Ficklebase: Looking into the Future to Erase the Past

Sumeet Bajaj [#1], Radu Sion [#2]

[#] *Computer Science, Stony Brook University*
Stony Brook, NY, USA
[1] sbajaj@cs.stonybrook.edu
[2] sion@cs.stonybrook.edu

***Abstract*—It has become apparent that in the digital world data once stored is never truly deleted even when such an expunction is desired either as a normal system function or for regulatory compliance purposes. Forensic Analysis techniques on systems are often successful at recovering information said to have been deleted in the past.**

Efforts aimed at thwarting such forensic analysis of systems have either focused on (i) identifying the system components where deleted data lingers and performing a secure delete operation over these remnants, *or* (ii) designing *history independent* data structures that hide information about past operations which result in the current system state.

Yet, new data is constantly derived by processing existing (input) data which makes it increasingly difficult to remove all traces of this existing data, i.e., for regulatory compliance purposes. Even after deletion, significant information can linger in and be recoverable from the side effects the deleted data records left on the currently available state.

In this paper we address this aspect in the context of a relational database, such that when combined with (i) & (ii), complete erasure of data and its effects can be achieved ("un-traceable deletion").

We introduce Ficklebase – a relational database wherein once a tuple has been "expired" – any and all its side-effects are removed, thereby eliminating all its traces, rendering it un-recoverable, and also guaranteeing that the deletion itself is undetectable. We present the design and evaluation of Ficklebase, and then discuss several of the fundamental functional implications of *un-traceable deletion*.

I. Introduction

The "delete" operation in modern computer systems can at many times be an $illusion$ [65]. Although once deleted, data may no longer be accessible via legitimate system interfaces, numerous instances [29, 68, 70] have demonstrated that presumably erased data can be recovered with simple mining techniques.

Preserving un-wanted data not only jeopardizes user privacy & confidentially but can also violate retention policies set forth by legislations such as HIPAA [17], FERPA [8], FISMA [9], EU Data Protection Directive [7] and the Gramm–Leach–Bliley Act [10]. E.g., the fifth directive of the Data Protection Act [20] mandates that information shall not be retained for any longer than its intended purpose.

Prior work has addressed this issue on two fronts: "secure deletion" and "history independent" data structures.

The observation that data artifacts can linger in systems for a significant period after deletion [32, 40] gave rise to the requirement of "secure deletion". This is important since numerous system sub-components such as memory [33], storage mediums [39] and file systems [30] have shown to preserve deleted data. Applications such as databases hold on to deleted data in transaction logs, error logs, temporary tables, de-allocated data pages, index entries and audit logs [38, 69].

These data remnants can later be recovered by employing forensic analysis [3] techniques. In good hands [6, 14, 53, 62, 67, 71] these techniques are very helpful in incriminating wrong-doings by malicious users, however in the wrong hands they pose a grave threat to data & user privacy.

Mechanisms have therefore been designed to identify system parts where deleted data artifacts linger and subsequently remove them. Solutions have been proposed for general storage media [39, 45, 54, 72], file systems [24, 56] and database applications [69]. Also, off-the-shelf tools can now be used to perform such a secure data erase [5].

"History independent" data structures [48] (also referred to as "uniquely represented" data structures) have the property that their storage layout is a function of only the current state and not of the history of past operations that led to it. Such data structures reveal no additional information to an adversary outside of what can be inferred anyway via legitimate interfaces. If a delete operation is part of a system interface, then utilizing a "history independent" data structure ensures that – an adversary subsequently gaining access to the system storage is unable to infer whether the delete operation was performed at all. This is critical since *traditional data structures (e.g., B-Tree indexes) preserve (in their current state layout) information about past operations*. "History independent" variants have been developed for Hash Tables [61], 2-3 Trees [59], B-Trees [42] and Skip-Lists [43].

A third, largely ignored aspect in preventing erased data from being recovered concerns its relationship to the current state (data present in the system now). The main observation here is that side-effects of deleted data persist within the current state – this can be then exploited to derive information about the deleted data items in direct violation of regulation and the intent behind deleting the data in the first place.

For true regulatory compliance, "secure deletion" and "history independent" data structures are not sufficient by themselves but rather need to be augmented with mechanisms for full erasure of post-deletion data side-effects. We term this

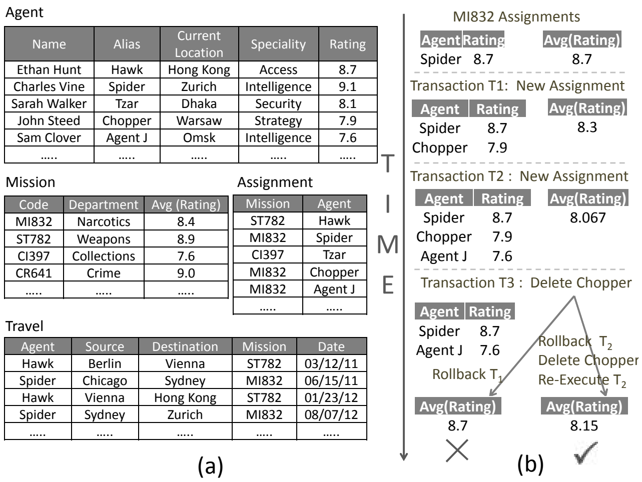


Fig. 1. (a) Intelligence Agency Database Snippet (b) Rollback vs Transaction Re-Execution

"un-traceable[1] deletion" (Section II).

In the following we introduce Ficklebase a relational database with fully un-traceable deletion guarantees.

II. Motivation And Concepts

The numerous regulatory compliant data expiration mandates derive from real-life privacy concerns in today's increasingly digital societies.

To illustrate how deleted/expired data can leave side-effects behind, consider a snippet from a hypothetical intelligence agency database (Figure 1(a)). As a simple example - suppose that once agent Sarah leaves the agency all evidence of her existence in the database needs to be eradicated. This would mean deletion of agent information tuples from the $Agent$ relation, travel information from $Travel$ relation and mission assignments ($Mission$). In addition, the $Avg(Rating)$ attribute in $Mission$ would need to be recomputed for each mission the agent was assigned to. Note that simple rollback of transactions that computed the $Avg(Rating)$ will not suffice, but they need to be re-executed to compute the new correct values (Figure 1(b)). If such a re-computation is not performed, any adversary that gains access to the database in the future can infer that deletion took place as well as potentially significant additional information about the deleted agent, by looking at properties of the tuples and indexes for the $Mission$ and $Agent$ relations.

At first glance it may appear that such a deletion can simply be performed by application logic. However, it becomes quickly apparent that, deletion of all data linked to the agent gets complicated, e.g., in the case of a transaction that uses agent information and mission data to generate new travel assignments for others. Also, implementing such a deletion in application logic (e.g. by using pre-defined Compensating Transactions [34, 55]) requires detailed semantic knowledge of all database transaction operations. Finally, this would also significantly increase the burden on database application developers. Ideally, removal of all traces of the deleted agent from the agency records should be supported transparently by the underlying database. Ficklebase achieves exactly this.

A. Concepts

To generalize, consider the following notation for a transaction T_j in a relational database - $T_j(R) \rightarrow (M)$, where R represents the read operations performed by T_j while M indicates the data modifications (update & insert) operations of T_j ($|R| \geq 0$,$|M| \geq 0$). Let r_{t_i}, u_{t_i} and i_{t_i} denote the read, update and insert operation respectively of a tuple t_i. Also $\mathcal{TS}(o_i)$ denotes the commit timestamp of the transaction that performs operation o_i.

Now suppose that the following transactions have been executed & committed (in sequence): $T_1() \rightarrow (i_{t_1})$, $T_2(r_{t_1}) \rightarrow (i_{t_2})$, $T_3(r_{t_2}) \rightarrow (u_{t_5}, u_{t_6})$, $T_4(r_{t_5}, r_{t_4}) \rightarrow (u_{t_7}, i_{t_{10}})$.

