# An Architecture for Regulatory Compliant Database Management

Soumyadeb Mitra [#1], Marianne Winslett [#2], Richard T. Snodgrass [+3], Shashank Yaduvanshi [*4], Sumedh Ambokar [*4]

[#]*Department of Computer Science, University of Illinois at Urbana-Champaign*
[1]mitra1@cs.uiuc.edu [2]winslett@cs.uiuc.edu

[+]*Department of Computer Science, University of Arizona*
[3]rts@cs.arizona.edu

[*]*Department of Computer Science, Indian Institute of Technology*
[3]hyaduvanshi@gmail.com, sumedh@cse.iitb.ac.in

***Abstract*— Spurred by financial scandals and privacy concerns, governments worldwide have moved to ensure confidence in digital records by regulating their retention and deletion. These requirements have led to a huge market for *compliance storage servers*, which ensure that data are not shredded or altered before the end of their mandatory retention period. These servers preserve unstructured and semi-structured data at a file-level granularity: email, spreadsheets, reports, instant messages. In this paper, we extend this level of protection to structured data residing in relational databases. We propose a compliant DBMS architecture and two refinements that illustrate the additional security that one can gain with only a slight performance penalty, with almost no modifications to the DBMS kernel. We evaluate our proposed architecture through experiments with TPC-C on a high-performance DBMS, and show that the runtime overhead for transaction processing is approximately 10% in typical configurations.**

## I. Introduction

Recent regulations require many corporations to ensure trustworthy long-term retention of their routine business documents. The US alone has over 10,000 regulations that govern how business data should be managed, many of which focus on ensuring that business documents are trustworthy during their mandated multi-year retention periods [9]. Compliance is assessed by periodic external audits, with recent fines exceeding a million dollars for each non-complying Wall Street firm [34]. For example, Securities and Exchange Commission Rule 17a-4 [33] and the Sarbanes Oxley Act (SOX) [32] specify integrity requirements for financial data and retention periods of 3 years and 7 years, respectively. Other regulations that impose lengthy data retention and trustworthiness requirements include the Gramm-Leach-Bliley Act, Food and Drug Administration regulations [31], the Health Insurance Portability and Accountability Act (HIPAA) [30], and the Department of Defense Records Management Program under directive 5015.2.

In response to these laws and directives, a huge market has developed for compliance storage servers sold by IBM, EMC, Network Appliance, HP, Sony, and other vendors. For example, all medium and large financial firms use compliance storage servers to help them comply with SOX and Rule 17a-4. These storage servers provide an approximation of write-once read-many (WORM) storage to ensure that files committed to the WORM server are read-only during their retention periods (*term-immutable*), and cannot be deleted or altered during that term even by a company system administrator or a hacker with administrative privileges. All these products work at the granularity of files, and their security guarantees focus on providing term-immutability for files. But much corporate data is stored and managed at a much finer granularity, that of relational tuples. If we can ensure term-immutability for tuples at reasonable cost, society can benefit from the resulting improvements in accountability and reduction in fraud—benefits already claimed for SOX and Rule 17a-4 in the case of larger documents [8].

The simplest way to make tuples term-immutable is to leverage the file level interface of existing WORM storage systems, but the obvious approaches are impractical. For example, storing each tuple and all its subsequent versions in a separate file would hobble performance, due to the high cost of opening and closing files. Storing the entire database in one file on WORM media would require committing a new copy of the database file after every transaction, thus imposing severe space and time overheads. Further, an adversary could tamper with the database file during copying.

In this paper, we first review the accepted threat model for WORM storage, emphasizing two parameters that are central to term-immutable databases: a regret interval and a query verification interval. We then propose a term-immutable DBMS architecture along with two refinements that reflect different tradeoffs between security and efficiency. Their security assurances extend from just ensuring the integrity of the current DBMS snapshot to guaranteeing the correct execution of all past transactions. Higher security assurances incur more cost, both in terms of space requirements on WORM media and transaction runtime overheads. Database crashes and aborted transactions present an opportunity for the adversary to tamper with the database, and we show how to modify crash recovery and auditing to identify these situations and handle them correctly. We show that the initial architecture

and both refinements are competitive in terms of space and runtime performance. We finish by presenting algorithms for shredding expired tuples.

## II. Threat Model

An auditor must be able to differentiate data tampering from legitimate modifications. In the retention scenarios targeted by SOX and Rule 17a-4, tampering occurs after a tuple has been initially correctly stored, so tampering takes the form of changes to previously stored data. To clarify this threat, it is helpful to think in terms of transaction-time databases, which record the entire history of their data [13].

One realization of a transaction-time relation augments the attributes declared in the `CREATE TABLE` statement with an additional *start time* attribute maintained by the DBMS and not normally seen by the application. (This approach has been called a *tracking log* [23]; another prominent realization uses another attribute, the *stop time*.) The start time attribute holds the time at which a tuple was inserted, that is, the commit time of the transaction that created the tuple [18], [26]. When a tuple is updated, the old copy of the tuple (and its start time) is left intact, and a new version of the tuple with a new start time is inserted. Deletions are handled by inserting a special *end-of-life* tuple whose start time is the time that deletion took place. In this way, the entire version history of every tuple is maintained in the database.

Transaction-time databases support temporal SQL queries, where time can be specified as an additional argument. When no time parameter is given, the query is evaluated on the latest versions of all tuples that have not reached the end of their lives; to make this more efficient, the different versions of a tuple $t$ can be threaded together on the page where they reside, so that the slot pointer for $t$ leads to the most recent version of $t$ [16]. In this paper we use a transaction-time DBMS.

We make several trust assumptions that are appropriate for the fraud scenarios targeted by SOX and Rule 17a-4. We trust that the WORM server operates properly. In particular, we trust that it records the metadata and data of files correctly, and never overwrites a file during its retention period. Regulators also make this trust assumption. We assume the server allows us to append to files, so that it can hold logs. We trust the DBMS software and surrounding environment used to execute transactions, and we trust that transactions commit properly; standard techniques [6], [28] from the security community can be employed to help ensure that this is so. We also trust the querying process. For example, a prosecutor can have a company's disks removed and brought to her office for querying and analysis using her own DBMS software. Or an auditor can use signature-based techniques that are well-understood in the security community to ensure that he is running an authentic copy of the DBMS, OS, and supporting software. Finally, we trust that the adversary cannot tamper with data while it resides in the DBMS buffer cache. Buffer cache attacks can be prevented by kernel patches that keep processes, even those owned by root, from getting read-write access to other processes' memory [35]. The only way to bypass such patches is to replace the kernel and reboot the DBMS machine, which is hard to carry out without being detected at the next compliance audit.

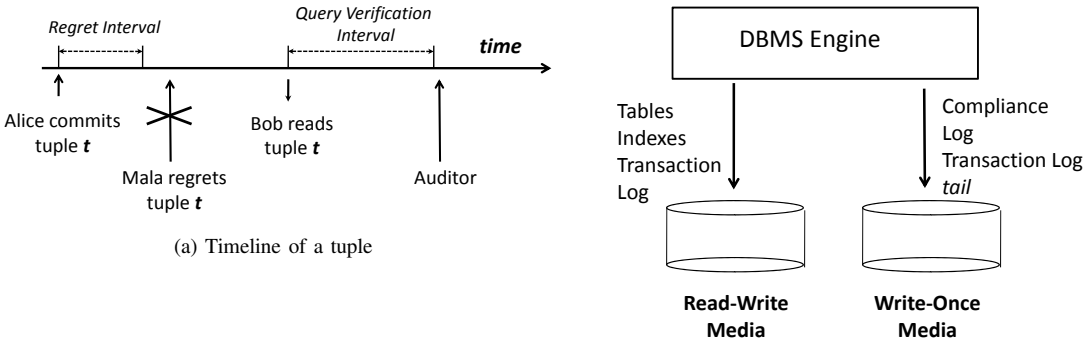
For compliance with SOX and Rule 17a-4, the key threat to tuple term-immutability that we must address is *undetectable alteration or shredding of existing unexpired committed tuples at the behest of company insiders* who wish to retroactively hide activities recorded in the organization's DBMS. (Under current regulatory interpretation, detectable tampering leads to presumption of guilt, and is punishable by stiff fines and prison sentences.) For example, a CEO may want to hide illegal asset shuffling recorded in the company's financial database.

