unstructured data in [5]. This scheme for documents allows only restricted queries on the keyword index. Auditors cannot access documents not under a litigation hold. Unfortunately, this scheme does not apply to the fine grained structured data stored in relational databases.

Researchers have looked into issues related to data retention policies in relational databases. Ataullah et al. addressed the related problem of writing and enforcing tuple retention and shredding policies, expressed as restrictions on the operations that can be performed on views [2]. Their approach relies on the DBMS to enforce the policies. Here we discuss the related and complimentary problem of how to enforce these retention requirements when the DBMS operator himself is a potential adversary.

Recently, researchers have developed a framework for auditing changes to a database while respecting retention policies [11, 12]. This framework enables policy based redaction and removal of sensitive information from the database and its history, and handling the uncertainties in answering audit queries from the resulting incomplete table and history. The litigation hold schemes described in this paper can be used on conjunction of such schemes.

Many regulations require shredding of expired data. Often companies have incentives for shredding old and expired data from a database. Several techniques have been proposed for scrubbing a database. For example, Stahlberg et al. discussed techniques for removal of any trace of a deleted tuple in [20].

Regulatory compliance database architectures have been proposed in [14]. This scheme prevents an attacker from surreptitiously modifying unexpired data items. The techniques proposed here can augment such compliance schemes to allow placement of litigation holds.

Snodgrass et al. discussed forensic analysis of databases in [19]. Such schemes allow detection of database modifications. However, removal of an expired tuple is indeed a legitimate operation, only suspended under specific condition. These forensic analysis techniques need to be augmented to take into account litigation holds using our schemes.

Various security problems related to outsourced database and file management have been discussed by many researchers [9, 10, 13, 17, 22]. Such schemes address the special case where the data owner is trustworthy, but the database server is not trusted. But in the compliance storage scenario, the data owner/operator is the primary adversary and has an incentive to tamper data.

9. SUMMARY

Litigation is an inevitable part of doing business. In this paper, we provided the first formalization of litigation holds in the context of relational databases for long-term, high-integrity retention of ordinary business records. We analyzed the legal rules and regulations to arrive at a set of research problems related to litigation holds. We then proposed a series of schemes for supporting litigation holds in databases for records retention, with different security and performance guarantees. Our schemes guarantee that once a litigation hold is placed, database content under the litigation hold cannot be surreptitiously removed from the database, even by the database administrator. The schemes we defined have different tradeoffs, which we displayed using empirical evaluation of different stages of the schemes.

While the law as of now does not precisely define the process of setting litigation holds, we posit that in near future, we will get clearer guidelines on the scope and extent of litigation holds on database data. This paper is the first step towards a systematic analysis of the litigation hold problem, and providing scalable, efficient solutions.

Acknowledgments. This work was supported by NSF under the grants CNS-0716532, IIS-0803280, and CCF-0938071.

10. REFERENCES

[1] Zubulake v. UBS Warburg LLC. 217 F.R.D. 309, 322 (S.D.N.Y. 2003), 2003.

[2] A. A. Ataullah, A. Aboulnaga, and F. W. Tompa. Records retention in relational database systems. In *Proc. of CIKM*, 2008.

[3] R. M. Barker, A. T. Cobb, and J. Karcher. The legal implication of electronic document retention: Changing the rules. *Business Horizons*, 52:177–186, 2009.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] N. Borisov and S. Mitra. Restricted queries over an encrypted index with applications to regulatory compliance. In *Proc. of ACNS*, pages 373–391, 2008.

[6] U. S. Congress. Federal rules of civil procedure. Online at http://www.law.cornell.edu/rules/frcp/, 2006.

[7] Congress of the United States. Gramm-Leach-Bliley Financial Services Modernization Act. Public. Law No. 106-102, 113 Stat. 1338, 1999.

[8] Y. Cui and J. Widom. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2), June 2000.

[9] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in database service provider model. In *Proc. of SIGMOD*, 2002.

[10] H. Hacigumus, S. Mehrotra, and B. Iyer. Providing database as a service. In *Proc. of ICDE*, 2002.

[11] W. Lu and G. Miklau. AuditGuard: a system for database auditing under retention restrictions. In *Proc. of VLDB*, 2008.

[12] W. Lu and G. Miklau. Auditing a database under retention restrictions. In *Proc. of ICDE*, 2009.

[13] G. Miklau and D. Suciu. Implementing a tamper-evident database system. In *Proc. of the Asian Computing Science Conference*, 2005.

[14] S. Mitra, M. Winslett, R. T. Snodgrass, S. Yaduvanshi, and S. Ambokar. An architecture for regulatory compliant databases. In *Proc. of ICDE*, 2009.

[15] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.

[16] J. Ruhnka and J. W. Bagby. Litigation support and risk management for pretrial discovery of electronically stored information. *The CPA Journal*, pages 2–6, May 2007.

[17] R. Sion. Query execution assurance for outsourced databases. In *Proc. of VLDB*, 2005.

[18] J. Skyt, C. S. Jensen, and L. Mark. A foundation for vacuuming temporal databases. *Data Knowl. Eng.*, 44(1):1–29, 2003.

[19] R. T. Snodgrass, S. S. Yao, and C. S. Collberg. Tamper detection in audit logs. In *Proc. of VLDB*, 2004.

[20] P. Stahlberg, G. Miklau, and B. Levine. Threats to privacy in the forensic analysis of database systems. In *Proc. of SIGMOD*, 2007.

[21] U.S. Public Law No. 107-204, 116 Stat. 745. The Public Company Accounting Reform and Investor Protection Act, 2002.

[22] M. Xie, H. Wang, J. Yin, and X. Meng. Integrity auditing of outsourced data. In *Proc. of VLDB*, 2007.