Let us first determine tuple side-effects in the above execution. Transaction T_2 read tuple t_1 and inserted t_2. Hence the insertion of t_2 is a side-effect of t_1. Similarly, transaction T_3 read tuple t_2 and updated tuples t_5 and t_6. Hence updates to t_5 and t_6 are side effects of t_2. In addition T_4 read tuple t_5 updated by T_3. Hence modifications made by T_4 i.e. update of t_7 and insertion of t_{10} are also side-effects of t_2 and so on.

Overall, the side-effects ($\mathcal{SA}$) are as follows:
$\mathcal{SA}(t_1)$ = (i_{t_2} by T_2, $u_{t_5}u_{t_6}$ by T_3, $u_{t_7}i_{t_{10}}$ by T_4)
$\mathcal{SA}(t_2)$ = ($u_{t_5}u_{t_6}$ by T_3, $u_{t_7}i_{t_{10}}$ by T_4)
$\mathcal{SA}(t_5)$ = $\mathcal{SA}(t_4)$ = (u_{t_7} $i_{t_{10}}$ by T_4).

It is to be noted from the above illustration that data side-effects go beyond simple primary-foreign key relationships. In fact, any data that is read & then results in modification of other data constitutes a side-effect and needs to be hidden after deletion (of the read data item).

Now, consider the case when tuple t_2 expires and is to be deleted. An *un-traceable delete* of t_2 should leave the database in a state such that no trace of t_2 is left behind, not even its effects on other data. This requires the following: (1) rollback of transactions $T_4 \& T_3$. (2) deletion of t_2 (or rollback of T_2). (3) re-execution of transactions T_3 & T_4. (this is necessary as illustrated by the average example of Figure 1).

This results in the execution schedule - $T_1T_3T_4$. Any database that performs operations equivalent to the above steps and achieves a schedule where the transaction that inserted t_2 never took place would achieve *un-traceable deletion* of t_2 (proofs in Section V)[2]. We define *side-effects* and *un-traceable deletion* in the following.

Definition 1: Side-effects of a tuple t_i ($\mathcal{SA}(t_i)$) are represented by the set of all data modifications (update and insert) operations such that

1) If $\exists T_j(R_j) \rightarrow (M_j)$ s.t. $r_{t_i} \in R_j$ and $\mathcal{TS}(T_j) > \mathcal{TS}(i_{t_i})$ then, $\forall o_{t_m} \in M_j, o_{t_m} \in \mathcal{SA}(t_i)$.
2) $\forall o_{t_m} \in \mathcal{SA}(t_i)$, If $\exists T_j(R_j) \rightarrow (M_j)$ s.t. $r_{t_m} \in R_j$ and $\mathcal{TS}(T_j) > \mathcal{TS}(o_{t_m})$ then, $\forall o_{t_n} \in M_j, o_{t_n} \in \mathcal{SA}(t_i)$.

[1]To differentiate from a "secure deletion" performed by overwriting.

[2]*Secure deletion* & *history independence* would still be required to truly erase t_2 (section IV-E)

Note that this definition is recursive but not circular. There are two reasons for this: (a) under a fully serializable mode of execution there exists a serial schedule (i.e. sequential with no overlap in time) of database transactions; (b) $\mathcal{SA}(t_i)$ includes database operations and not the tuples themselves. Consider the sequence $T_1(r_{t_1}) \rightarrow (u_{t_2})$, $T_2(r_{t_2}) \rightarrow (u_{t_1})$. Although it may appear circular at first glance, the operations u_{t_1} and u_{t_2} are distinct, thereby $\mathcal{SA}(t_1) = \{u_{t_2}\}$ and $\mathcal{SA}(t_2) = \{u_{t_1}\}$.

Definition 2: Un-Traceable Delete. Let the current database state be achieved by the following serialized transaction execution sequence - $\Gamma^{\mathcal{S}} = ...T_{j-2}T_{j-1}T_jT_{j+1}T_{j+2}$, where tuple t_i was inserted by transaction T_j. Then an *un-traceable delete* of t_i is a (set of) operation(s) that changes the current database state into a state *computationally indistinguishable*[3] from a state resulting from the execution sequence $\Gamma^{\mathcal{E}}$=$...T_{j-2}T_{j-1}T'_jT_{j+1}T_{j+2}$, where $T'_j = \phi$ or $T'_j = T_j - i_{t_i}$.

$T'_j = \phi$ when the application logic dictates that non-insertion of t_i means complete rollback of transaction T_j. In this case $\Gamma^{\mathcal{E}} = \Gamma^{\mathcal{S}} - T_j$. Otherwise, $T'_j = T_j - i_{t_i}$ e.g. when T_j inserts t_i using an insert-select query.

B. Applications

It is important to note that such an *untraceable delete* operation is very often not desirable – especially in scenarios involving data with real-life artifacts such as cash. E.g., consider a banking application that records money transfer between clients. If a client A has transfers recorded with another client B, then deletion of client A (when A closes its account), does not justify deletion of $A \leftrightarrow B$ transfers and their side-effects – since these "side-effects" are in fact the cash that now belongs to B!

On the other hand consider a privacy sensitive application that maintains confidential documents, records document accesses by its users and generates statistical or cross-document intelligence information. Once a document D is to be purged it is important to properly erase all associated access records & intelligence information deduced from D, lest this would reveal its existence as well as leak information from therein.

A third category of applications where an equivalent of *un-traceable delete* operation is desired is not privacy but rather functionality-centric: economic data such as the Current Population Survey (CPS) [4] are permitted to undergo revisions. A simple case for revision could be that an individual I is wrongly classified, which means deletion of I's information from the data set and its effects on computed statistics (e.g. average earnings).

C. Discussion

A database providing *un-traceable deletion* will in certain aspects function differently than a traditional database without it. Here we discuss some of these differences.

[3]No non-uniform probabilistic polynomial time algorithm exists that can distinguish between them [52]. Ficklebase in fact offers stronger information theoretic guarantees but we formulate this definition in terms of computational adversaries to allow for the deployment of cryptography in the underlying data structures and mechanisms.

Time-sensitive Queries. If a query running over "past" data (previously generated) is repeated, then the intuition is that database responses should be unchanged since the past has already occurred (e.g. order is shipped, patient is discharged etc). However, *un-traceable deletion* of one or more tuples that comprised the result set of such a query could result in a different response (for the same query) at a later time!

E.g., consider the query "find the number of agents that travelled on date d_t" on the sample database from Figure 1 which in SQL is – `SELECT COUNT(DISTINCT AGENT) FROM TRAVEL WHERE DATE =` d_t.

If a given agent was made un-traceable on date d_e, where $d_e > d_t$ then, the responses of the above query will be different on two dates d_1 and d_2 ($d_t < d_1 < d_e < d_2$), although the expected answer for the query on both dates d_1 and d_2 would be the same (in a traditional database).

Committed Transactions. Traditionally, once committed, transactions are treated as permanent and irreversible. However, with *un-traceable deletion* this is no longer the case. In fact for a database to support *un-traceable deletion* it must employ mechanisms to change the effects of transactions committed in the past. This is not only required but is also the most challenging requirement to meet.

External Application Logic. Un-traceable deletion can not be easily applied transparently for databases that are agnostic to the application logic semantics, e.g., when most of the business logic or functionality resides in application programs which in turn access the database externally via a standard SQL interface.

The reason for this is straightforward. For untraceability, the database must be able to re-execute transactions. This means that database must have access and understand all application logic. E.g., in Figure 1 if the database did not know that the $Rating$ statistic was computed as an average it would not be able to correctly remove the effects of deleted agent $Chopper$. While often this can be alleviated by moving as much as possible from the application logic into the data layer, for fully external logic, things can get complicated.

Delete vs Un-Traceable Delete. Note that the transaction notations in Section II-A and the definition of *un-traceable delete* (definition 2) do not include traditional delete operations (d_i). This is done for simplicity and also to differentiate between traditional delete operations (delete queries) from an *un-traceable deletion*. However, the inclusion of delete query operations are required and strongly supported.

III. Model

Adversary. We assume an adversary with full access to *today's* database. She can employ any mining or forensic techniques and wishes to recover information about any tuples deleted in the past.