A secondary threat that we wish to address is the insertion of tuples with start times that have already passed, in an attempt to make it appear that an activity took place though in fact it did not. SOX and Rule 17a-4 do not consider this a key threat, but we find it compelling for tampering scenarios involving post-hoc insertion of government electronic records, such as records of births, deaths, marriages, property transfers, drivers' licenses, voter registrations, and so on.

The third threat that we must address is the possibility that evidence of a tuple's existence persists after its retention period is over and it has been shredded. In some domains, regulations require the shredding of expired records. For example, Section 42.1-82 of the *Code of Virginia* requires shredding of records containing social security numbers. Evidence of shredded tuples and the presence of expired but unshredded tuples and their metadata are corporate liabilities: such evidence can be subpoenaed and used against the company. Further, the evidence cannot be destroyed once it has been subpoenaed.

As shown in Figure 1(a), the first tampering threat arises during the interval between creation and querying, when an adversary may reach a point of regret and wish to alter or remove previously-stored data to cover up her misdeeds. Cover-up attempts are often directed by high-level insiders (e.g., the CEOs and CFOs of Enron, Global Crossing, Tyco, and Adelphi). System administrators are vulnerable to pressure and bribes from high-level insiders. Thus an attacker might have or assume the identity of any legitimate user or superuser in the system, and perform any action that person can perform. More precisely, when an adversary Mala starts to regret the existence of a tuple, she may take over root on the platform where the DBMS runs and issue any possible command to the WORM server in an attempt to modify one or more historical versions of that tuple in the database. Mala can target any database file, including data, indexes, logs, and metadata. The security aspects of a term-immutable DB can be captured by the following parameters.

- **Regret Interval**: the minimum time interval we can assume between when a tuple is committed and when an adversary tries to tamper with it. In current legal interpretations of email SEC compliance [33], that interval is zero. However, for financial records under SOX compliance, we can assume an interval of, say, 5 minutes or more before anyone is likely to regret the presence of a new tuple.




(a) Timeline of a tuple

(b) Log-consistent DBMS architecture

Fig. 1. Threat model parameters and term-immutable DBMS architecture. (a) The *regret interval* is the minimum time between when a tuple is committed and when it is tampered. The *query validation interval* is the time gap between querying and validation of the query results. (b) Log-compliant architecture: the compliance log, the tail of the transaction log, and a snapshot of the database state at the previous audit are stored on WORM. The database itself and the transaction log reside on ordinary read/write media.

- **Query Verification Interval**: the interval between the time $t_1$ when a user issues a transaction and the time $t_2$ when the user finds out that the database was *compliant* at $t_1$, i.e., had not been tampered.

## III. Related Work

Database security has received significant research attention in the context of outsourced databases, specifically, ensuring data privacy and query correctness when the database is stored on an untrusted machine [11], [12], [20]. These techniques typically rely on a trusted data owner who encrypts and signs the data and index entries appropriately, much like proposals for cryptographic file systems [2], [4], [10], [14], [19]. All these techniques require the original owner of the data to be trustworthy, while in our setting the data owner is a potential adversary who may have strong incentives to alter and re-sign data and indexes. More generally, a DBA or system administrator can take on the data owner's identity, allowing her to decrypt and modify data, files, and indexes and re-sign them. Also, unlike the outsourcing threat model, we trust the WORM server not to shred committed tuples before they expire.

Ahmed et al. address the problem of writing and enforcing tuple retention and shredding policies, expressed as restrictions on the operations that can be performed on views [1]. They trust the DBMS to enforce the policies. We address the complementary problems of ensuring correct query answers and securing DB tuples against tampering, even by superusers who edit the DB file directly. Our techniques can be used to ensure that their policy enforcement queries have been executed properly.

The other DB security work most relevant to this paper is on tamper-proof audit logs [24]. In this work, the authors have the DBMS hash transactional data with cryptographically strong one-way hash functions. This hash is periodically signed by a trusted external digital notary, then stored in the DB. A separate validator checks the database state against these signed hashes to detect any compromise. If tampering is detected, a separate forensic analyzer uses other hashes computed during previous validation runs, to pinpoint when the tampering occurred and roughly where in the database the data was tampered [17]. The use of a notary prevents an adversary, even an auditor or a buggy DBMS, from silently corrupting the database. The regret interval of this approach is the interval between consecutive notarization events. Since notarization is done by a third party, it is challenging to ensure a small regret interval. Furthermore, the integrity of the state of the database is only verified at validation time and so does not guarantee the correctness of queries that ran between validation points. If an adversary tampers with the database immediately after notarization, undoes the tampering before the next notarization, and repeats the process, the attack will never be detected, even though all queries will see tampered data.

Researchers have also proposed techniques for protecting critical DBMS structures against errors [5], [29]. These techniques mainly deal with abnormal corruption caused by software errors, not deliberate attacks. An adversary can bypass these checks by using a file editor on the database file.

## IV. The Log-Consistent DB Architecture

Our *log-consistent compliant database architecture* and its refinements store the database state (including tuples, indexes, and transaction log) on conventional read-write media, and store a special log of all data changes on WORM. As is usual for a transaction-time database, all data modification operations (`INSERT`, `UPDATE`, and even `DELETE`) create a new physical version of the tuple(s) that they affect. The *compliance log* $\mathcal{L}$ on WORM contains all such new tuples added to the database since the last audit. The compliance log is separate from the usual DBMS logs, and its NEW_TUPLE log records are sent to the WORM server after each transaction commits. (In a later section, we discuss synchronization requirements for the transaction log and the compliance log.)

In addition, the auditor places a complete snapshot of the current database state on WORM after every audit, together with the auditor's digital signature testifying that the snapshot is correct. This snapshot plus the log lets the next auditor verify that the new database state is compliant, as described below.[1] We also require the tail (the last two regret intervals) of the DBMS's transaction log to be kept on WORM. An untrusted process can migrate the transaction log tail to ordinary storage in the background. The tail of the transaction log at the time of a crash must be preserved on WORM until the end of the next audit; if no crash occurs, the WORM copy of the transaction log can be deleted once two regret intervals have passed.

The log-consistent architecture is space-efficient because each snapshot can expire and be deleted from WORM once the next snapshot is in place, if desired. Similarly, the compliance log file can be deleted after every audit. The current tail of the transaction log must be kept on WORM, plus the tails that were active at the time of any crashes; but all of these can be deleted after the next audit.

The auditor must check the signature on the previous snapshot, and then check that all the tuples in the snapshot, plus all tuples in NEW_TUPLE records in $\mathcal{L}$, are present in the current database state. (We defer consideration of shredded tuples to Section VIII. Until a tuple is shredded, it is still visible to temporal queries, even if it has been deleted.) Let $D_f$ be the set of tuples in the database state, which we will refer to as the *final state*. All the tuples in the previous snapshot state $D_s$ on WORM and in the compliance log $\mathcal{L}$ on WORM must be included in the final state. Slightly abusing notation, we can write this as:

*Tuple Completeness Condition*: $D_f = D_s \cup \mathcal{L}$ .

As shown in Section IV-C, tuple completeness is necessary but not sufficient: the auditor must also verify all indexes and the logical ordering of the versions of each tuple. Before getting into these details, we first discuss the manner in which the compliance log is written, and propose techniques for verifying tuple completeness.

For good performance, we do not want to make transactions wait to commit until their $\mathcal{L}$ records have reached the WORM server. Fortunately, we expect that in almost all compliance scenarios, the regret interval will be on the order of minutes. As shown in Section VII, this gives us plenty of time to prepare the transaction's $\mathcal{L}$ entries and flush them to the WORM server. If at any point we are unable to write to $\mathcal{L}$, transaction processing must halt until the problem is fixed. For this reason, it is advisable to use standard availability and fail-over techniques to ensure that $\mathcal{L}$ can always be written to WORM; we do not consider this issue further in this paper.