Suppose that a tuple t_i expires and is made un-traceable at time E_t. Let D_{c_t} denote the database state at time c_t. Then the goal of *un-traceable deletion* is to prevent the adversary from recovering any information (via side-effects) about t_i

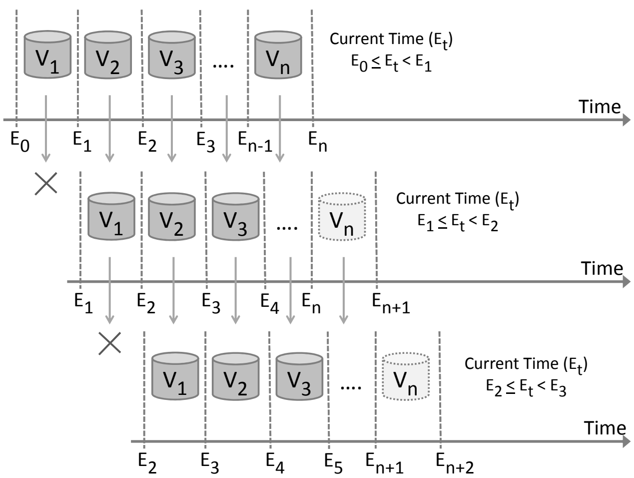


Fig. 2. Version maintenance & expiration with progression of time. V_j = Version$_j$.

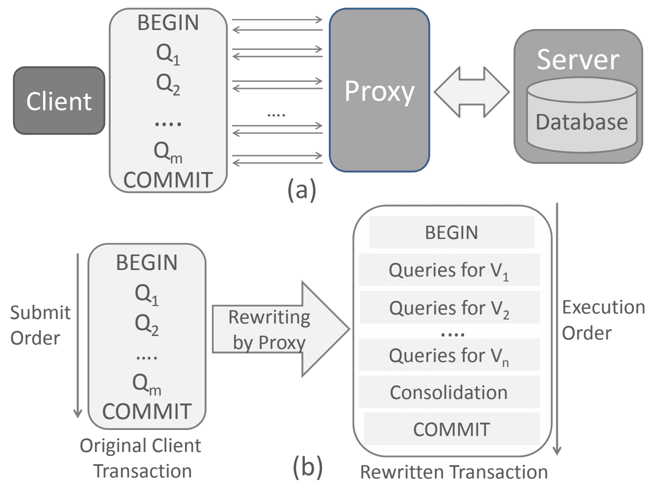


Fig. 3. Overview of (a) Architecture (b) Query Re-writing. Q_i = Query$_i$, V_j = Version$_j$.

(including its existence), while having full access to any (or all) of the database states D_{c_j}, where $c_j > E_t$.

Note that the case where the adversary gains access to any two database states D_{c_m} and D_{c_n} where $t_i \in D_{c_m}$ and $c_m < E_t < c_n$ is trivial, since the adversary can detect the deletion of t_i by merely computing the difference between the states D_{c_m} and D_{c_n}.

Data Expiration. Tuples come with associated expiration times, specified/computed at the time of their database insertion (or generation).

It is at this expiration time that a tuple needs to be deleted (un-traceably). Moreover, tuples are expired at fixed time interval granularities e.g., daily, weekly, monthly etc – a daily policy of tuple expiration means that tuples are deleted at the end of each day (say at midnight). For simplicity, we assume expiration times of all tuples coincides with the end of such a period.

Traditional delete operations (delete queries) can be executed at any time by clients. However, a tuple is deleted untraceably only at its expiration.

IV. ARCHITECTURE

A. Overview

At an overview level, one of the main insights behind Ficklebase is to maintain virtual "future" versions of the database in which the expired tuples are not supposed to exist. Ongoing transactions are then applied to all these (current and future) versions. This in effect constructs directly from the *untraceable delete* definition (Section II) in which if all transactions are applied to a database instance except for the insertion of tuple t_j, then t_j has undergone an *untraceable delete* in that instance.

This approach avoids two key problems: (i) keeping track of all system-wide side-effects, and (ii) retroactive rollbacks of committed transactions & their re-execution.

The maintenance of future versions, transaction application and expiration are entirely achieved using versioning and runtime query re-writing.

To exemplify, recall from Section III that tuples expire at fixed intervals of time (based on policy). We denote the end time of each such interval as E_i. For each time interval range a separate logical database version V_i is maintained (Figure 2) that contains only tuples with an expiration time $\leq \mathcal{E}_x(V_i)$, where $\mathcal{E}_x(V_i)$ is the time when V_i will be fully "expired". At any time E_t, $E_0 \leq E_t < E_1$, database versions V_1 to V_n exist with $\mathcal{E}_x(V_1) = E_1$, $\mathcal{E}_x(V_2) = E_2$ and so on.

Each client transaction T_j is then *transparently* applied to all these versions with the following restrictions: (i) when applied to version V_i, only tuples with expiration times $\leq \mathcal{E}_x(V_i)$ are visible to queries in T_j, and (ii) insertion of a tuple t by T_j in V_i is ignored if expiration time of t is $\leq \mathcal{E}_x(V_{i-1})$. Both (i) and (ii) are achieved through query re-writing (Section IV-C).

The net effect is that Version V_i is a database version wherein all tuples with expiration times $\leq$ $\mathcal{E}_x(V_{i-1})$ were never inserted. As a result, their side-effects are never propagated to any transactions and data structures in V_i (including underlying indexes etc). In effect all such tuples underwent an untraceable delete when version V_{i-1} expired at time $\mathcal{E}_x(V_{i-1})$ (see section V for proof).

The client application is not aware of the existence of multiple versions (other than the current version V_1) nor the application of transactions to versions other than V_1.

At any given time E_t only versions V_i where $\mathcal{E}_x(V_i) > E_t$ exist ($i \geq 1$). A version V_i is expired (utilizing a *secure delete* operation) at its expiration time $\mathcal{E}_x(V_i)$ (section IV-E).

To illustrate, at any time E_t, $E_0 < E_t < E_1$ versions $V_1, V_2, ..., V_n$ exist, with V_1 being the current version visible to clients (Figure 2). Once the current time approaches E_1, version V_1 is deleted, V_2 become V_1, V_3 becomes V_2 and so on. Also, $\mathcal{E}_x(V_1) \leftarrow E_2$, $\mathcal{E}_x(V_2) \leftarrow E_3$ and so on.

Finally, a new version V_{n+1} is created when a tuple with expiration time $> E_n$ is inserted by any client transaction T_j.

Components. Figure 3 (a) illustrates the main Ficklebase components. The main query re-writing logic resides in the Ficklebase proxy. The proxy intercepts all client queries and communicates with the server on behalf of the clients. The database server is an off-the-shelf DBMS.

Execution Model. A client transaction T_j is a set of serializable SQL statements T_j={begin,Q_1,Q_2,Q_3,...,Q_m,commit} where Q_j is a DDL (create/drop), DML (insert/update/delete) or a select query.

Algorithm 1 EXDT2VER

Input: sver INT, exdt DATE, policy INT
Output: version BIT(β_v)

ver $\leftarrow$ 0
ever $\leftarrow$ 0
switch(policy)
    case 1:
        ever = datediff(exdt, curdate()) + 1
    case 2:
        ever = period_diff(extract(year_month from exdt), extract(year_month from curdate())) + 1
    case 3:
        ever = (period_diff(extract(year_month from exdt), extract(year_month from curdate())) div 3) + 1
    case 4:
        ever = year(exdt) - year(curdate()) + 1
end switch
if ever $\geq$ sver **then**
    ver $\leftarrow 2^{sver-1}$
end if
return ver

B. Versioning

All versions are maintained within a single database instance. To limit storage overheads tuple copies are combined i.e. if tuple attributes have the same value across multiple versions then only a single copy of the tuple is maintained for all such versions.

A special *VERSION* attribute is transparently added to each relation by query rewriting (section IV-C) and is not visible to clients. The *VERSION* attribute is a bit field of size β_v wherein a bit b_i ($0 \leq i \leq \beta_v$) is set iff the tuple is valid in version V_i i.e. expiration time of tuple $\leq \mathcal{E}_x(V_i)$ [4].

Client queries only specify the tuple expiration times on insertion via the *EXPIRATION TIME* tuple attribute. Rewriting of insert queries (figure 5(a)) converts this expiration time into the correct value of the *VERSION* attribute[5]. This is accomplished using the *EXDT2VER* function (depicted in algorithm 1). Note that this is a sample function that implements daily, monthly, quarterly & yearly expirations. For additional functionality (e.g. hourly) necessary modifications should be made.

Also, tuples are copied "on write" only, when an update modifies an attribute value causing it to differ between versions. The version attribute is then automatically modified by query rewrites to indicate distinct tuple versions.

As an example, consider the following tuple t_j with k attributes – t_j = {*VERSION*=$0000..11$, *ATTR*$_1$=$value_1$, *ATTR*$_2$=$value_2$, ... , *ATTR*$_k$=$value_k$}. The *VERSION* attribute has bits b_1 & b_2 set indicating that the same tuple copy is valid in both, versions V_1 & V_2 i.e. expiration time of $t_j \leq \mathcal{E}_x(V_2)$. Now, suppose that an update query being applied to V_2 modifies *ATTR*$_1$ from $value_1$ to $value_1'$. Then a new copy of t_j is created such that
t_j = {*VERSION*=$0000..01$, *ATTR*$_1$=$value_1$, , *ATTR*$_2$=$value_2$, ... , *ATTR*$_k$=$value_k$} and
t_j' = {*VERSION*=$0000..10$, *ATTR*$_1$=$value_1'$, , *ATTR*$_2$=$value_2$, ... , *ATTR*$_k$=$value_k$}
The version fields of the original and copied tuples are updated (by query rewrites, Figure 5(b)) to correctly maintain distinct version copies.

C. Query Rewriting

Ficklebase relies heavily on query re-writing within the proxy. Each client query is transformed into a set of queries each of which is then classified as a *version specific* or *consolidation* query. Figure 3 (b) gives an overview of query re-writing along the order of query submission by the client and the order of execution after re-writing. *BEGIN* and *COMMIT* statements are executed as is at the start and end of the re-written transaction.

Version specific queries only affect the version that they are applicable to while *consolidation* queries ensure compact storage by combining tuple copies across versions.

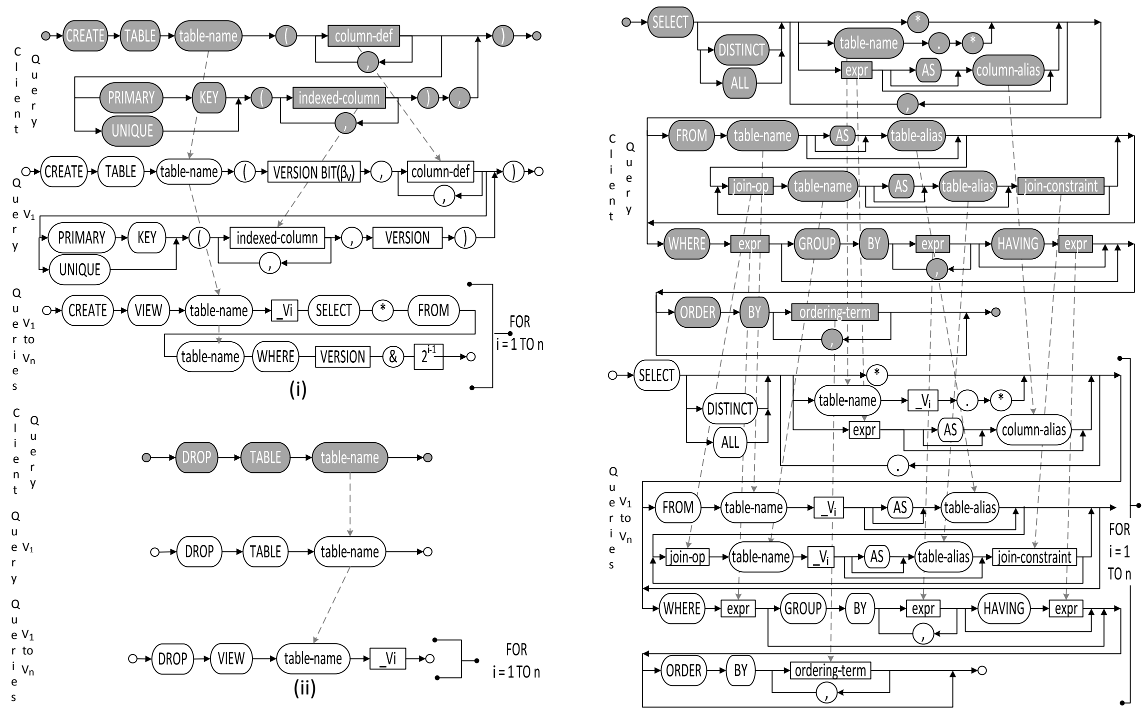
Figures 4(a) - 5(b) detail the query re-writing operations. We briefly discuss them in the following.

DDL (Create/Drop) statements [figure 4(a)]. DDL statements are re-written in order to (1) Transparently add the *VERSION* attribute to the relation being created. The *VERSION* attribute is also added (as the terminal field) on any indexes. (2) To create and drop *Version Specific* views which are later used in rewriting select & DML (insert/update) queries applicable to each version. A *Version Specific* view on a relation R for a version V_i selects only the tuples from R that are valid in version V_i. (3) Create additional indexes on the *VERSION* attribute thereby improving overall performance of the re-written transaction.

Select statements [figure 4(b)]. A select statement is re-written for each version. When applied to a version V_i it is ensured that the select only reads tuples valid in that version. This is achieved by replacing all table references in the select statement with the corresponding *Version Specific* views. Only the results of the select statement executed on version V_1 are returned to the client. The remainder of the results are filtered out by the proxy component. Thus the existence of other versions is hidden from client applications.

[4] The last bit (b_{β_v}) is used for consolidation and does not represent any version.

[5] Expiration times are only specified/computed in insert queries & cannot be updated at a later time – an almost pervasive requirement of most information life-cycle regulations.



(a) (b)

Fig. 4. Query Rewrites for (a) DDL (Create & Drop) statements. (b) Select statement.

The case where a transaction is read-only (i.e. comprises of only select statements e.g. a reporting application) is handled differently. In this case all queries within the transaction are applied only to the current version V_1 (and the results of each statement returned to the client). This is sufficient since read-only transactions do not modify any tuples, and hence do not generate side-effects.

DML (insert/update) statements [figures 5(a),5(b)]. Similar to select, DML statements are also re-written to replace table references with *Version Specific* views. DML statements also create new tuples (or modify existing tuples in case of updates), thereby bringing additional tuple copies into existence. These extraneous copies are combined together by consolidation queries which are generated for each relation in which either a tuple is inserted or modified by the client transaction. Also, similar to select, only results of statements executed on current version V_1 are seen by clients.

D. Version Specific Rollbacks

It is often desired by application logic that transactions be rolled back under certain conditions e.g. tuple not found. Note that these are not error/failure conditions such as duplicate key or deadlock (in which case the transaction is implicitly rolled back by the DBMS) but rather a part of application functionality. E.g. consider the following two queries submitted as part of a client transaction.

```
SELECT @d_next_o_id := d_next_o_id, d_tax
FROM DISTRICT WHERE d_id = 1 AND d_w_id = 1

ROLLBACK(ISNULL(@d_next_o_id))
```

Here, the client application desires that if no tuple is selected by the first select query then the transaction be rolled back. The *ROLLBACK* syntax is specially provided by Ficklebase for this purpose. Recall from Section II that it is essential for the database to posses the entire application logic. The *ROLLBACK* as shown in this example enables the specification of such conditions within transaction queries.

Now, it is entirely possible that when being applied to distinct versions a rollback may occur for certain versions, but not for others. Query rewriting in Ficklebase handles this at different levels (1) The client *ROLLBACK* statement is also re-written for each version, including creation of separate copies (for each version) of user-defined variables (like @d_next_o_id in the example above). (2) A savepoint is created on the database before execution of queries for each version.