The discussion above assumes that the database is quiescent during audit. The simplest version of a non-quiescent audit begins by not admitting any new transactions, and waiting for the current ones to finish and their dirty pages to reach disk. For performance reasons, a transaction-time DBMS often uses the transaction ID as a temporary commit time value in a tuple, and does a lazy update of the commit time later [16], [18]. The audit must wait for these lazy updates to reach disk as well. Once this is done, the current file for $\mathcal{L}$ is permanently closed, a new one is opened, and new transactions are admitted as usual. When the auditor examines the database state or writes the next snapshot, the auditor will ignore all tuples whose commit time is later than that of the last of these transactions to finish. Since audits are rare events—perhaps once a year—we expect that a brief pause to clear out active transactions will be acceptable. If not, the pause can be further reduced, using techniques not described here due to space limits.

The preceding discussion did not consider the effect of schema changes. All metadata changes that do not have an impact on existing tuple storage—such as adding or dropping a relation or index, updating the statistics kept for a relation, or changing a permission—are handled just like any ordinary tuple insertion, deletion, or update. (Of course, dropping a relation or index does not cause it to actually go away; its tuples/nodes and the corresponding metadata will be kept until they expire, just like any other data.) Metadata changes whose implementation involves eager updates to stored tuples can also be handled in this manner (e.g., eagerly adding a new attribute with a computed value to each tuple of a relation, or eagerly dropping an attribute). Lazy implementations of such metadata changes are quite popular in commercial DBMSs; to support them, either the next audit must wait until all of these lazy updates have completed, or else we must use additional care in constructing hash functions (to be discussed in the next section) to ensure that the differences between the logical state of the database and its actual physical state on disk do not cause the audit to fail.

### A. Verifying Tuple Completeness

The simplest technique for verifying that the final database state $D_f$ contains exactly the right tuples is to examine each tuple in $D_f$ and look up that tuple in the snapshot state $D_s$ and the log $\mathcal{L}$. Assuming that both the snapshot and the final state are sorted on the relation name and primary key (as is the case when the tuples are maintained in B$^+$-trees), this check can be done by (i) sorting $\mathcal{L}$ on relation name plus primary key, which takes $O(|\mathcal{L}|\log(|\mathcal{L}|))$ operations; (ii) merging the result with $D_s$, which takes $O(|D_s| + |\mathcal{L}|)$ operations; and (iii) comparing the result with $D_f$, which takes $O(|D_s| + |\mathcal{L}| + |D_f|)$ operations. Including the cost to write out the new snapshot, we have a total completeness check time of $O(|\mathcal{L}|\log(|\mathcal{L}|) + |D_s| + |D_f|)$. The merge and comparison steps can be executed concurrently in a single pass over the database states.

Since audits may be infrequent, $\mathcal{L}$ may be extremely long and sorting it can be costly. We avoid sorting entirely by using a cryptographically strong incremental hash function for sets that has the following properties.

- **Input**: The hash function $H$ operates on a set $\{a_1, \ldots, a_n\}$.

[1] We omit correctness arguments and algorithm pseudocode in this paper due to space limitations.

- **Incremental**: Given $a_n$ and $H(\{a_1, \ldots, a_{n-1}\})$, one can efficiently compute $H(\{a_1, \ldots, a_n\})$.
- **Commutative**: The value of $H$ is independent of the order of the items in the set.
- **Cryptographically Secure** (Pre-image Resistant): Given a set $\{a_1, \ldots, a_n\}$, one cannot efficiently find $\{b_1, \ldots, b_m\}$ $(\neq \{a_1, \ldots, a_n\})$ such that $H(\{a_1, \ldots, a_n\}) = H(\{b_1, \ldots, b_m\})$.

In our implementation, we used the ADD_HASH function proposed by Bellare and Micciancio [3]:

$$ADD\_HASH(a_1, \ \ldots, \ a_n) = \sum_{1 \leq i \leq n} h(a_1),$$

where $h$ is a big (512 bits or more) secure one-way hash function and the sum is taken modulo a large number.

Using such a hash function, the auditor can incrementally compute a hash over $D_s \cup \mathcal{L}$ and $D_f$. Pre-image resistance ensures that $H(D_s \cup \mathcal{L}) = H(D_f)$ iff $D_s \cup \mathcal{L} = D_f$. Each hash operation takes $O(1)$ time, and the completeness check now requires just a single pass over $D_f$, $\mathcal{L}$, and $D_s$, plus the creation of the new snapshot, which need not be sorted. Thus the total cost of the tuple completeness check is $O(|D_f| + |\mathcal{L}| + |D_s|)$. The completeness check cost can be further reduced by storing $H(D_f \cup \mathcal{L})$ on WORM at the end of each audit (together with the auditor's digital signature over the hashes), and using the stored value instead of computing $H(D_s)$ during the next audit. In this situation, we do not need to store the snapshot on WORM, which reduces auditing costs to $O(|D_f| + |\mathcal{L}|)$. However, we may wish to store the snapshot anyway, as it enables fine-grained forensic analysis if the next audit finds evidence of tampering.

Lazy timestamping may cause a NEW_TUPLE entry in $\mathcal{L}$ to contain a transaction ID rather than a commit time. After a transaction commits, we require the compliance logger to append a STAMP_TRANS record to $\mathcal{L}$, recording the transaction's ID and commit time. During its pass over $\mathcal{L}$ to check tuple completeness, before hashing a NEW_TUPLE record, the auditor must replace any transaction ID by the commit time from the appropriate STAMP_TRANS record. To make this process efficient, the compliance logger creates an auxiliary WORM log file listing the transaction ID and location in $\mathcal{L}$ of each STAMP_TRANS record. The auditor scans this log and creates an in-memory hash table mapping transaction IDs to their commit times. During the scan, the auditor must check the log's integrity, e.g., only one STAMP_TRANS record per transaction ID. The auditor must also check that the commit times in consecutive STAMP_TRANS records are separated by no more than one regret interval. This ensures that once the regret interval has passed, an adversary cannot edit the database state and append log entries to make it appear that a transaction took place that actually did not. If a regret interval is about to pass by without a transaction ending, the compliance logger must append a dummy STAMP_TRANS record to show that the system is still live.

In implementing this approach, our primary design goals were to maximize performance and minimize the changes to the underlying DBMS (Berkeley DB in this case [22]). To avoid changing DBMS internals, we wrote a compliance logging plugin that taps into the pread/pwrite system calls of Berkeley DB. When a page is written out with pwrite, this plugin parses the page, finds the tuples that are present in the buffer-cache page but not on the disk page, and logs them to $\mathcal{L}$ on WORM. To find these new tuples, we must fetch the disk page before overwriting it, incurring an additional storage server I/O. We reduce this cost by caching a separate copy of the page in available memory, or else on the local disk of the DBMS server, on each pread.

Before any adversary has time to regret the existence of a new tuple, its NEW_TUPLE record must reach WORM storage. Because NEW_TUPLE records are not generated until a pwrite call is made, this means that pwrite must be called for a dirty database page within one regret interval (e.g., 5-30 minutes) from the time it commits. We enforce this by marking all dirty pages once every regret interval, after calling pwrite on all dirty pages that were marked during the previous cycle. As part of the marking procedure, we create a new empty file on WORM that will stand as a witness that the DBMS was alive during this interval. These empty files can be deleted after the next audit.

### *B. Handling UNDO and REDO*

The preceding sections assume that all NEW_TUPLE records are for transactions that eventually commit. In practice, however, most commercial DBMSs allow the buffer manager to steal page frames from uncommitted transactions that may subsequently abort. When this happens, the DBMS will eventually read the page back in, undo the changes made by the aborted transaction, and write the page back out.

There are several alternatives for how to handle aborted transactions, and the choice also has serious implications for the complexity of crash recovery and post-crash audits. We chose an approach that minimizes changes to the original DBMS. First, when a transaction fails, the compliance logger will output an ABORT record containing the transaction's ID. If the aborted transaction performed tuple writes that are later undone, the compliance logger will notice that a tuple version has been removed from the page; beyond the ABORT record for this transaction, the compliance logger does not need to append additional log records pertaining to the UNDO action.

During its pass through $\mathcal{L}$, the auditor includes in its hash chain only those NEW_TUPLE records having a corresponding STAMP_TRANS record. This extra task does not affect the computational complexity of the audit. The auditor can ignore duplicate ABORT or STAMP_TRANS records, but an ABORT and STAMP_TRANS record for the same transaction or two different STAMP_TRANS records for the same transaction indicate a tampering attempt; for example, Mala may append spurious ABORT records to $\mathcal{L}$ to try to hide the existence of tuples that she regrets. This is why the compliance logger must wait to write ABORT and STAMP_TRANS records until

the transaction has actually committed/aborted. Commit times that are not in strictly increasing order, or that occur during periods of supposed inactivity, also indicate tampering.

UNDO/REDO crash recovery also has an impact on compliance logging. To prevent major complications in crash recovery, the compliance logger must ensure that every tuple version on disk has a corresponding NEW_TUPLE record in $\mathcal{L}$. To do this, we require *all* data page writes to wait until their corresponding NEW_TUPLE and/or STAMP_TRANS records have reached the WORM server, both during crash recovery and during normal processing. This requirement does not delay transaction commits, though it does slightly delay the write-behind of data pages.

When the DBMS comes up after a crash, the compliance logger places a timestamped START_RECOVERY record on $\mathcal{L}$, because a crash can introduce long gaps in commit times that can cause the auditor's regret-interval time checks to fail. When checking these intervals for transactions affected by the crash, the auditor must increase the interval by the amount of time between the last entry on $\mathcal{L}$ before the crash and the timestamp of the START_RECOVERY record. This must be done carefully, since crash recovery represents an opportunity for the adversary; we omit the details here.

Next, the recovery process generates a list of all transactions that should be aborted or committed. The compliance logger appends the corresponding ABORT and STAMP_TRANS records to the compliance log, and flushes the compliance log to WORM. The remainder of recovery proceeds as usual.

After an UNDO action is taken during recovery and the corresponding data page is being written out, the compliance logger may detect that a tuple version on the page has been removed. In this case, the logger does not record any information on $\mathcal{L}$, because it already has logged an ABORT record for this transaction. REDO actions will also lead to page writes. If the page actually changes, the compliance logger will add the appropriate NEW_TUPLE record to $\mathcal{L}$. Recovery can cause $\mathcal{L}$ to contain duplicate NEW_TUPLE records; to prevent these duplicates from causing the audit to fail, the auditor uses a temporary hash table to identify duplicates and include them only once in its hash chain. A REDO that does not actually change a data page could result in a situation where $\mathcal{L}$ does not contain a NEW_TUPLE record for a page that was successfully changed on disk before the crash; as mentioned above, to avoid this problem, we require data page writes to wait until their corresponding NEW_TUPLE records have gone to WORM.

If the DBMS crashes within one regret interval after a transaction $\mathcal{T}$ commits, then pwrite may not have been called on some of $\mathcal{T}$'s dirty pages. Since the corresponding NEW_TUPLE records will not be on WORM, an adversary could alter the transaction log to remove $\mathcal{T}$'s updates before recovery. This is why we require the tail of the transaction log (the last two regret intervals) to be on WORM, and that it be retained until the next audit. The adversary could also try to hide the fact that the DBMS had crashed, get rid of the transaction log, and recover without $\mathcal{T}$'s updates. However, recall that we create one empty file on WORM per regret interval; the create time of this file shows that the DBMS was alive then. If the DBMS crashes and does not recover completely within one regret interval, then the empty file will not be written. This allows the auditor to detect crashes and look for the appropriate log tails. If an attacker crashes the DBMS, she will not be motivated to hide the crash by creating a new empty file at the right moment, because the tuples she wants to tamper will have been committed at least one regret interval ago, and therefore will have NEW_TUPLE records on WORM. She can, however, undetectably commit transactions that were active at the time of the crash. This is consistent with our threat model.

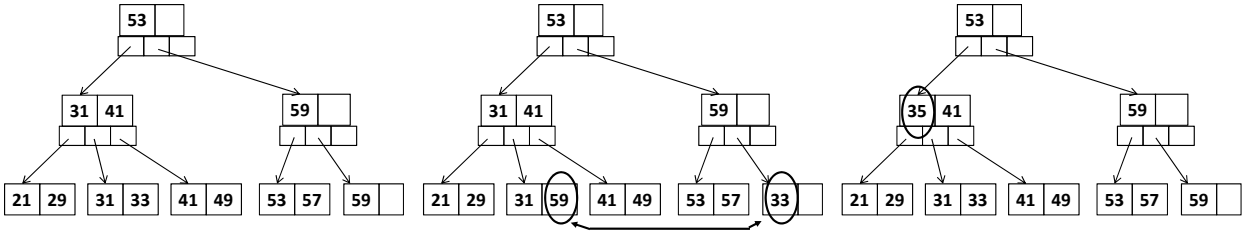
With these provisions, as long as recovery is completed before the next audit takes place, the audit will work properly. Since every recovered transaction $\mathcal{T}$ had committed, $\mathcal{T}$'s updates must be in $\mathcal{L}$ or the transaction log tail on WORM. The auditor must verify that the sequence of NEW_TUPLE and STAMP_TRANS records appended to $\mathcal{L}$ during recovery is consistent with the transaction log. We trust the WORM server to correctly record the create times of files, because it has special anti-tampering provisions in its clock functions (e.g., SnapLock from NetApp uses "Compliance Clock, a secure time mechanism"). Thus the auditor will notice if the transaction log has been replaced with a tampered copy, or an adversary tried to hide a crash.

### C. Verifying Data and Index Integrity

Of course, it is not enough for a page to contain the right tuples. The auditor must also check that the slot pointers on the page are set up correctly, the tuples are in sorted order across the pages (if that is required), the different versions of a tuple are all threaded together in commit-time order (if the DBMS uses version threading), and all other stored metadata is correct (the magic number on the page, the count of tuples on the page, etc.). We do not further consider these checks, though we note that most commercial DBMSs, including the one used for this project, have such an integrity checker.

In addition to checking tuple completeness and integrity, the auditor must verify that all indexes are correct and complete. If tuples are stored in a $B^+$-tree as in Figure 2(a), the attacker can logically hide a tuple by moving it from its correct position in the leaf node or by tampering with the nodes above it. Figure 2(b) shows the former attack, where leaf node elements 33 and 59 have been swapped. Figure 2(c) illustrates the latter attack, where index element 31 pointing to the leaf node containing $(31, 33)$ has been changed to 35. A lookup of 33 after either attack will fail.

The auditor checks for these corruptions by scanning the leaf nodes to verify that their keys are stored in increasing order and that all tuples are represented in the tree, and then verifying that the keys and pointers in internal nodes are consistent with the leaf nodes. This can be done by locating the minimal key in each node and comparing it to the corresponding key in its parent node. The integrity checker mentioned above can perform this check for every non-root




(a) Original B$^+$-tree

(b) B$^+$-tree with wrong element order

(c) B$^+$-tree with tampered key value

Fig. 2. Tampered B$^+$-trees.

node of the tree. Any inconsistency can be repaired and reported as a tampering attempt. The auditor can use similar techniques to check the integrity of other kinds of indexes, such as hash tables.

If Mala appends spurious entries to $\mathcal{L}$, then the audit will fail, even when the database state is correct. The audit will also fail if Mala adds spurious tuples to the database and does not record them in $\mathcal{L}$. Such acts of sabotage are outside the scope of compliance regulations such as SOX and Rule 17a-4. If Mala adds tuples *and* records them in $\mathcal{L}$, then this attack is indistinguishable from a legitimate transaction during audit. While this is permitted by our model, it may be undesirable for some applications. For example, perhaps no single user would have been able to get all the permissions needed to carry out that transaction, due to separation of duty constraints; or perhaps the transaction would have violated an integrity constraint. We leave the detection of this as future work.

## V. Refinement: Hash Page on Read

With a file editor, an adversary can make arbitrary changes to a log-consistent database, as long as she undoes them before the next audit. Such changes cannot be detected by the audit, and so the log-consistent architecture has a query verification interval of infinity: users receive no guarantees that their query results were correct. The effectiveness of this state reversion attack can be greatly reduced by performing unannounced audits at unpredictable intervals.