When a rollback occurs (i.e. the condition in the *ROLLBACK* statement evaluates to true) for any version the Ficklebase proxy issues a SQL *ROLLBACK* statement to the database, rolling back the transaction up to the previous savepoint. This undoes the effects of all queries executed on that specific version. The effects of queries on other versions remain intact.

This is not unlike the case of nested transactions [64] with a distinct sub-transaction for each version. Individual sub-transactions can be rolled back without affecting the parent transaction.

E. Expiration

As illustrated in Figure 2 when the current time E_t approaches $\mathcal{E}_x(V_1)$, version V_1 expires. Expiration involves

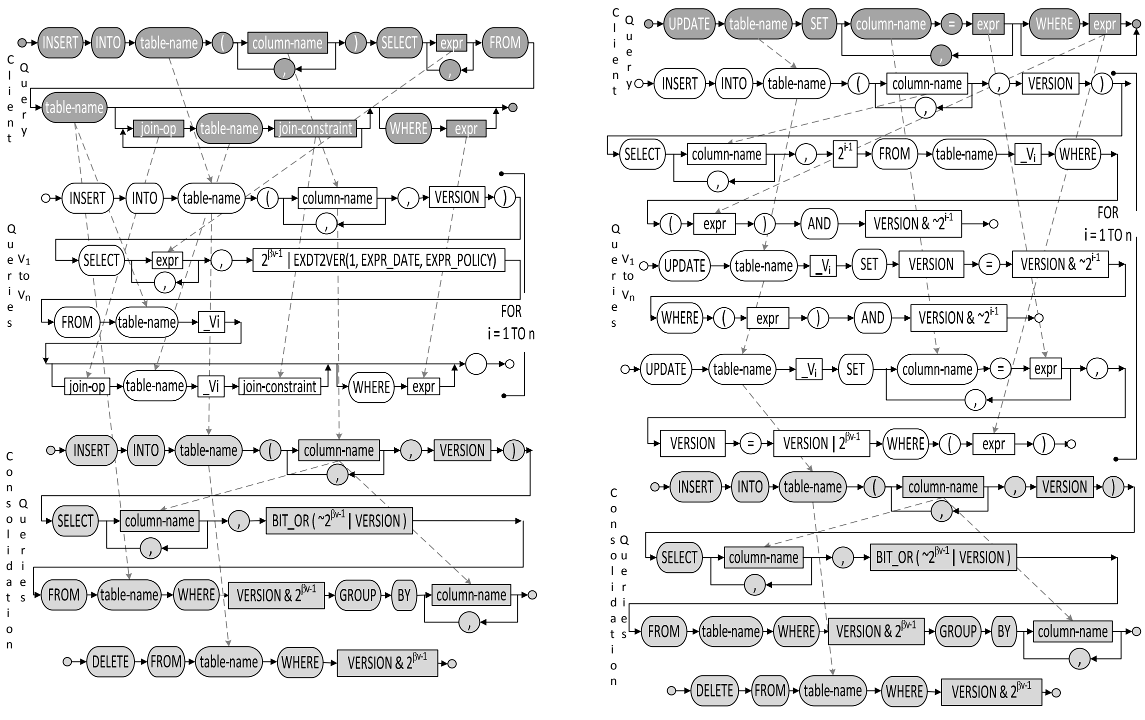


Fig. 5. Query Rewrites for (a) Insert statement. (b) Update statement.

the following (1) Physical delete of all tuples that were valid only until version V_1 i.e. tuples with expiration time $\leq E_t = \mathcal{E}_x(V_1)$. (2) Setting the next version V_2 as the current version V_1 visible to clients i.e. $V_i \leftarrow V_{i+1}, i \geq 1$ and thus $\mathcal{E}_x(V_i) \leftarrow \mathcal{E}_x(V_{i+1})$.

(1) + (2) are achieved by a scheduled task that executes (at the expiration interval $\mathcal{E}_x(V_1)$) a transaction comprising of the following queries for each relation R_i.

```
UPDATE Ri SET VERSION = VERSION >> 1

DELETE FROM Ri WHERE VERSION = 0
```

The next effect of the above queries is the deletion of all tuples that were valid only in version V_1 and not in any other version $V_i, i > 1$.

Secure Deletion. However, the execution of the above two queries is not sufficient to physically delete all tuples from the expired version for two reasons (1) Many database systems do not physically delete (at time of execution of delete statement) but rather mark tuples for deletion at a later time. This limitation of many popular database systems has been analyzed in [69]. (2) Other database components such as the transaction log, temporary files etc may still reveal deleted content.

Thus deleted tuples will remain in existence even after expiration. To avoid such leakage of deleted content we suggest adoption of *secure deletion* mechanisms from [69] wherein (1) During the execution of delete statement the space where tuple content resides is overwritten (by zeros). (2) Individual log records in the transaction log are encrypted to avoid revealing any deleted tuples. We refer the reader to [69] for details.

History Independence. Data structures such as B-Trees are commonly used by the underlying database storage engines. The storage layout of B-Trees (or variations such as B$^+$-Trees) is often a function of the (sequence of) operations performed on them due to their deterministic insertion & deletions. Thus, even after *un-traceable delete* & *secure deletion* (as above) an adversary gaining access to the database storage that utilizes these structures can potentially (by analyzing their layouts) still reveal deleted content. We note that although such deductions are difficult in practice [53, 69] they are nonetheless possible and in certain very specific cases trivial [53] (e.g. an index based on incrementing values). For B$^+$-Trees in particular the amount of information regarding past operations decreases as the fan-out increases and fan-outs in typical usages are usually large. Ideally a fan-out of N (N is the total number of tuples) stores all index values sorted in a single root node and is completely *history independent*, but not practical since it affects performance.

Such leakages from storage layouts of data structures can be prevented using one of the below two approaches (1) By the adoption of *history independent* versions of storage data structures such as B-Treaps [42] or B-SkipLists [43]. (2) By re-creation of index on expiration.

Although ready-to-use, well evaluated implementations of *history independent* B-trees are not yet easily available, it is nonetheless a promising direction to pursue.

(2) is a rather simple technique to employ and can be achieved by executing the following queries (or equivalent operations) for each relation R_i

```
CREATE TABLE Ri_tmp LIKE Ri

INSERT INTO Ri_tmp SELECT * FROM Ri
ORDER BY Ai

DROP TABLE Ri

RENAME TABLE Ri_tmp TO Ri
```

where A_i is a set of (any) attributes of Ri such that $|A_i| \geq 1$. All indices of the newly created relation would not have any delete operations performed on them and hence the effects of delete operations will not be evident in their layouts.

Thus, as suggested earlier a combination of *secure deletion* & *history independence* play a crucial role (along with Ficklebase) to ensure true erasure of deleted content.

Having adopted the above solutions the final component to tackle would be the underlying file system (if the database is not deployed using raw disks). A file system has mechanisms (also deterministic) to allocated free pages to the DBMS on request (e.g. when a B$^+$-Tree node is full and requires splitting). Hence not unlike the storage data structures this also warrants the use of *history independent* file systems. Although research on *history independent* data structures clearly points out their applicability to file systems providing usable implementations needs further investigation. An alternative approach (until the easy availability of such file systems) is to use a separate disk partition when re-creating the index. The new partition will have no imprints of prior file system operations.

F. Storage Analysis

Suppose the database comprises of N tuples and the number of active versions is n. Then in the worst case every tuple has a distinct copy in each version $V_i, 1 \leq i \leq n$ giving a overall storage requirement of $N \cdot n$. In the best case each tuple has the same attribute values for all of its versions and only N storage is required.

Now, suppose that each tuple is equally likely to expire at any $E_i, 1 \leq i \leq n$ and it has distinct copies for each of its version V_j s.t. $\mathcal{E}_x(V_j) \leq E_i$. Then the storage requirements are $N \cdot \frac{(n+1)}{2}$.

If we further assume a random distribution of client queries such that each tuple expiring at $E_i, 1 \leq i \leq n$ is equally likely to have j copies ($1 \leq j \leq i$), then the average storage requirement is $N \cdot \frac{(n+3)}{4}$.

This gives an overall storage complexity of $O(N \cdot n)$.

V. Untraceability

We now show that query rewriting and versioning indeed achieves *un-traceable deletion* as defined (Section II).