We can eliminate this vulnerability completely. First, we augment the NEW_TUPLE log records in $\mathcal{L}$ to include the unique page number (PGNO) where the tuple is stored. We also introduce a new type of log record, PAGE_SPLIT, which is appended to the log whenever a page splits. Each PAGE_SPLIT record contains the PGNO of the initial page, the PGNOs of the two new pages created, and the content of the two new pages immediately after the split.

Suppose a transaction $\mathcal{T}$ wants to verify the content of a page $P_{121}$. $\mathcal{T}$ first finds the sequence of pages which were split since the last audit to obtain $P_{121}$. For example, suppose page $P_1$ was split to produce $P_{12}$, which was later split to create $P_{121}$. $\mathcal{T}$ then applies all the non-aborted NEW_TUPLE and STAMP_TRANS operations from $\mathcal{L}$ on the snapshot version of $P_1$, up to the point where it was split to create $P_{12}$. This gives the correct content of $P_1$ at the time of the split. $\mathcal{T}$ then verifies that the split is correct and complete—the union of the tuples on the two new pages $P_{11}$ and $P_{12}$ should be the tuples on $P_1$. $\mathcal{T}$ must also do the usual page integrity checks that an auditor does, e.g., that the tuples are properly sorted and the other page metadata is correct. $\mathcal{T}$ repeats this process for $P_{11}$ and then for $P_{121}$.

This verification process is still too costly to be executed by each transaction, but we can greatly reduce its cost by deferring it to audit time. When the DBMS reads a page $P$ from the disk, the compliance logger hashes $P$'s content (tuples) and records the hash and PGNO in $\mathcal{L}$. The auditor subsequently reconstructs $P$ from $\mathcal{L}$ and compares the hash of the reconstructed $P$ to the hash that $\mathcal{T}$ stored in $\mathcal{L}$. If the two hashes match, then $\mathcal{T}$ read the "correct" copy of $P$.

The reconstruction of page $P$ is slightly different from that used to check the final state of the database, because the auditor wants to reconstruct the page exactly as it was at the moment when its hash was appended to $\mathcal{L}$. In particular, the hash includes any uncommitted tuples on $P$ that are aborted later in $\mathcal{L}$. The auditor hashes each tuple on $P$ with its transaction ID $\mathcal{T}$ if the STAMP_TRANS $P$ record for $\mathcal{T}$ appears later in $\mathcal{L}$; otherwise she hashes the tuple with its commit time. She excludes from the hash all tuples on $P$ that have already been aborted. The auditor can verify all the read operations of all transactions in a single pass, by storing and reconstructing the states of all pages as $\mathcal{L}$ is scanned.

This approach retains the linear time complexity of auditing without hash-page-on-read, but is inefficient when the pages do not fit in main memory. One can reduce the storage requirement through more sophisticated replay techniques that sort the log entries in $\mathcal{L}$ by PGNO and reconstruct one page at a time, but this approach requires sorting the log. Sorting is undesirable when the log is very large.

To avoid sorting, we could hash the tuples on each page using a commutative incremental hash function $H_c$, rather than the incremental hash function $H$ discussed earlier. Unfortunately, secure commutative incremental hash values are long (200+ bytes). With a 32 KB page, the $H_c$ values for a 1 TB database will occupy over 10 GB, and recording the hashes in $\mathcal{L}$ will be costly for transactions that read a lot of data.

To avoid the overhead of $H_c$ while still supporting query verification, we employ a sequential hash function $H_s$ and an additional attribute for each tuple, called its *tuple order*

*number*. When a transaction writes out a new tuple for page $P$, the compliance logger finds the largest tuple order number on that page, increments it, and stores it with the new tuple as its tuple order number. The logger also logs a NEW_TUPLE record on $\mathcal{L}$ as usual, together with the tuple's order number. UNDO activity in pwrite requests may cause gaps in the stored tuple order numbers, but this will not cause a problem with auditing. When a transaction reads $P$, we sort the tuples on $P$ based on their order numbers. Once the tuples are in sorted order $r_1, \ldots, r_n$, the logger uses the following sequential hashing function:

$$H_s(r_1, \ldots, r_n) = H(h(r_1), H(r_2, \ldots, r_n)),$$

where $h$ is any conventional secure hash function (e.g., the 256-bit SHA-1 function). With such a hash function, the page hashes for a 1 TB database with 32 KB pages will occupy only 1 GB.

The compliance logger records the hash of $P$ on $\mathcal{L}$. In our implementation, the compliance logging plugin parses each page that is read by pread and hashes its tuples with $H_s$. A READ record is appended to $\mathcal{L}$, containing the $H_s$ value and PGNO. We do not hash or log transaction reads that hit the buffer cache, which is why we must trust the buffer cache.

The auditor reads the $H_s$ value for each snapshot page from WORM. It then scans $\mathcal{L}$, incrementally updating the $H_s$ value for page $P$ every time it reaches a non-aborted NEW_TUPLE record for $P$, being careful to use the correct commit time or transaction ID for each tuple. Whenever the auditor encounters an $H_s$ value for $P$ in $\mathcal{L}$, it compares it to the $H_s$ value it has computed so far for $P$; a mismatch signals tampering, because $P$'s tuples appear on $\mathcal{L}$ in the order that they were inserted into $P$. Page split records are handled as usual.

The preceding discussion does not consider the effect of aborted transactions. With fine-granularity locking, a transaction $\mathcal{T}_1$ that eventually commits may read tuple $t_1$ on a page $p$ where tuple $t_2$ has been written by another transaction $\mathcal{T}_2$ that eventually aborts. (Note that tuple $t_2$ could not have been read by a transaction that eventually commits. The problem we are addressing here is only that $t_2$ is included in the hash value for page $p$.) In this case, to verify that $\mathcal{T}_1$ read the right content on $p$, the hashes of $p$ computed by $\mathcal{T}_1$ and the auditor must both include $t_2$. As the auditor continues to scan the log and finds the ABORT record for $\mathcal{T}_2$, it will recognize that the NEW_TUPLE record for $t_2$ should not be included in the hash used to check the final database state, as usual. However, the auditor must also determine at which point $t_2$ was removed from $p$ and was no longer visible to transactions. For this purpose, the compliance logger must record additional information in $\mathcal{L}$ for UNDO activities, beyond the simple ABORT records needed to check the final state. Whenever the logger detects that a tuple version has been removed from a page that is being written out, it must log this as an UNDO $t_2$ record on $\mathcal{L}$, together with $p$'s PGNO; it must send this UNDO to WORM before the corresponding data page goes to disk. When the auditor encounters an UNDO record while scanning $\mathcal{L}$, it must "roll back" its hash computation for $p$ to a point in $\mathcal{L}$ just before the NEW_TUPLE $t_2$ record for $p$, then roll forward the hash chain computation from that point, including each subsequent NEW_TUPLE $t_3$ record for $p$ in the hash chain iff there is no subsequent UNDO $t_3$ record for $p$ before the UNDO $t_2$ record in $\mathcal{L}$.

If a crash occurs after an UNDO record has gone to WORM but before the tuple has been removed on disk, then a duplicate UNDO record will be sent to WORM during the UNDO phase of recovery. Duplicate UNDO records do not affect query verification, assuming that new transactions are not admitted until the UNDO phase of recovery is complete.

The compliance plugin also hashes and logs the contents of index pages, so that the auditor can verify that all the index pages read by the transaction were correct. As with the data pages, the plugin computes and logs the hash of every index page read from disk. The plugin also logs the changes in the index pages to $\mathcal{L}$. Specifically, when a data page splits, the plugin logs the new entry that is inserted into the parent index node. Similarly, when an index node splits, the logger records the new element inserted in the parent index node.

These log entries enable the auditor to replay all the operations applied to the index and construct the state of the index at any given time. This replaying is efficient because during audit, the index interior nodes typically fit in memory. The auditor computes the hashes of the index pages as it reconstructs them, and compares those hashes to the ones recorded in the log.

## VI. Refinement: WORM Migration

The log-consistent compliant database architecture and its hash-page-on-read refinement require the auditor to check the integrity of every page. Since a temporal DB records all the different versions of a record, the database size and hence the auditing effort grows with time. The WORM migration refinement of the log-consistent architecture, introduced in this section, addresses this problem by migrating historical versions of tuples onto WORM, placing them in a separate append-only file on WORM. This migration must be done carefully, as the leaf pages of a conventional B$^+$-tree in a temporal DB contain a mixture of live and historical records; such leaf pages cannot be moved wholesale to WORM storage. We utilize *time-split B$^+$-trees* [15] to obtain leaf pages that only have historical records.

A second concern is that the auditor must verify that the tuples have been migrated properly. To do so we reuse the techniques used to verify that pages have been split correctly. As a result, once a page of tuples is committed to WORM storage, these tuples can be ignored during subsequent audits.

A tuple indexed in a time-split B$^+$-tree is identified by a $(k, t)$ pair, where $k$ is the tuple's key value and $t$ is the timestamp or start time for that tuple in the DB. The tree defines an ordering between two keys $(k_1, t_1)$ and $(k_2, t_2)$, as $(k_1, t_1) \leq (k_2, t_2)$ iff $(k_1 \leq k_2) \ \vee \ ((k_1 = k_2) \ \wedge \ (t_1 \leq t_2))$.

Time-split B$^+$-trees are like normal B$^+$-trees, except that leaf nodes can split on key *or time* values [15]. Key splits are handled in the same way as for B$^+$-tree node splits: using the ordering defined above, all index entry tuples where $k$ is less

than the splitting value are copied to one node, while entries with larger keys are copied to the other node. An entry with the splitting key is inserted into the parent index node.

Time splits are governed by the splitting time $t$. All versions of the data tuples that are only valid before time $t$ (that is, for which there exists a subsequent version with a timestamp before $t$) have their entries moved to the *historical page*, while the entries for tuples that were valid after time $t$ are copied to the *live page*. Entries for tuples whose period of validity overlaps $t$ are copied onto both the pages, by creating an intermediate version at time $t$. The historical page is then stored on WORM. For example, the index entry $(31, 4)$ overlaps the splitting time 5. Hence, a new tuple $(31, 5)$ is created and included on the live page. Time splits help merge recent versions of records into a smaller number of live pages. The historical pages will never be split again, and hence can be put on WORM. At the next audit, the auditor uses the PAGE_SPLIT records in $\mathcal{L}$ as usual to verify that the split was carried out properly. Then the historical pages on WORM can be exempted from future audits.

The decision whether to split a node on key or time is based on the *split-threshold* parameter. If the number of distinct keys in a leaf page is less than the split-threshold fraction of the total number of tuples, the page is split on keys; otherwise it is split on time. The intuition here is that pages with a small number of unique keys (and hence a large number of updates to those keys) should be time-split to move the old records to WORM. If a page has many distinct keys (and hence few updates), splits on time are ineffective.

## VII. Empirical Evaluation

We evaluated the log-consistent architecture using the TPC-C benchmark implemented atop Berkeley DB. Berkeley DB is a highly optimized, open source implementation of the lower levels of a DBMS stack—a transaction engine, a logging and locking infrastructure, and a $B^+$-tree implementation. The remaining layers of the stack, such as a query optimizer and processor, are not part of the system. We chose TPC-C because it is a standard benchmark for OLTP, which will be the most common workload for compliance databases. We ported the Shore TPC-C implementation to work with Berkeley DB and implemented the compliance logging features, including time-split $B^+$-trees. Except where otherwise mentioned, we set the number of warehouses in the benchmark to 10, which results in a 2.5 GB database. The hash-page-on-read and migrate-to-WORM refinements require storing the tuple order number with each tuple in the database. We modified the TPC-C schema to include this additional attribute for each relation.

Our hardware platform was a 32 bit, 3 GHz Intel Pentium machine with 512 KB L2 cache and 1 GB memory. The database was created on an NFS mounted Network Appliance NFS file server. All the logs, including the database transaction log and the compliance log, were created on this remote server. This models the hardware configuration in most enterprises, where the database server and the storage server are hosted on separate machines. Unless otherwise stated, the buffer cache size was set to 256 MB in our experiments.

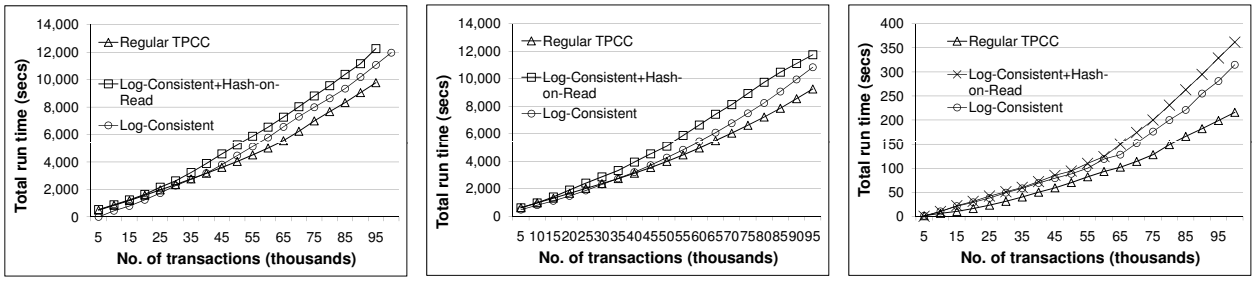
*a) Space Overhead:* Log $\mathcal{L}$ on WORM contains all inserted tuples, hence grows with the number of transactions. After 100K TPC-C transactions, $\mathcal{L}$ was approximately 100 MB. The hash-page-on-read refinement appends page hashes to $\mathcal{L}$. After 100K TPC-C transactions with a 256 MB (resp. 32 MB) buffer cache of 4 KB pages, these logged hashes occupied 3 MB (resp. 44 MB). The number of read operations and hence the number of logged hashes increases as the DBMS buffer cache size decreases. The hash-page-on-read approach also stores a PGNO field (4 bytes) with each log entry and a tuple sequence number (2 bytes) with each tuple. The space overhead of these was under 10%.

The WORM migration refinement slightly increases the total space occupied by the database. For example, the STOCK relation occupies 70,000 pages when the tuples are stored in a normal $B^+$-tree. A time-split $B^+$-tree with a splitting threshold of 0.5 occupies 18,000 live pages and 55,000 historical pages. This additional space is acceptable, given that the WORM pages need not be scanned during audits.

*b) Run Time Overhead:* Figure 3(a) compares the TPC-C run time with native Berkeley DB, the log-consistent architecture, and its hash-page-on-read refinement as a function of the number of transactions. The log-consistent architecture incurs the overhead of computing the difference between the in-memory page and on-disk page, while the hash-page-on-read refinement also hashes each page read from disk. These schemes also incur the overhead of sending dirty pages to disk once every regret interval (set to 5 minutes). Instead of using the more sophisticated scheme described earlier in this paper, we implemented this feature by calling db_checkpoint once every regret interval, which is slower but less intrusive. As evident from the figure, the log-consistent architecture slows down transaction processing by approximately 10%. If we also support verification of the contents of each page read by a transaction, processing is slowed by approximately 20%, compared to native Berkeley DB.

We studied the effect on performance of varying the cache and DB size. Figure 3(b) shows the overhead with a 512 MB DBMS buffer cache. Figure 3(c) shows the overhead for the extreme case of an almost memory resident database of 320 MB (1 warehouse) and 256 MB buffer cache size. The impact of the log-consistent architecture is more profound here because the DBMS accumulates many dirty pages that must be written to disk. Still, the slowdown was under 30% on average, even after the knee of the curve where the data no longer fits in memory.

*c) Audit Time:* Another important parameter is the time the auditor takes to verify the integrity of the DBMS. For the log-consistent architecture, the total time to compute the incremental hash $H_s$ over the previous snapshot, log entries corresponding to 100K TPC-C transactions, and the final state were 121, 85, and 145 sec., respectively, for a total of 351 sec. For the hash-page-on-read refinement, the additional audit time was 104 sec., which includes incrementally computing the




(a) 10 Warehouses, 256 MB DBMS Cache

(b) 10 Warehouses, 512 MB DBMS Cache

(c) 1 Warehouse, 256 MB DBMS Cache

Fig. 3. Effect of the log-consistent architecture on TPC-C total run time

hash of all the pages and comparing the computed hashes to the page read hashes logged by all transactions. The audit time is tiny compared to the 2-3 hours to execute the transactions.