Let $\Gamma_{\mathcal{S}}$ denote the set of all transactions submitted by the client until time ℓ. Let tuple t_k with expiration time $E_t > \ell$ be inserted by a transaction T_j where $T_j \in \Gamma_{\mathcal{S}}$. The expiration of tuple t_k will coincide with the expiration of some version V_j i.e. $\mathcal{E}_x(t_k) = \mathcal{E}_x(V_j) = E_t$ (as per the expiration model in Section III).

Let $\Gamma_{\mathcal{E}}^{V_i}$ denote the set of transactions successfully executed (without being rolled back) on version i until time ℓ.

Then, Ficklebase guarantees that each version does not see any tuples that already should have been expired by the expiration time of the previous version. More formally:

Theorem 1: $\forall V_i$, $T_j \in \Gamma_{\mathcal{E}}^{V_i}$ iff. $\mathcal{E}_x(t_k) \geq \mathcal{E}_x(V_i)$, ("*un-traceable deletion* of t_k") [6]

Proof: (summarized). This follows naturally by the construction of query rewriting. We enumerate all transactions that "touch" V_i. First, for any incoming transaction, function EXDT2VER (Figure 5(a)) determines whether t_k is valid in (and should touch) for each specific version V_i i.e. whether $\mathcal{E}_x(t_k) \geq \mathcal{E}_x(V_i)$. If not, t_k is inserted with a zero-valued VERSION attribute such that t_k is not visible to any subsequent query on V_i. In the end, all tuples with zero VERSION attributes are deleted by the consolidation queries before the transaction commits (Figure 5(a)). Second, if the transaction rolls back on V_i then the rollback mechanism of Section IV-D ensures its queries have no effect on V_i. Third, when the current time is $> E_t$, all versions V_j where $\mathcal{E}(V_j) \leq E_t$ will expire and be securely deleted (Section IV-E). The only versions left would have expiration time $> E_t$. ■

VI. Discussion

Forensic Analysis. At first glance it may seem that *un-traceable deletion* rules out additional security mechanisms such as maintenance of audit logs to detect/prevent data tampering, since any additional recorded data opens up another avenue through which deleted content can be leaked. However, this is not the case. The only aspect that must be considered while recording any such information is making it non-readable to the adversary (e.g. by using encryption) as in [67].

Future Work. In section II we laid down the requirement that for *un-traceable deletion* all application logic should be in possession of the database. Ficklebase requires all application logic to be written as database queries. Another direction via which this can be achieved is by utilizing stored procedures [15]. Under this scenario re-writing of SQL queries is insufficient, instead mechanisms are needed for execution of arbitrary procedural code in the context of versioning.

We note that although Ficklebase's approach of query re-writing makes it independent of the underlying DBMS, it limits its performance for scenarios where large number of versions need to be maintained. Hence, for greater performance we are looking in to (1) moving parts of Ficklebase functionality in to the DBMS by modifying the storage engine and (2) designing new data structures, that efficiently allow maintenance & expiration of versions.

Finally, it is important to continue to integrate Ficklebase with the work on history independent file systems to ensure cross-layer, end-to-end assurances.

Limitations. The current implementation does not provide support for re-writing user defined triggers and custom views. We plan to implement these features in the future.

[6] This can also be written as $\Gamma_{\mathcal{E}}^{V_i} = \Gamma_{\mathcal{S}} - T_j$ when $\mathcal{E}_x(V_i) > \mathcal{E}_x(t_k)$.

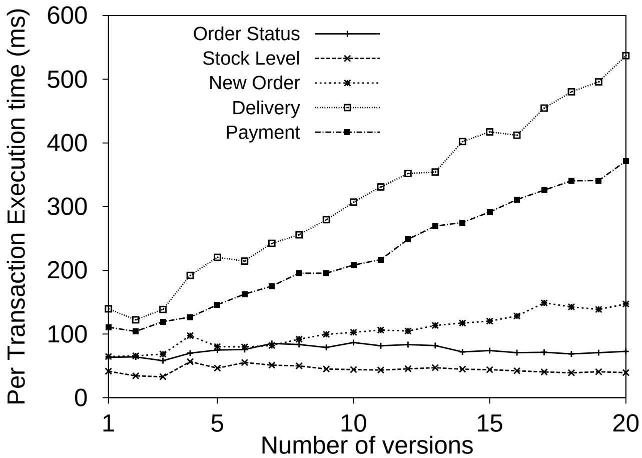


Fig. 6. Execution times for TPC-C transactions.

VII. Experiments

Benchmark. We evaluate the performance of Ficklebase using the TPC-C benchmark [19]. The benchmark data is set up with 16 warehouses giving a total database size (on disk) of 1.5 GB for each run. The database buffer pool size is 200MB. In the initial versioned database, tuples in relations *oorder*, *order line* and *new order* are given random expiration times. The tuples in other relations have fixed maximum expiration times. New tuples inserted during the benchmark transactions are also given random expiration times.

Setup. The database server runs on an Intel Xeon 3.4 GHz, 4GB RAM Linux box (kernel 2.6.18). The server DBMS is off-the-shelf MySQL version 14.12 Distrib 5.0.45. The client system is an Ubuntu VM running on an Intel core i5 at 1.60 GHz with 2 GB RAM. The Ficklebase proxy is implemented in Lua [11] and runs within the mysql proxy [12] component version 0.8.2. To simulate the TPC-C clients we use the BenchmarkSQL tool [1], modified so that all TPC-C logic is comprised in SQL queries.

Measurements. To measure the TPC-C transaction execution times we execute $n_i \times 50$ runs of each TPC-C transaction using a single client and record the average execution time (n_i is the target number of versions the test database instance is set up for). The multiplicative factor ($\times$ 50) ensures that targets of insert/update queries are distributed across all versions of the test database instance. Figure 6 shows the results for each of the TPC-C transactions with varying number of versions.

We observe the following overheads for each added version *New Order* (≈4.9 %), *Delivery* (≈7.8 %), *Payment* (≈6.7 %).

Version maintenance and query-rewriting (section IV) may initially give the impression that each added version in theory will result in an overhead of close to 1x i.e. if a transaction takes time t to complete execution on one version, then on two versions it would require $2t$ time, on three versions $3t$ and so on. This is because each client transaction is applied to all logical database versions. In practice however, this is not the case and actual overheads are far lower as seen above.

This is due database caching and co-location of tuple versions. Updates to individual tuples can cause distinct copies to be present in the database. However, these copies reside close together and are very often located in the same storage node (i.e. leaf node of the underlying B^+-tree). Hence queries applicable to a specific version often locate their target tuples in the database caches, where they were processed for previous versions.

In addition, re-writing of create statements further ensures this by adding the *VERSION* attribute only as the terminal field of primary keys or other indexes (Figure 4(a)). Thus even if tuples are valid in different versions (and differ in their *VERSION* attribute) they will not be dispersed within the storage indexes.

Stock level and *order status* are both read-only transactions. Note from section IV-C that read-only transactions are executed only on version one. Hence increasing number of versions to not contribute any overheads on these transactions.

VIII. Related Work

Secure Deletion. Solutions providing *Secure Deletion* employ either (1) Overwriting *or* (2) Encryption to erase deleted content from storage media.

Methods to recover erased data from magnetic storage were originally presented in [45] along with schemes to make this recovery significantly more difficult. In fact [45] suggests that it may be necessary to overwrite deleted content up to 35 times to completely ensure non-recovery.

Later [51, 72] claim that at least software-based data recovery can be made impossible by a single overwrite. [51] also provides extensions to the Ext3 file system that implement overwriting not just of deleted file content but also of file meta-data (e.g. name,owner,group,size etc).

[72] investigates the possibility of recovering deleted content utilizing an electron microscope concluding that although recovery of an individual bit is possible, the likelihood of recovering sizeable data using this technique is negligible.

An extension for the Ext2 file system was made available by [24]. Here an asynchronous overwriting mechanism is employed which causes less interference with user tasks but sacrifices security for a short interval (from deletion time to overwrite operation).