*d) Evaluation of Time-split $B^+$-trees:* We evaluated the performance impact of the threshold that determines whether to execute a time or key split in a $B^+$-tree. Figures 4(a) and 4**(b)** plot the number of live pages and the number of WORM pages at the end of 100K transactions for two of the largest relations of the TPC-C benchmark, STOCK and ORDER_LINE. STOCK had 400K updates for 100K tuples, and ORDER_LINE had 118K updates for 100K tuples.

The number of WORM pages for the STOCK relation is high even at a low threshold value (even 0). This is because the tuple updates in the STOCK relation are highly skewed; only a few tuples (the popular items) are updated, and these many times. The leaf pages containing these heavily updated records are time-split even for low threshold values. The dip in the number of live pages and the increase in the number of historic pages at the threshold value of 0.5 is because most of the $B^+$-tree pages for STOCK are half filled at the end of DB setup. The fraction of distinct keys in the $B^+$-tree leaf nodes is close to 0.5. If the split-threshold is more than 0.5, these pages are time-split, otherwise key-split. The number of leaf pages in a regular $B^+$-tree for STOCK was close to 77K. The corresponding time-split $B^+$-tree had only 15K live leaf pages. This drop in the number of live pages reduces the auditor's effort by the same fraction, since only live pages are audited.

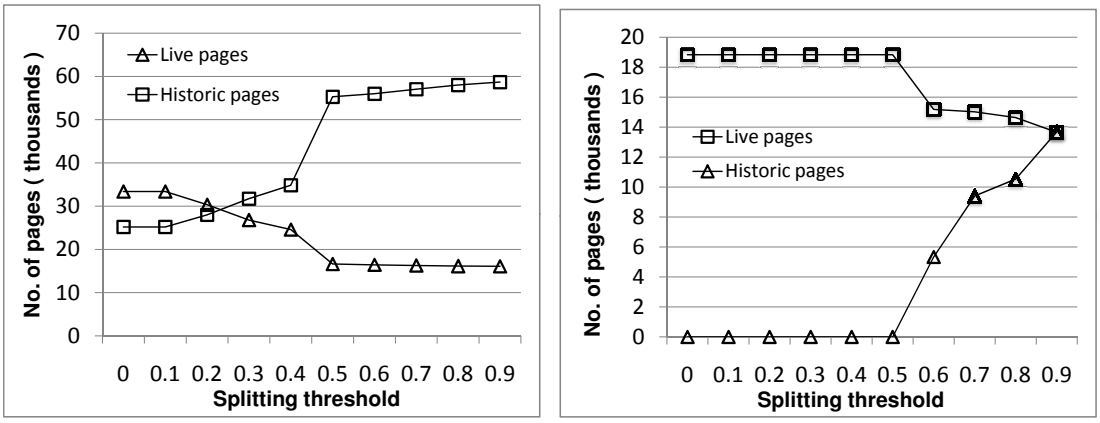
TPC-C updates the tuples in the ORDER_LINE relation uniformly, with each tuple being updated at most once. Thus the number of distinct keys in any leaf node is always more than half of the total. Hence, no nodes move to WORM if the splitting threshold is under 0.5. Higher thresholds gradually decrease the number of live pages but rapidly increase the number of historic pages. For example, suppose the threshold increases from 0.8 to 0.9. Then all the leaf pages with 80-90% unique keys are time-split. After a leaf is time-split, it has the same number of distinct keys after the split as before the split. Hence, it can be time-split again when those keys are updated: if the node had 90% distinct keys and each key is updated once, the page can be time-split 9 times. On the other hand, a page can be key-split only once because the resulting half-filled pages have enough space to accommodate all future updates. Thus increasing the threshold from 0.8 to 0.9 replaces one key-split by 9 time-splits per page.

For both relations, it is optimal to choose a threshold close to the fill factor of the initial relation.

## VIII. SHREDDING

Our approach to shredding expired tuples assumes that auditing is done sufficiently frequently (e.g., once a year) that every tuple committed to the database will be retained through at least one audit. We run a periodic vacuuming process [21], [27] to physically erase expired database tuples. So that an adversary cannot simply vacuum up tuples that she regrets, the auditor must verify that vacuumed tuples really had expired. For this purpose, we store an Expiry relation that records the expiration time of each data tuple. The best way to implement the Expiry relation depends on the application domain. In the simplest case, the Expiry relation contains a single tuple giving the amount of time that all database records must be retained. At the other extreme, the Expiry relation may list an exact expiration time for every tuple in every relation. For current regulations, it usually suffices to remember a single retention period per relation, and we take that approach in the remainder of this section. In addition to verifying the contents of Expiry as for any other transaction-time relation, the auditor will need to ensure that any changes to the retention periods in Expiry are consistent with current regulations.

When the vacuum process runs, it must append to $\mathcal{L}$ a timestamped SHREDDED record, listing the tuple ID (e.g., relation name, key, and START time), its PGNO, and a hash of the tuple contents, for every tuple it wishes to vacuum. The SHREDDED record must be sent to WORM before the tuple(s) listed on it can be vacuumed. When a page is vacuumed, the alterations are captured and logged on $\mathcal{L}$ when the page is written out, just as for ordinary page writes; the logger will record the vacuuming activity in ordinary READ and UNDO records. After a crash, the compliance routines need to finish vacuuming any tuples that are listed in a SHREDDED record on $\mathcal{L}$, but are still in the DB; the simplest




(a) STOCK Relation

(b) ORDER_LINE Relation

Fig. 4. Effect of the splitting threshold on the number of live and historic pages

implementation is just to re-vacuum after recovery. All tuples listed in SHREDDED records must be vacuumed before the next audit, or the audit will fail. The SHREDDED record and the previous snapshot still contain evidence that a shredded tuple did exist in the past, and that evidence is a liability. However, the SHREDDED records and the old snapshot will be destroyed after the next audit, and then the tuple will truly cease to exist. Other evidence of the tuple's past existence may be visible if one carefully examines the internals of the DB and DBMS, and this evidence should also be scrubbed away using techniques developed by others [25].

Shredding changes the auditor's algorithms. When the auditor computes the page hashes on the final database state, vacuumed tuples will not be included; but the vacuumed tuples are present in the previous snapshot. For this reason, the auditor cannot simply reuse the snapshot hashes computed at the last audit. Instead, the auditor must find the SHREDDED records in $\mathcal{L}$ (the techniques described earlier for quick processing of ABORT records can be used to make this fast), and recompute the hash for every snapshot page that has been vacuumed.

The auditor can use its usual algorithm for checking the READ records in $\mathcal{L}$ under the hash-page-on-read refinement. The vacuumed tuples appear in UNDO records, which will trigger the auditor to include or exclude them in its hashes as the auditor scans $\mathcal{L}$. However, during its scan of $\mathcal{L}$, the auditor does need to verify that every UNDO record it scans is listed in a previous ABORT record or a previous SHREDDED record. The auditor also verifies that every shredded tuple indeed had expired, by checking its commit time in the previous snapshot and verifying that the retention period in Expiry had ended before the time recorded in the timestamp on the tuple's SHREDDED record. Evidence of the shredded tuples will remain visible until the next audit is complete, at which point the compliance log and previous snapshot are removed and the shredded tuples really are gone.

When WORM migration is used, many expired tuples may reside on WORM and their pages must be migrated back to regular media for shredding. This re-migration must be documented on the compliance log in the same manner as for the initial migration to WORM, and verified by the auditor. After vacuuming, the page can then be migrated back to WORM if desired. Note, however, that the unit of deletion on WORM is an entire file. Thus one cannot truly delete a page on WORM until the file in which it resides has expired. For this reason, the migration of time-split tuples and index entries to WORM will be most effective if all the migrated data in the file will expire at approximately the same time. Then all the pages in the file can be vacuumed at the same time, so the entire file can be deleted at once.

## IX. Conclusion and Future Work

Current technology does not allow any strong compliance guarantees for database data. Extending the level of protection afforded by existing record management systems to the vast amounts of structured data residing in databases is a challenging research problem. The write-once nature of WORM storage devices makes them very resistant to the insider attacks at the heart of compliance regulations, but also makes it very hard to lay out, update, index, query, update, delete, and shred tuples efficiently. Thus it is especially challenging to provide trustworthiness while also retaining the classical scalable performance guarantees of a DBMS.

In this paper we presented the log-consistent architecture for term-immutable databases, which keeps a transaction-time database and transaction log on conventional storage, along with an auxiliary *compliance log*, the tail of the transaction log, and a snapshot from the previous audit on inexpensive WORM storage. The compliance log tracks the history of modifications to the database contents in such a manner that in a single-pass algorithm, an auditor can determine whether the current state of the database is consistent with all past modifications. Database crashes provide an excellent opportunity for the

adversary to tamper with the database or log contents, but the details of how the compliance log is written ensure that these attacks will be discovered and thwarted during crash recovery.

For applications where it is important to be able to verify that queries did not access tampered data, we presented a hash-page-on-read refinement that logs a hash of each page read from disk; the auditor can verify that all pages had the correct contents, using a single-pass algorithm. Because the cost of an audit increases as the database size grows, we provided a WORM-migration refinement that moves tuples of historical interest and their index entries to WORM storage, which exempts them from future audits. We also provided a way to shred expired tuples, using an auditable vacuuming process that documents its activities on the compliance log.

We implemented the log-consistent architecture and its refinements atop Berkeley DB, in a manner that involved very few changes to the DBMS core; most of the compliance functionality is isolated in a plugin that is invoked on each pread/pwrite request, with additional functionality inside the crash recovery component. Our experiments with a 10-warehouse configuration of TPC-C atop Berkeley DB showed that the log-consistent architecture imposes an approximately 10% runtime overhead in typical configurations, with reasonable space requirements on WORM. If we also wish to support verification of the contents of the database pages read by each query, that adds an additional 10% to the overhead. These results suggest that the log-compliant architecture has the potential to allow more effective application of existing laws and regulations mandating how business data should managed, thereby increasing societal confidence in business, government, and personal data. Currently, we are working on support for "litigation holds", which ensure that subpoenaed but expired tuples are not shredded.

## X. Acknowledgements

This work was supported by an IBM PhD Fellowship and by NSF under CNS-0803280, CNS-0716532, and CNS-0524695. We thank Nikita Borisov for helpful discussions regarding hash functions.

## References

[1] Ahmed Ataullah, Ashraf Aboulnaga, and Frank W. Tompa, "Records Retention in Relational Database Systems," in *CIKM*, 2008.

[2] Christopher Batten, Kenneth Barr, Arvind Saraf, and Stanley Treptin, "pStore: A Secure Peer-to-Peer Backup System." Online at `http://citeseer.ist.psu.edu/batten01pstore.html`.

[3] Mihir Bellare and Daniele Micciancio, "A New Paradigm for Collision-free Hashing," in *Eurocrypt*, 1997.

[4] Matt Blaze, "A Cryptographic File System for UNIX," in *CCS*, 1993.

[5] Philip Bohannon, Rajeev Rastogi, S. Seshadri, Avi Silberschatz, and S. Sudarshan, "Using Codewords to Protect Database Data from a Class of Software Errors," in *ICDE*, 1999.

[6] B. Chen and R. Morris. "Certifying Program Execution with Secure Processors," in *USENIX HotOS Workshop*, 2003.

[7] Andrew Conry-Murray, "Appetite for Destruction," *Information Week*, pp. 27–33, June 9, 2008.

[8] Financial Executives International's Survey on Costs and Benefits of SOX, 2007. Online at `http://fei.mediaroom.com/index.php?s=43&item=204`.

[9] P. A. Gerr, B. Babineau, and P. C. Gordon, "Compliance: The Effect on Information Management and the Storage Industry," Enterprise Storage Group Technical Report, May 2003, `http://www.enterprisestrategygroup.com/ESGPublications/ReportDetail.asp?ReportID=201`.

[10] Eu-Jen Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh, "SiRiUS: Securing Remote Untrusted Storage," in *NDSS*, 2003.

[11] Hakan Hacigumus, Balakrishna R. Iyer, and Sharad Mehrotra, "Providing Database as a Service," in *ICDE*, 2002.

[12] Hakan Hacigumus, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra, "Executing SQL over Encrypted Data in the Database Service Provider Model," in *SIGMOD*, 2002.

[13] Christian S. Jensen and Curtis E. Dyreson (eds), A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in *Temporal Databases: Research and Practice*, Ophir Etzion, Sushil Jajodia, and Suri Sripada (eds.), Springer-Verlag, pp. 367–405, 1998.

[14] Mahesh Kallahalla, Erik Riedel, RamSwaminathan, Qian Wang, and Kevin Fu, "Plutus—Scalable Secure File Sharing on Untrusted Storage," in *USENIX Conference on File and Storage Technologies*, 2003.

[15] David B. Lomet and Betty Salzberg, "The Performance of a Multiversion Access Method," in *SIGMOD*, 1990.

[16] David B. Lomet, Roger S. Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, Yunyue Zhu, "Transaction Time Support Inside a Database Engine," in *ICDE* 2006.

[17] Kyriacos Pavlou and Richard T. Snodgrass, "Forensic Analysis of Database Tampering," in *SIGMOD* 2006.

[18] Betty Salzberg, "Timestamping After Commit," in *Proceedings of the Conference on Parallel and Distributed Information Systems*, pp. 160–167, 1994.

[19] Bruce Schneier and John Kelsey, "Secure Audit Logs to Support Computer Forensics," *ACM Transactions on Information System Security* 2(2):159–176, 1999.

[20] Radu Sion, "Query Execution Assurance for Outsourced Databases," in *VLDB*, 2005.

[21] Janne Skyt, Christian S. Jensen, and Leo Mark, "A Foundation for Vacuuming Temporal Databases," *Data and Knowledge Engineering* 44(1):1–29, 2003.

[22] Sleepycat Software Inc., *Berkeley DB*, 2001.

[23] Richard T. Snodgrass, *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann, 1999.

[24] Richard T. Snodgrass, Stanley Shilong Yao and Christian Collberg, "Tamper Detection in Audit Logs," in *VLDB*, 2004.

[25] P. Stahlberg, G. Miklau, and B. N. Levine, "Threats to Privacy in the Forensic Analysis of Database Systems," in *SIGMOD*, 2007.

[26] Michael Stonebraker, "The Design of the POSTGRES Storage System," in *VLDB*, 1987.

[27] Michael Stonebraker and Lawrence Rowe, "The Design of Postgres," in *SIGMOD*, 1986.

[28] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. "Aegis: Architecture for Tamper-evident and Tamper-resistant Processing," in *ICS*, 2003.

[29] Mark Sullivan and Michael Stonebraker, "Using Write Protected Data Structures to Improve Software Fault Tolerance in Highly Available Database Management Systems," in *VLDB*, 1991.

[30] U.S. Dept. of Health and Human Services, The Health Insurance Portability and Accountability Act (HIPAA), 1996. Online at `http://www.cms.hhs.gov/HIPAAGenInfo/`.

[31] U.S. Food and Drug Administration, "Guidance for Industry Part 11, Electronic Records; Electronic Signatures and Application," August 2003. Online at `http://www.fda.gov/cder/gmp/index.htm`.

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

[33] The U.S. Securities and Exchange Commission, Rule 17a-4, 17 CFR Part 240: Electronic Storage of Broker-Dealer Records, 2003. Online at `edocket.access.gpo.gov/cfr_2002/aprqtr/17cfr240.17a.htm`.

[34] Ari Weinberg, "Wall Street Fine Tracker," *Forbes*, July 15, 2004. Online at `http://www.forbes.com/2002/10/24/cx_aw_1024fine.html`.

[35] Windows Kernel Patch Protection. Online at `http://www.microsoft.com/whdc/driver/kernel/64bitpatch_FAQ.mspx`.