Many available off-the-shelf tools aid in *secure deletion* [5]. A survey of these sanitization & forensic analysis tools is provided in [39].

[33] addresses identification and removal of deleted content from main memory. The goal here is to reduce the lifetime of data in main memory (referred to as *secure deallocation*). On deallocation the solution overwrites the heap/stack content with zeros to prevent recovery.

[54] posit that overwriting is insufficient and instead employ encoding/decoding to protect sensitive data. AES encryption/decryption is used (within a modified firmware) to protect deleted content.

[56] designed a NAND flash file system based on YAFFS to support *Secure Deletion*. Encryption is used to delete files, while a single block is allocated for storage of all keys. The

key store block is erased using overwriting and once this is done all deleted(encrypted) content becomes un-recoverable.

Encryption is also employed in [74] to dispose of relevant index entries when a record expires. *Secure deletion* for a versioning file system is provided in [63]. Here, a special stub is stored with each encrypted data block. On deletion only the stub is overwritten which renders the associated block un-recoverable.

For a more detailed survey on *secure deletion* we refer the reader to [37].

History Independence. Both *secure deletion* and *history independent* data structures [48] are complimentary to Ficklebase since all three are essential & need to exist in tandem to achieve complete erasure of deleted content.

Initial work on *History Independence* focussed on hash tables [25, 26, 61] and is not directly applicable to relational databases (unless specific hash indices are used).

B-Treaps [42] and B-Skip-Lists [43] are promising alternatives for use in database storage engines. Both offer the same functions as a standard B-Tree and have the same depth O($\log_B n$), where B is the block transfer size. The only advantage of B-Skip-Lists over B-Treaps is their simplicity making them easier to implement.

A comprehensive survey and explanation of these *history independent* data structures is available via [41].

Compensating Transactions. A *compensating transaction* on execution undoes the effect of a previously committed transaction without resorting to cascading aborts. Hence, compensating transactions can potentially be utilized to undo the side effects of deleted tuples as in Ficklebase. However, Compensating Transactions are application-dependent [55], need to be pre-defined and can only be minimally automated. Ficklebase on the other provides support for *un-traceable deletion* at the database level.

Guidelines for designing compensating transactions are discussed in [55]. [34] uses an example of an online bookshop transaction to review several notations for compensation including their syntax and semantics.

Sagas [28] is a flow composition language which achieves atomicity based on compensation. In case of a long running transactions that fails to complete, compensation is employed to undo its effects. [28] also addresses parallel composition, nesting and exception handling. In addition, composition languages such as BPEL4WS [2] enable programmers to specify compensations for associated transactions.

Multiversion Databases. On the flip side of deletion is the requirement to record every single change made to data. This may be required for historical queries or to document system evolution. Research on *multiversion databases* achieves this by designing data structures that are efficient for both storing & retrieving versioned data. Here, no information is ever deleted but is rather made available for later querying by version or by time.

Designed data structures range from basic B-Trees [57] to transactional B^+-Trees [46] with concurrency support. In addition [49, 50] address branched evolution while [58] enables creation of views on multi-versioned data.

A summary of various multi-versioned data structures is available in [47]. Commercial [13] and open source implementations [16, 18] are also available.

Statistical Databases. *Statistical databases* [27] are used for maintaining statistics over data in an OLAP (online analytical processing) model. The main security concern here is to prevent an adversary from deducing very specific information by issuing statistical queries. Typical approaches to prevent such leakage include (but are not limited to) – only supporting aggregate queries, refusal to answer queries with small result sets, returning ranges instead of specific values etc [31, 36]. At first glance it may seem that *statistical databases* achieve *un-traceable deletion* at least for aggregates. E.g. if a data item is deleted then all aggregates will be updated to remove its effects. However, such databases are designed for the OLAP model and are not intended for data modification operations such as deletion.

Forensic Analysis. *Forensic analysis* [44], related research [53, 60, 71] and available tools [6, 14] serve to enable detection/prevention of tampering of system data. Several forensic algorithms are discussed in [62]. In some cases *forensic analysis* can be complimentary to Ficklebase, e.g. [67] makes audit logs unreadable by the adversary thereby closing another avenue of a possible leakage of deleted data items.

Data Degradation. Data Degradation [22] is a work-in-progress to address removal of sensitive data. Here the goal is to gradually degrade sensitive information over time eventually making it un-recoverable. Although comprehensive techniques are yet to be designed [21] gives a simple introductory solution. Here, a data item is degraded in steps from specific to more general values. E.g. an address field may initially contain the entire detailed address. In the next iteration the street part is removed, a following iteration removes the state & zip leaving only the country code and so on.

Information Flow Control. Although not dealing with removal of side-effects *information flow control* and related implementations [35, 66, 73] enable tracking of sensitive data across system components. This can be used along with Ficklebase to detect and later delete copies of data items that have crossed system boundaries (e.g. moved to another node on a distributed system).

Other. [23] enables application developers to specify destructive policies on business records. These policies are stored and later executed as stored procedures. The execution is triggered by additional policies that define a critical view which comprise of sensitive data. The destructive policies here need to be predefined not unlike compensating transactions.

IX. Conclusion

In this paper we introduced *un-traceable deletion* which along with *secure deletion* and *history independence* is integral in ensuring complete erasure of deleted content.

We provide insights into the new functional aspects of this new assurance in the context of databases and present the design and evaluation of Ficklebase, a relational database which achieves *un-traceable deletion* via versioning and query-rewriting.

REFERENCES

[1] BenchmarkSQL. Online at http://sourceforge.net/projects/benchmarksql/.

[2] Business Process Execution Language for Web Services. Online at http://www.ibm.com/developerworks/library/specification/ws-bpel/.

[3] Computer Forensics. Online at http://en.wikipedia.org/wiki/Computer_forensics.

[4] Current Population Survey (CPA). Online at http://www.bls.gov/cps/.

[5] Deleted but not gone. Online at http://www.nytimes.com/2005/11/03/technology/circuits/03basics.html?pagewanted=print.

[6] EnCase. Online at http://www.guidancesoftware.com/forensic.htm.

[7] EU's Data Protection Directive. Online at http://ec.europa.eu/justice/data-protection/index_en.htm.

[8] Family Educational Rights and Privacy Act (FERPA). Online at http://www2.ed.gov/policy/gen/guid/fpco/ferpa/index.html.

[9] Federal Information Security Management Act (FISMA). Online at http://csrc.nist.gov/groups/SMA/fisma/index.html.

[10] Gramm–Leach–Bliley Act (GLB). Online at http://en.wikipedia.org/wiki/Gramm-Leach-Bliley_Act.

[11] Lua programming language. Online at http://www.lua.org/.

[12] MySQL Proxy. Online at http://forge.mysql.com/wiki/MySQL_Proxy.

[13] Oracle Total Recall. Online at http://www.oracle.com/us/products/database/options/total-recall/overview/index.html.

[14] Sleuth Kit. Online at http://www.sleuthkit.org/.

[15] Stored Procedure. Online at http://en.wikipedia.org/wiki/Stored_procedure.

[16] Tau BerkelyDB. Online at http://www.cs.arizona.edu/projects/tau/tbdb/.

[17] The Health Insurance Portability and Accountability Act of 1996 (HIPAA). Online at http://www.hhs.gov/ocr/privacy/.

[18] TimeDB. Online at http://www.timeconsult.com/.

[19] TPC-C Benchmark. Online at http://www.tpc.org/tpcc/default.asp.

[20] UK Data Protection Act 1998 (DPA). Online at http://en.wikipedia.org/wiki/Data_Protection_Act_1998#Data_protection_principles.

[21] Nicolas Anciaux, Luc Bouganim, Harold van Heerde, Philippe Pucheral, and Peter M. G. Apers. Instantdb: Enforcing timely degradation of sensitive data. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 1373–1375, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Nicolas Anciaux, Luc Bouganim, Harold van Heerde, Philippe Pucheral, and Peter M.G. Apers. Data degradation: making private data less sensitive over time. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 1401–1402, New York, NY, USA, 2008. ACM.

[23] Ahmed A. Ataullah, Ashraf Aboulnaga, and Frank Wm. Tompa. Records retention in relational database systems. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 873–882, New York, NY, USA, 2008. ACM.

[24] Steven Bauer and Nissanka B. Priyantha. Secure data deletion for linux file systems. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 12–12, Berkeley, CA, USA, 2001. USENIX Association.

[25] Guy E. Blelloch. Strongly history-independent hashing with applications. In *In Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, pages 272–282, 2007.

[26] Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '07, pages 272–282, Washington, DC, USA, 2007. IEEE Computer Society.

[27] Claus Boyens, Oliver Gnther, and Hans j. Lenz. Statistical databases.

[28] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. *SIGPLAN Not.*, 40(1):209–220, January 2005.

[29] Simon Byers. Scalable exploitation of, and responses to information leakage through hidden data in published documents. *ATT Research*, 2003.

[30] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley Professional, 2005.

[31] Francis Y. Chin. Security in statistical databases for queries with small counts. *ACM Trans. Database Syst.*, 3(1):92–104, March 1978.

[32] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[33] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 22–22, Berkeley, CA, USA, 2005. USENIX Association.

[34] Christian Colombo and Gordon J. Pace. A compensating transaction example in twelve notations. Technical Report CS2011-01, Department of Computer Science, University of Malta, 2011. Available from http://www.um.edu.mt/ict/cs/research/technical_reports.

[35] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. *SIGARCH Comput. Archit. News*, 35(2):482–493, June 2007.

[36] Dorothy E. Denning and Jan Schlörer. A fast procedure for finding a tracker in a statistical database. *ACM Trans. Database Syst.*, 5(1):88–102, March 1980.

[37] Sarah M. Diesburg and An-I Andy Wang. A survey of confidential data storage and deletion methods. *ACM Comput. Surv.*, 43(1):2:1–2:37, December 2010.

[38] Kevvie Fowler. *SQL Server Forensic Analysis*. Addison-Wesley Professional, 2008.

[39] Simson L. Garfinkel and Abhi Shelat. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security and Privacy*, 1(1):17–27, January 2003.

[40] Tal Garfinkel, Ben Pfaff, Jim Chow, and Mendel Rosenblum. Data lifetime is a systems problem. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, New York, NY, USA, 2004. ACM.

[41] Daniel Golovin. *Uniquely represented data structures with applications to privacy*. PhD thesis, Pittsburgh, PA, USA, 2008. AAI3340637.

[42] Daniel Golovin. B-treaps: A uniquely represented alternative to b-trees. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I*, ICALP '09, pages 487–499, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] Daniel Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. *CoRR*, abs/1005.0662, 2010.

[44] Mario A. M. Guimaraes, Richard Austin, and Huwida Said. Database forensics. In *2010 Information Security Curriculum Development Conference*, InfoSecCD '10, pages 62–65, New York, NY, USA, 2010. ACM.

[45] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, pages 8–8, Berkeley, CA, USA, 1996. USENIX Association.

[46] Tuukka Haapasalo, Ibrahim Jaluta, Bernhard Seeger, Seppo Sippu, and Eljas Soisalon-Soininen. Transactions on the multiversion b+-tree. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 1064–1075, New York, NY, USA, 2009. ACM.

[47] Tuukka K. Haapasalo, Ibrahim M. Jaluta, Seppo S. Sippu, and Eljas O. Soisalon-Soininen. Concurrency control and recovery for multiversion database structures. In *Proceedings of the 2nd PhD workshop on Information and knowledge management*, PIKM '08, pages 73–80, New York, NY, USA, 2008. ACM.

[48] Jason D. Hartline, Edwin S. Hong, Alexander E. Mohr, William R. Pentney, and Emily Rocke. Characterizing history independent data structures. In *Proceedings of the 13th International Symposium on Algorithms and Computation*, ISAAC '02, pages 229–240, London, UK, UK, 2002. Springer-Verlag.

[49] Linan Jiang, Betty Salzberg, David B. Lomet, and Manuel Barrena García. The bt-tree: A branched and temporal access method. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 451–460, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[50] Khaled Jouini and Geneviève Jomier. Indexing multiversion databases. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 915–918, New York, NY, USA, 2007. ACM.

[51] Nikolai Joukov, Harry Papaxenopoulos, and Erez Zadok. Secure deletion myths, issues, and solutions. In *Proceedings of the second ACM workshop on Storage security and survivability*, StorageSS '06, pages 61–66, New York, NY, USA, 2006. ACM.

[52] J. Katz and Y. Lindell. *Introduction to modern cryptography*. Chapman & Hall/CRC cryptography and network security. Chapman & Hall/CRC, 2008.

[53] Peter Kieseberg, Sebastian Schrittwieser, Martin Mulazzani, Markus Huber, and Edgar Weippl. Trees cannot lie: Using data structures for forensics purposes. In *Proceedings of the 2011 European Intelligence and Security Informatics Conference*, EISIC '11, pages 282–285, Washington, DC, USA, 2011. IEEE Computer Society.

[54] Marek Klonowski, MichałPrzykucki, and Tomasz Strumiński. Information security applications. chapter Data Deletion with Provable Security, pages 240–255. Springer-Verlag, Berlin, Heidelberg, 2009.

[55] Henry F. Korth, Eliezer Levy, and Avi Silberschatz. A formal approach to recovery by compensating transactions. Technical report, Austin, TX, USA, 1990.

[56] Jaeheung Lee, Junyoung Heo, Yookun Cho, Jiman Hong, and Sung Y. Shin. Secure deletion for nand flash file system. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1710–1714, New York, NY, USA, 2008. ACM.

[57] David Lomet and Betty Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 315–324, New York, NY, USA, 1989. ACM.

[58] Claudia Bauzer Medeiros, Marie-Jo Bellosta, and Geneviève Jomier. Multiversion views: constructing views in a multiversion database. *Data Knowl. Eng.*, 33(3):277–306, June 2000.

[59] Daniele Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 456–464, New York, NY, USA, 1997. ACM.

[60] Soumyadeb Mitra, Marianne Winslett, Richard T. Snodgrass, Shashank Yaduvanshi, and Sumedh Ambokar. An architecture for regulatory compliant database management. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 162–173, Washington, DC, USA, 2009. IEEE Computer Society.

[61] Moni Naor, Gil Segev, and Udi Wieder. History-independent cuckoo hashing. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 631–642, Berlin, Heidelberg, 2008. Springer-Verlag.

[62] Kyriacos E. Pavlou and Richard T. Snodgrass. Forensic analysis of database tampering. *ACM Trans. Database Syst.*, 33(4):30:1–30:47, December 2008.

[63] Zachary N. J. Peterson, Randal Burns, Joe Herring, Adam Stubblefield, and Aviel D. Rubin. Secure deletion for a versioning file system. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association.

[64] R. F. Resende and A. El Abbadi. On the serializability theorem for nested transactions. *Inf. Process. Lett.*, 50(4):177–183, May 1994.

[65] James M. Rosenbaum. In defence of the delete key. *The Green Bag*, 3(4), 2000.

[66] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: practical fine-grained decentralized information flow control. *SIGPLAN Not.*, 44(6):63–74, June 2009.

[67] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2):159–176, May 1999.

[68] Jitesh Shetty and Jafar Adibi. The enron email dataset database schema and brief statistical report, 2004.

[69] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 91–102, New York, NY, USA, 2007. ACM.

[70] Latanya Sweeney. Protecting job seekers from identity theft. *IEEE Internet Computing*, 10(2):74–78, March 2006.

[71] Maolin Tang and Colin Fidge. Reconstruction of falsified computer logs for digital forensics investigations. In *Proceedings of the Eighth Australasian Conference on Information Security - Volume 105*, AISC '10, pages 12–21, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.

[72] Craig Wright, Dave Kleiman, and Shyaam Sundhar R.S. Overwriting hard drive data: The great wiping controversy. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 243–257, Berlin, Heidelberg, 2008. Springer-Verlag.

[73] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. *Commun. ACM*, 54(11):93–101, November 2011.

[74] Qingbo Zhu and Windsor W. Hsu. Fossilized index: the linchpin of trustworthy non-alterable electronic records. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 395–406, New York, NY, USA, 2005. ACM.